Chap H : Introduction à la programmation de Genesis

Plan

I. Initialisation

Après avoir lancé Genesis, on a accès aux commandes du SLI (Script Language Interpreter). Dans l'interpréteur, il est possible d'effectuer des commandes Linux (compatibles UNIX) et des commandes Genesis.

```
genesis #2 > ls
AUTHORS  Hyperdoc  README   Usermake    convert  lib  rallpack  startup
Doc      Libmake   Scripts  VERSION_2.1  include  man  src
```

ls permet d'afficher la liste des fichiers présents dans le répertoire courant.

```
genesis #3 > ls -a
.          .nxsimrc  AUTHORS   Libmake  Usermake     include  rallpack
..         .simrc    Doc       README   VERSION_2.1  lib      src
.minsimrc  .simrc~   Hyperdoc  Scripts  convert      man      startup
```

ls -all permet d'afficher *tous* les fichiers dans le répertoire courant.

```
genesis #5 > cd ..
```

cd .. permet de remonter d'un cran dans l'arborescence des fichiers présents sur le disque.

```
genesis #1 > pwd
/home/rodrig/genesis
```

pwd permet de connaitre le chemin d'accès au répertoire courant.

```
genesis #9 > listcommands

Available commands :
abort          abs          acos          addaction
addalias       addclass     addescape     addfield
addforwmsg     addjob       addmsg        addmsgdef
addobject      addtask      affdelay      affweight
argc           arglist      argv          asciidata
```

```
asin            atan            balanceEm       calcCm
calcRm          call            callfunc        cd
ce              cellsheet       check           chr
clearbuffer     clearerrors     closefile       connect .....
```

listcommands est un ordre *genesis* produisant une liste de toutes les commandes possibles en utilisant le langage *SLI*.

Il est possible d'utiliser des commandes *Linux* combinées avec des commandes *genesis*.

```
genesis #17 > listcommands | more

Available commands :
abort           abs             acos            addaction

addalias        addclass        addescape       addfield

addforwmsg      addjob          addmsg          addmsgdef

addobject       addtask         affdelay        affweight
argc            arglist         argv            asciidata
–Encore–
```

listcommands | more permet d'afficher la liste des commandes de *genesis* avec la possibilité de controler le défilement par l'ordre *more*. Pour poursuivre le défilement, taper *Return*.

```
genesis #5 > listcommands > lscomm.txt
```

listcommands > monfichier.txt envoie le texte produit par *listcommands* dans un fichier intitulé *monfichier.txt*

II. Les éléments et objets de Genesis

Les composants de base utiles pour construire des simulations sont appelés *éléments*. Ces derniers sont créés à partir de structures appelées *objets* ou bien *types élémentaires*. La liste des objets disponibles peut être visualisée par l'ordre *listobjects*

```
.genesis #6 > listobjects

AVAILABLE OBJECTS :
Ca_concen        Kpores          Mg_block        Napores
PID              RC              asc_file        axon
axonlink         channelA        channelB        channelC
channelC2        channelC3       compartment     concchan
concpool         ddsyn           defsynapse      dif2buffer
difbuffer        diffamp         difshell        disk_in
disk_out         diskio          efield          enz
expthresh        fixbuffer       freq_monitor    funcgen
fura2            ghk             graded          hebbsynchan
hh_channel       hsolve          interspike      leakage      .....
```

Pour avoir une information plus détaillée sur un *objet* , taper *showobject <nomobjet>*

```
genesis #11 > showobject compartment

object          = compartment
datatype        = compartment_type
function        = Compartment().....

          x0              0
          y0              0
          z0              0
          activation      0
          Vm              0       compartment potential
          previous_state  0
          Im              0
```

Em	0	resting potential
Rm	1	membrane resistance
Cm	1	membrane capacitance
Ra	0	axial resistance
inject	0	injected current
dia	0	compartment diameter
len	0	
initVm	0	initial Vm value at reset

L'objet *compartment* (compartiment) est très utilisé dans les simulations de Genesis pour construire des modèles de neurones. Les valeurs de *Ra, Rm, Cm,inject,Vm* caractérisant cet objet sont relatives au modèle de circuit élémentaire tel que celui qui est représenté ci dessous.

