A Simplifed Introduction to NEURON Simulator
—（私的）NEURON 事始め—

Keiji Imoto
Department of Information Physiology
National Institute for Physiological Sciences
Okazaki 444-8787, Japan

March 13, 2009

目次

1 はじめに

Ted Carnevale と Michael Hines により開発され、現在も開発され続けている NEURON シミュレータは、もっともポピュラーな神経細胞、神経ネットワークのシミュレータである。10 年以上の歴史があり、多くの論文でも利用されている。基本的に NEURON は、神経細胞をコンパートメントに分け、コンパートメント i の電位変化を、微分方程式

$$\frac{dV_i}{dt} = -\frac{1}{C_i}\sum_j I_{i,j}$$

で表し、これらを数値的に解くものである。神経細胞の形態を再現しようとするとコンパートメントの数が多くなり、またそれぞれのコンパートメントに含まれるチャネルのキネティックスの定義にもよく微分方程式が用いられるため、多数の微分方程式を並行して計算することが必要となる。NEURON はそのような機能を備えたソフトウェアである。NEURON は現在もすこしずつ機能が付け足されてきており、多数の（モデル）神経細胞からなるネットワークのシミュレーションも可能である。

このように NEURON は高度な性能を持つソフトウェアであるが、2006 年に The NEURON Book (Canbridge University Press) が出版されるまで、まとまった入門書がなかったこともあり、なかなか取っ付きにくいというのが一般的な印象だろうと思われる。筆者自身、何度か NEURON をコンピュータにインストールし、チュートリアルなどを試してみたが、結局使えるほどの理解を得るには至らなかった。問題点を考え直してみたところ、少なくとも個人的には、プログラムの中核である oc インタープリタをどう使うのかがわからない、ということが初歩のチュートリアル以上に進めない理由なのではないか、と思うようになった。

この（私的）事始めでは、GUI 部分を出来るだけ利用せず、NEURON の根幹である oc インタープリタを動かすための hoc ファイルの理解を中心に説明を試みた。

NEURON シミュレータの特徴として 3 つの点があげられる。

1. oc は NEURON の根幹となるインタープリタであり、非常に数多くの連立微分方程式の数値解を求めることができる。
2. mknrnmodl（システムによっては mknrndll）は、イオンチャネル等のコンポーネントを新たに作製し、プログラムに組み込むことができる。
3. template 機能を用いて、object 指向的なプログラムが可能である。

2 は微分方程式等の詳細を記載した mod ファイルをシステムに組込む操作である。初歩的な機能要素は既定のシステムに備わっているので、備わっていない機能要素を加える場合に必要。

また付属の機能として、次のようなものがある。

- CellBuilder: 神経細胞の複雑な形態を、円柱で近似されるコンパートメント（section と呼ばれている）のつながりとしてプログラム化する。
- NetBuilder: 神経細胞間の情報伝達を event の伝達として処理するようにプログラム化する。

これらは GUI を主体とした部分であるが、複雑な構造やネットワークを作っていくには、むしろ古典的なテキストファイルを用いる方法の方が適している場合もある。

NEURON は（一旦学べば）比較的手軽に使用できるシミュレーション環境である。特に有用な課題は、

1. 複数のコンパートメント（section）からなる神経細胞での、イオンチャネル機能の検討。特にイオンチャネル間の相互作用やクランプエラー等の測定誤差の理解に有用。
2. 細胞集団におけるスパイクの同期性等におよぼす要素の検討。但し、この種類のシミュレーションをパソコンレベルのコンピュータで行なう場合には、単純化したモデルニューロンを使用する必要がある。

なお、文中の  はコメント、  は注意すべき事項を示す（*dangerous bend* 危険な曲がり道）。

2 hoc ファイルと oc インタープリタ

2.1 インタープリタとしての oc

oc は、Windows/Cygwin の場合、nrngui のアイコンをクリックするか、コマンドプロンプトで neuron とコマンド入力することで起動できる。Mac OS の場合、nrngui のアイコンをクリックするか、コマンドから開く場合は、open -a nrngui で起動できる（ソースコードからコンパイルした場合は、異なる)。いずれの環境の場合も、nrngui の場合は、nrngui.hoc がロードされ実行されている。Mac OS でコンパイルした場合や Linux の場合は、PATH をしかるべく設定すれば、nrniv で起動できるはずである。

oc は、一行一行入力することも可能であるが、通常 hoc ファイルを読み込ませて作業を行う。その場合には、neuron *hoc_file_name* -とする。最後の'-' を付けると、hoc ファイルの実行を終えても oc は終了しない。

hoc プログラム上の要点。

- 数値変数は宣言を必要としない。
- 数値変数は全て **double**（8 バイト実数)。整数として用いられる場合には truncate される。
- C 言語のように **double** i というように宣言すると、エラーとなる
- 配列の場合は数値でも **double** の宣言必要。配列要素の数は定数でなくてもよい。
- 数値変数を含め、変数は通常グローバル変数 global として扱われる。
- 数値変数を関数内に限局するには **local** の宣言が必要。
- 数値以外の変数には、色々なオブジェクトのレファレンス、文字列等があり、**objref**、**strdef** 等の宣言を必要とする。[*1]
- オブジェクトのレファレンスを関数内で局所的に使用するには、**objref** の代わりに **localobj** を使用する。(**strdef** の local 版はないようだ。)
- ポインターを使用することができる。
- シミュレーションの要素としては、コンパートメントである section、そこで局所的に起きる現象 point process がある。section は global として扱われるようである。point process は単独で使用することは出来ず、ある section の point process として使用される。概念的な point process を使用する場合は、形式的に section を定義して用いる。
- 関数は、func()、proc() で定義される。func() は数値を返すのに対して、proc() は返り値なし。引数は、数値の場合$1、文字列の場合$s1、**objref** の場合$o1 として表される。func()、proc() の括弧の中には引数を書かない。
- main() は存在しない。各種の宣言（変数、