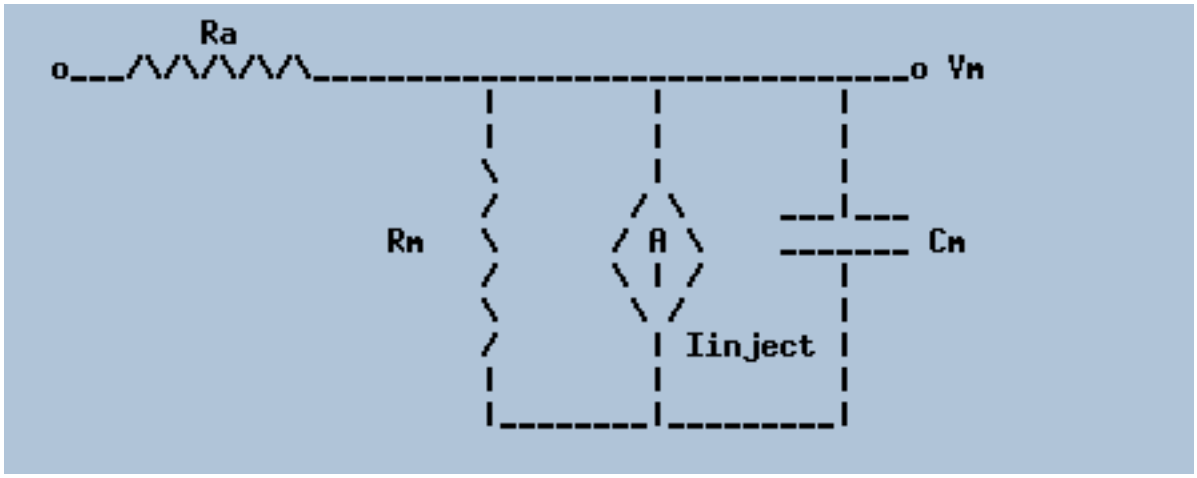


figure 1

Pour avoir une information plus détaillée sur un objet, taper *help nomobjet | more*

genesis #2 > help compartment | more

Object Type : compartment

Description : Axially asymmetric compartment. Ra is located on
one side of the compartment. This is slightly more
computationally efficient than the symmetric counterpart. .....

```
Fields :        Rm              total membrane resistance
                Cm              total membrane capacitance
                Em              membrane resting potential
                Ra              axial resistance
                inject          injected current in membrane
                dia             compartment diameter
                len             compartment length
                Vm              voltage across the membrane
                previous_state  Vm at previous time step
                Im              total membrane current
                initVm          initial value to set Vm on reset          .....
```

III. Créer et détruire des éléments

Pour créer un élément d'un *type* donné (c'est à dire correspondant à un objet de base), on utilise l'ordre *create*. Une aide en ligne permet de préciser l'*usage* de cet ordre. Pour cela, il siffit de taper *create* sans arguments.

```
genesis #7 > create
create : too few command arguments
usage : create object name -autoindex [object-specific-options]
```

On va créer un compartiment appelé soma

```
genesis #9 > create compartment /soma
OK
```

Les éléments sont maintenus dans une hiérarchie, comme cela est le cas dans l'organisation des fichiers et répertoires du système *Linux (UNIX).*

Ici, */soma* avec / indique que le soma est placé à la "racine " (en haut) de la hiérachie. On pourra construire des (modèles de) neurones appelés par exemple */celluleL11* composés d'un soma *somaL11* (par exemple), de dendrites *dendsL11* pourvues de canaux *Ex_ canaux* et d'un axone *axoneL11*. Il suffira de créer des compartiments dans une hiérarchie de la forme */celluleL11/somaL11, /celluleL11/axoneL11, /celluleL11/dendsL11, /celluleL11/dendsL11/Ex_ canaux*

Pour cela, il sera nécessaire de créer l'élément */celluleL11* d'un type approprié. Genesis procure un objet à cet effet dénommé *neutral*. Un élément de ce type est un élément vide, qui n'effectue aucune action et qui sert généralement d'élément parent auquel vont se greffer de nombreux autres objets. Un exemple de construction pourrait être :

```
genesis #0 > create neutral /celluleL11
OK
genesis #1 > create compartment /celluleL11/somaL11
OK
```

On peut également éliminer un élément par l'ordre *delete*

```
genesis #2 > delete /celluleL11/somaL11
OK
```

IV. Examiner et modifier des éléments

Les commandes permettant de se déplacer dans la hiérarchie des éléments créés par Genesis sont analogues à celles existant dans le système *Linux*. Par exemple, la commande permettant de faire une liste des éléments créés au niveau courant de la hiérarchie est *le* (list elements)

```
genesis #3 > le
*proto                          output
celluleL11
```

Chaque élément contient des *champs (fields)* qui contiennent les valeurs des paramètres et des variables d'état utilisés par l'élément. Pour connaître le contenu de ces champs, on utilise la commande *showfield nomélément* suivi de *-all* (en abrégé *-a*) si l'on désire connaître les valeurs numériques de tous les champs. Sinon, il suffit de lancer la commande *showfield nomélément nomduchamp*

```
genesis #10 > create compartment /celluleL11/dends
OK
genesis #11 > showfield /celluleL11/dends -all
```

```
[ /celluleL11/dends ]
x0y0z0      = ( 0.000000e+00 , 0.000000e+00 , 0.000000e+00 )
xyz         = ( 0.000000e+00 , 0.000000e+00 , 0.000000e+00 )
flags       = 0
FUNCTIONAL
Clock [ 0 ] = 1.000000e+00
0 incoming messages
0 outgoing messages
-------------------------------

activation          = 0
Vm                  = 0
previous_state      = 0
Im                  = 0
Em                  = 0
Rm                  = 1
Cm                  = 1
Ra                  = 0
inject              = 0
dia                 = 0
len                 = 0
initVm              = 0
```

Les champs (*fields*) sont ici *Vm, Rm, Cm, Ra, len, initVm,....*
Pour connaître la valeur de *Rm :*

```
genesis #12 > showfield /celluleL11/dends Rm

[ /celluleL11/dends ]
Rm                  = 1
```

Quand on éxécute une simulation dans Genesis, on se trouve localisé à un certain niveau de la hiérarchie qui est dénommé l'élément de travail : *working element*. Certaines commandes nécessitent de conaître la localisation de ce niveau. Par exemple, la commande *le* doit, en principe être suivi du *chemin* d'accès à l'élément dont on désire faire la liste des objets affiliés.

```
genesis #14 > le /celluleL11
dends
```

Quand le *chemin* est omis, le *working element* est utilisé comme *chemin,* et *le* établit une liste de tous les éléments qui sont localisés sous ce *working element.*

```
genesis #15 > le
dends
```

Pour se déplacer dans la hiérarchie, on utilise l'ordre *ce* (change element). Par exemple, pour changer le *working element* (élément de travail) */celluleL11* en */celluleL11/soma* que l'on vient de (re)créer, taper

```
genesis #19 >  create compartment /celluleL11/somaL11
OK
genesis #25 > ce somaL11
```

Si l'on connait la nature de l'élément de travail, il suffira de taper *showfield -all* sans arguments pour avoir accès à toutes les valeurs des paramètres et variables d'état disponibles dans cet élément.

```
 genesis #27 > showfield -all

[ /celluleL11/somaL11 ]
x0y0z0     = ( 0.000000e+00 , 0.000000e+00 , 0.000000e+00 )
xyz        = ( 0.000000e+00 , 0.000000e+00 , 0.000000e+00 )
flags      = 0                    .....
```

On peut connaître le *working element* courant en tapant *pwe* (print working element). C'est l'équivalent *genesis* de *pwd* de *UNIX.*

```
genesis #28 > pwe
/celluleL11/somaL11
```

Par analogie avec *UNIX, Genesis* utilise le point "." pour signifier le *working element* et le double point ".." pour désigner l'élément situé un cran au dessus du *working element* dans la hiérarchie.

```
genesis #32 > ce .
genesis #33 > pwe
/celluleL11/somaL11
genesis #34 > ce ..
genesis #35 > pwe
/celluleL11
```

Il existe des commandes analogues à *pushd* et *popd* de *UNIX* qui sont *pushe* et *pope*. Ceci procure un moyen commode de changer de localisation, dans la hiérarchie, vers un nouveau *working element* par *pushe* et ensuite de retourner vers le *working element* précédent.

```
genesis #35 > pwe
/celluleL11
genesis #36 > pushe /celluleL11/somaL11
/celluleL11/somaL11
genesis #37 > pwe
/celluleL11/somaL11
genesis #38 > pope
/celluleL11
genesis #39 > pwe
/celluleL11
```

On peut changer le contenu des champs (*fields*) par la commande *setfield*. Par exemple, pour changer la valeur de *Rm* de /celluleL11/somaL11 :

```
genesis #39 > pwe
/celluleL11
genesis #40 > setfield /celluleL11/somaL11 Rm 10
OK
```

Plusieurs valeurs peuvent être modifiées en même temps

```
genesis #41 >  setfield /celluleL11/somaL11 Rm 20 inject 10
OK
genesis #42 > showfield  /celluleL11/somaL11 -all
..
```

```
Im              = 0
Em              = 0
Rm              = 20
Cm              = 1
Ra              = 0
inject          = 10
..
```

V. Exécuter une simulation Genesis

La commande *check* permet d'effectuer un test sur la cohérence du texte enregistré pour le programme. Ensuite, avant de débuter la simulation, les éléments doivent être placés dans un état initial connu. Ceci est fait à l'aide de l'ordre *reset.* L'éxécution de la simulation fait appel à l'ordre *step* suivi d'un nombre indiquant le nombre de pas à effectuer

```
genesis #43 > check
genesis #44 > reset
time = 0.000000 ; step = 0
genesis #45 > step 10
time = 10.000000 ; step = 10
completed 10 steps in 0.000000 cpu seconds

genesis #46 > showfield  /celluleL11/somaL11 -all

[ /celluleL11/somaL11 ]
..
Vm              = 78.69386806
previous_state      = 72.47436968
..
```

L'affichage de ces valeurs indique que le potentiel de */celluleL11/somaL11,* qui est décrit par un circuit électrique représenté dans la figure 1, est passé de la valeur $initVm = 0$ à la valeur $Vm = 78.69386806\ (mV)$. La variable *Vm* qui a des valeurs se modifiant au cours de la simulation est appelée *variable d'état.* Ces variables ne peuvent être modifiées par *setfield.*

VI. Introduire des prahiques

Genesis permet d'envoyer les résultats d'une simulation dans des fichiers pour leur exploitation. Il existe également une possibilité de visualiser le déroulement d'une simulation au cours de son éxécution.

A titre d'exemple, considérons le cas où on désire visualiser les variations de potentiel de */celluleL11/somaL11* qui a été créé plus haut. La mise en place de graphiques se fait en utilisant des objets graphiques appartenant à XODUS, la bibliothèque des objets graphiques de Genesis.

L'objet de base est la *xform,* sur laquelle vont se greffer d'autres objets. Ce sont les "fenêtres" ("windows"). Pour créer une *xform,* on utilise l'ordre *create*. Un nom est donné à la fenêtre (xform), par exemple *donnees* (s'abstenir de mettre des accents !)

genesis #47 > create xform /donnees
OK

La fenêtre n'apparaît pas à l'écran. Pour la visualiser, taper *xshow /donnees.* Une fenêtre vide va apparaître

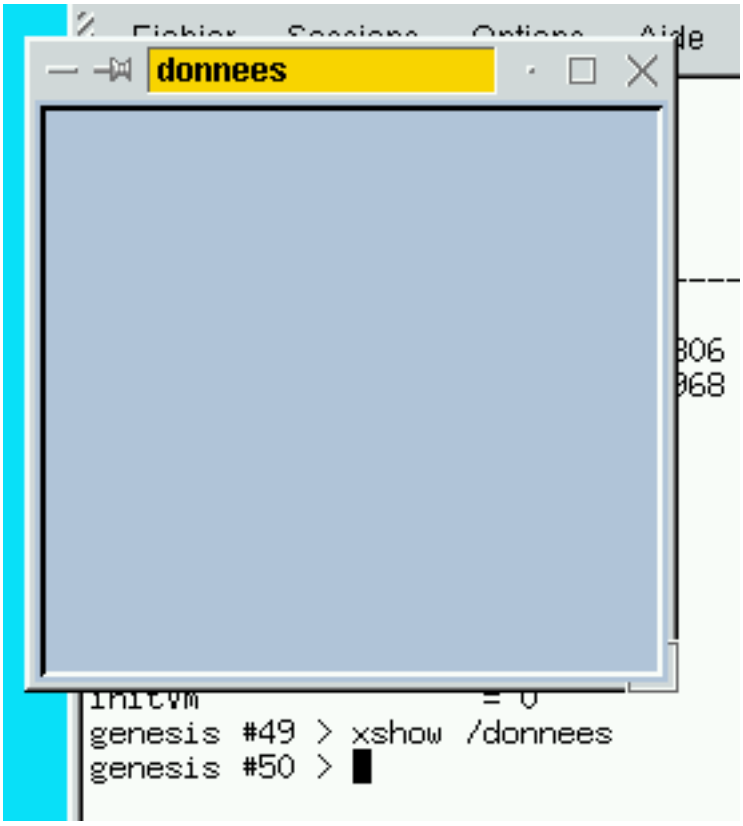


figure 2

Pour créer le graphe du potentiel de */celluleL11/somaL11* dans cette fenêtre, avec le nom *potentiel,* on utilise la commande *create xgraph* . Ce graphe sera affilié à la fenêtre précédemment créée. La syntaxe est alors :

genesis #50 > create xgraph /donnees/potentiel
OK

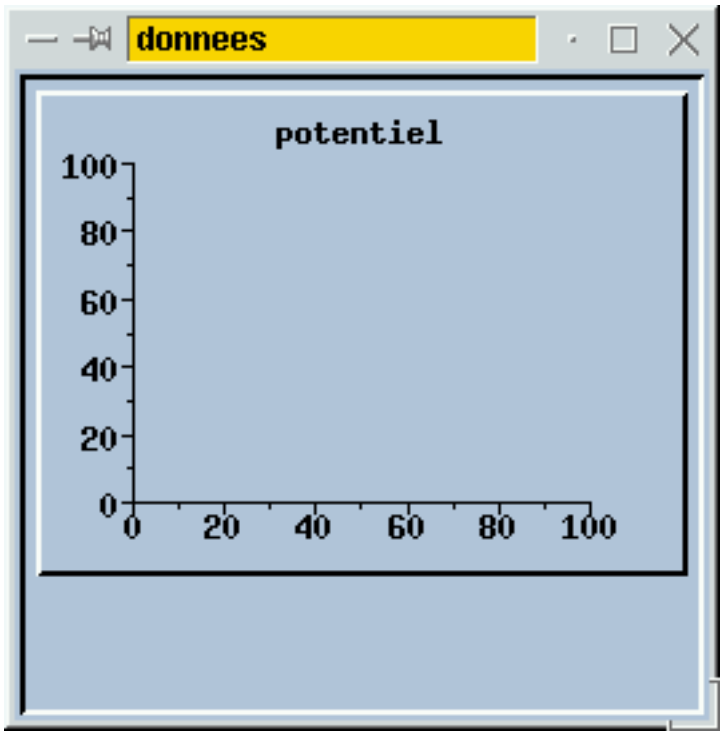


figure 3

On peut détruire la *xform* créée et son contenu par *xhide /donnees.*

VII. Envoyer des messages

A présent, disposant d'un soma et d'un graphe, on va faire passer des informations de l'un vers l'autre. Les communications inter-éléments dans Genesis sont réalisés à l'aide d'un système de liens appelés *messages.* La messagerie permet à un élément d'avoir accès aux champs (*fields*) d'un autre élément.

Par exemple, si on désire représenter les variations de potentiel du soma considéré ci dessus, on fait passer un message du soma au graphe en indiquant que le *champ* devant être représenté est *Vm*

.genesis #11 > addmsg /celluleL11/somaL11 /donnees/potentiel PLOT Vm *volts *green
OK
genesis #12 > reset
time = 0.000000 ; step = 0
genesis #13 > step 100
time = 100.000000 ; step = 100
completed 100 steps in 0.000000 cpu seconds

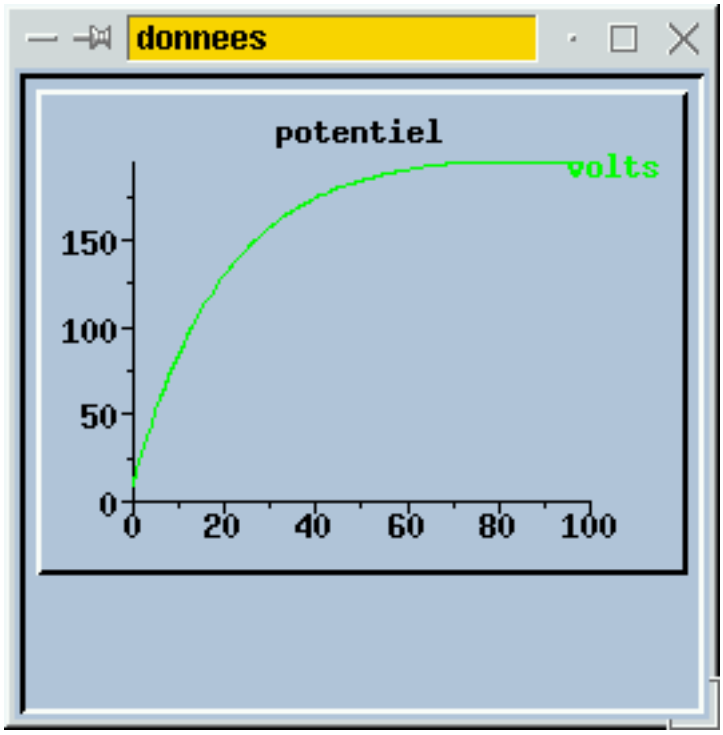


figure 4

Les 2 premiers arguments précisent la source et la destination du message. Le troisième argument indique quel type de message l'on est en train d'envoyer. Ici, il s'agit de *dessiner* (*PLOT*) le contenu du 4ème argument *Vm*. Les 2 derniers arguments sont un intitulé du dessin et sa couleur d'affichage.

Il est possible de représenter sur la même fenêtre les variations d'une autre variable, par exemple le courant injecté *inject ;*

.genesis #11>addmsg /celluleL11/somaL11 /donnees/potentiel PLOT inject *courant *red
OK

De la même façon que la commande *showfield* permet d'examiner le contenu des diverses variables, la commande *showmsg* permet de déterminer quels ont été les messages envoyés entre les éléments. Ceci est utile dans les mises au point des programmes.

genesis #3 > showmsg /donnees/potentiel

INCOMING MESSAGES
MSG 0 from '/celluleL11/somaL11' type [0] 'PLOT' < data = 199.991 > < name = volts > < color = green >
MSG 1 from '/celluleL11/somaL11' type [0] 'PLOT' < data = 20 > < name = courant > < color = red >

OUTGOING MESSAGES

Les messages "entrants" (incoming messages) 0 et 1 correspondent aux 2 messages qui ont été envoyés pour effectuer le graphe, avec la valeur de la variable affichée, à l'issue des 100 pas. Il n'y a pas de message "sortants" (outgoing messages) du graphe .
On peut détruire des messages avec la commande *deletemsg*

genesis #4 > deletemsg /donnees/potentiel 1 -incoming
OK

L'affichage du courant est annulé (dès l'enregistrement de la commande).

genesis #5 > reset
time = 0.000000 ; step = 0

L'affichage du courant ne se fait plus.

On peut également contrôler la nature des messages qui partent et arrivent au soma.

genesis #8 > showmsg /celluleL11/somaL11

INCOMING MESSAGES

OUTGOING MESSAGES
MSG 0 to '/donnees/potentiel' type [0] 'PLOT' < data = 199.991 > < name = volts > < color = green >

VIII. Ajouter des boutons à une xform

L'élément graphique *xbutton* est souvent utilisé pour activer une fonction quand un "click" est effectué sur la "souris"

genesis #9 > create xbutton /donnees/INITIAL -script reset
OK

Un bouton avec l'intitulé INITIAL est instantanément créé dans la fenêtre */donnees* (xform). Quand un click est effectué sur ce bouton, le graphe qui s'y trouvait éventuellement disparaît et l'initialisation est faite.
L'ajout d'un autre bouton se fait suivant les mêmes principes

genesis #10 > create xbutton /donnees/EXECUTION -script "step 100"
OK

Un nouveau bouton intitulé EXECUTION (ou RUN) apparaît dans la *xfom* instantanément. Ici, la fonction qui est appelée dans le "script" nécessite l'utilisation d'un paramètre (le nombre de pas à effectuer). Pour cela, il est nécessaire de disposer de guillemets.

Toutes les commandes qui ont été éxécutées dans l'interpréteur SLI peuvent être rassemblées dans un fichier qui portera un nom ayant l'extension *.g*, par exemple *chapH.g*. Il est habituel de mettre en tête du programme le commentaire (précédé de *//*) *//genesis*. Dans la version finale ci dessous, un bouton *QUITTER* a été ajouté, permettant de quitter la simulation en cours *sans* quitter *genesis*. Pour cela, il a fallu utiliser une commande intitulée *hidegraphics* qui se trouve dans une "bibliothèque" intitulée "bib1.g" qui est une collection de fonctions utiles que l'on s'attache par la commande *include bib1.g*.
//genesis

```
include bib1.g

create neutral /celluleL11
 create compartment /celluleL11/somaL11
 create xform /donnees
 create xgraph /donnees/potentiel
xshow /donnees

 setfield /celluleL11/somaL11 Rm 10 inject 20


 create xbutton /donnees/INITIAL -script reset
 create xbutton /donnees/EXECUTION -script "step 100"
create xbutton /donnees/QUITTER -script "hidegraphics /"

 addmsg /celluleL11/somaL11 /donnees/potentiel PLOT Vm *volts *green
 addmsg /celluleL11/somaL11 /donnees/potentiel PLOT inject *courant *red


check
reset
step 100
```

IX. Mode de fonctionnement de Genesis

Comment Genesis effectue les diverses opérations au cours d'une simulation ? Pour répondre à cette question, on notera tout d'abord que, contrairement aux divers programmes écrits en C, Pascal ou Fortran, il n'y a pas de boucle explicite en temps. Bien qu'il existe un terme *for* comparable à celui qui existe en C, celui ci n'est pas utilisé pour effectuer des itérations dans le temps.

Quand un script est lancé (par exemple chapH.g) ou bien quand des commandes sont entrées interactivement au clavier, le langage SLI éxécute la simulation en traitant l'une après l'autre les commandes présentes dans le script : création d'éléments, affectation de valeurs par *setfield,* établissement de messages entre éléments, etc ...

L'itération dans le temps est implicitement réalisée quand la commande *step* est appelée. La plupart des objets peuvent effectuer des *actions* , quand la commande

step est réalisée. Parmi ces actions, l'une dénommée PROCESS est éxécutée 100 fois, si la commande est *step 100*. Pour connaître toutes les actions pouvant être effectuées par un objet donné, il suffit de taper *showobject nomobjet*.

```
genesis #20 > showobject compartment

object              = compartment
....
VALID ACTIONS
      RESTORE2  SAVE2  SET  CHECK  RESET  PROCESS  INIT
...
```

On dit que le langage est *orienté-objet*. Chaque élément peut éxécuter ses propres actions et affecte les autres éléments seulement par l'échange de messages.

X. Insérer des commentaires dans un script

Il est recommandé, pour faciliter la lecture et la mise au point des programmes, d'y insérer des commentaires. Ceci se fait par l'introduction du signe // au début de chaque ligne devant servir de commentaire.

```
//genesis

// chargement d'une bibliothèque
include bib1.g

//création d'un élément parent
create neutral /celluleL11


//création d'un objet compartiment
 create compartment /celluleL11/somaL11

//affectation de valeurs a des champs internes
 setfield /celluleL11/somaL11 Rm 10 inject 20
```

```
//création et affichage d'une fenêtre pour
// visualiser un graphe
 create xform /donnees
 create xgraph /donnees/potentiel
xshow /donnees

//création de boutons pour éxécuter les commandes
 create xbutton /donnees/INITIAL -script reset
 create xbutton /donnees/EXECUTION -script "step 100"
create xbutton /donnees/QUITTER -script "hidegraphics /"

//envoi de messages du soma vers les graphes
 addmsg /celluleL11/somaL11 /donnees/potentiel PLOT Vm *volts *green
 addmsg /celluleL11/somaL11 /donnees/potentiel PLOT inject *courant *red


check     // test de consistance pour chaque élément
reset     // initialisation
```

Plusieurs lignes de commentaires peuvent être insérées entre les symboles /* et */

```
/* Un script Genesis pour simuler
un simple compartiment
avec injection de courant */
```

XI. Définir des fonctions

Le langage de Genesis permet de définir des fonctions, rendant le programme modulable et plus facilement modifiable. Ces fonctions doivent être groupées au début du script, précédant tout ordre les utilisant.

La structure générale d'une fonction est

```
function <nomdelefonction>(argument1,argument2,..)
  type1 argument1
  type2 argument2
  ..
```

```
 typeN variablelocale1
 ...
<commande1>
<commande2>
 ...
end
```

Par exemple, le calcul et l'affichage de la surface (d'un modèle cylindrique) d'un soma donne lieu à la fonction *calculsurface*

```
//genesis
function calculsurface(longueur, diametre)
        float longueur, diametre
        float surface
                float PI=3.14159
                surface=PI*diametre*longueur
                // affichage
                echo La surface est {surface}
end
```

Si on appelle *fonc.g* le script ci dessus (ne pas omettre */ /genesis !*), l'éxécution de la fonction peut se faire en 2 étapes

```
genesis #1 > fonc.g
genesis #2 > calculsurface 3 3
La surface est 28.27431
```

Les paramètres *longueur* et *diametre* ne sont pas placés entre parenthèses dans l'appel. La fonction prédéfinie *echo* permet l'affichage de ses arguments à l'écran. Le programme précédent construisant un soma passif est réécrit ci dessous à l'aide de fonctions.

```
//genesis
/* Un script Genesis pour simuler
un simple compartiment
avec injection de courant */
```

```
// A l'aide de fonctions


// chargement d'une bibliothèque
include bib1.g


function creercompartiment(chemin)  //chemin=/celluleL11
str chemin
//création d'un élément parent
create neutral {chemin}
//création d'un objet compartiment
 create compartment {chemin}/soma
//affectation de valeurs a des champs internes
 setfield {chemin}/soma Rm 10 inject 20
end


function affichage (chemin)
str chemin
//création et affichage d'une fenêtre pour
// visualiser un graphe
 create xform /donnees
 create xgraph /donnees/potentiel
xshow /donnees
//création de boutons pour éxécuter les commandes
 create xbutton /donnees/INITIAL -script reset
 create xbutton /donnees/EXECUTION -script "step 100"
create xbutton /donnees/QUITTER -script "hidegraphics /"

//envoi de messages du soma vers les graphes
 addmsg {chemin}/soma /donnees/potentiel PLOT Vm *volts *green
 addmsg {chemin}/soma /donnees/potentiel PLOT inject *courant *red

end

creercompartiment /celluleL11
```

```
affichage /celluleL11
check     // test de consistance pour chaque élément
reset     // initialisation
```

XII. Ajouter des canaux ioniques potentiels dépendants

On va construire un modèle de neurone à spikes à l'aide des méthodes décrites ci dessus basées sur l'utilisation de fonctions qui seront relatives à :

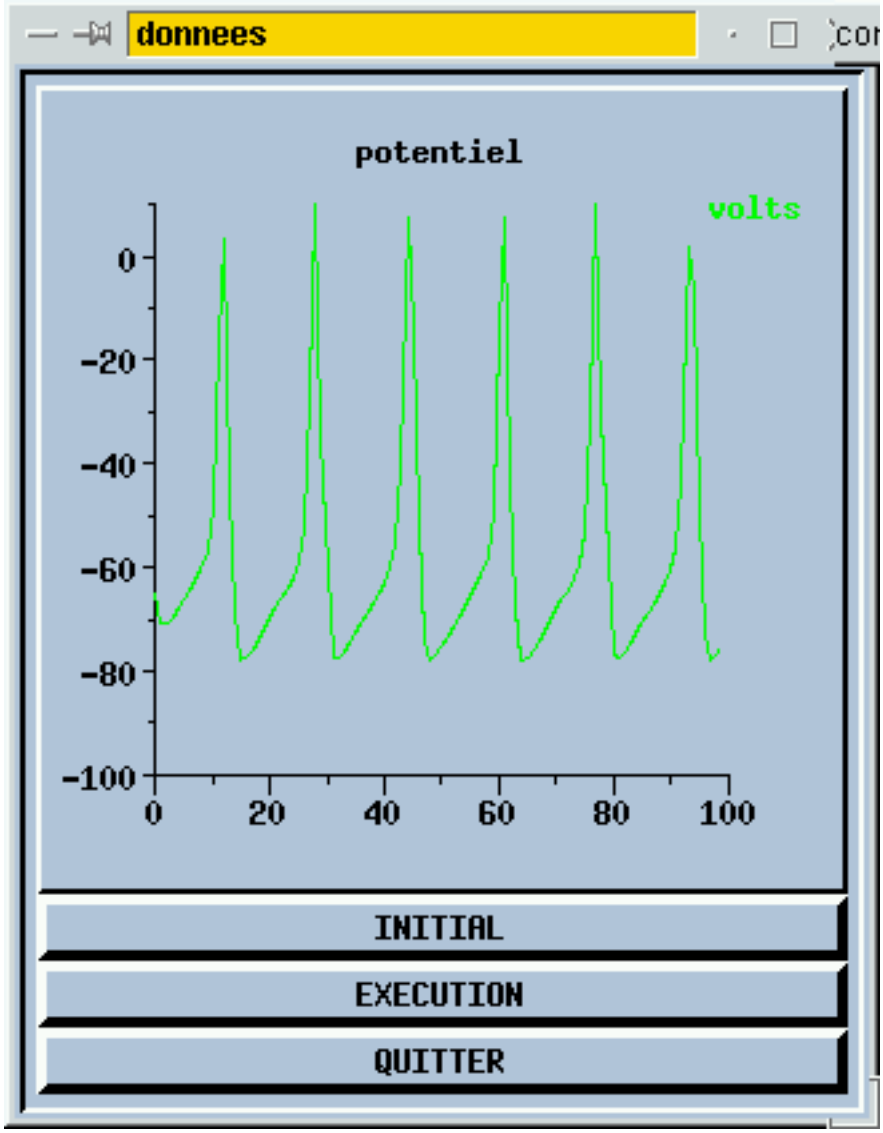
- la confection d'un compartiment soma à l'aide d'une fonction *makecompartment*
- le montage de canaux Sodium et Potassium avec *make_hhNa* et *make_hhK*
- l'affichage des résultats dans une fenêtre avec la fonction *affichage*

Ces fonctions ont été définies dans une bibliothèque appelée *bib1.g* . Le programme est le suivant :

```
//genesis
/* Un script Genesis pour simuler
un compartiment soma avec canaux ioniques
de type Hodgkin Huxley
avec injection de courant */

// A l'aide de fonctions definies dans bib1.g
// chargement de la bibliothèque
include bib1.g

// On utilise les unites "physiologiques"
//              SI            Physiologiques
// resistance  ohm                kilohm
//capacitance  farad              microfarad
//potentiel    volt               millivolt
//courant      ampere             microampere
//temps        seconde            milliseconde
//conductance  siemen               millisiemen
//longueur     metre              centimetre

// compartment dimensions (cm.)
```

```
// soma_l = 30e-4 // dans bib1.g
// soma_d = 30e-4 // dans bib1.g
// Eleak (potentiel déquilibre dans bib1.g)
// ENa, EREST_ACT dans bib1.g

float active_area = soma_l*PI*soma_d*1.0

create neutral /R15

makecompartment /R15/soma {soma_l} {soma_d} {Eleak}
make_hhNa /R15/soma Ex_channel {active_area} {ENa} {EREST_ACT}
make_hhK /R15/soma Inh_channel {active_area} {EK} {EREST_ACT}

setfield /R15/soma inject 0.0003  //unites physio =>0.0003microA
affichage /R15/soma    / fonction affichage dans bib1.g
check     // test de consistance pour chaque élément
reset     // initialisation
```

Lors de l'éxécution, ce soma à l'intérieur duquel une injection de courant de $0.3nA$ a été opérée, donne lieu à la génération d'un train de potentiels d'action comme cela est indiqué sur la figure cidessous :


donnees
potentiel
volts
0
-20
-40
-60
-80
-100
0
20
40
60
80
100
INITIAL
EXECUTION
QUITTER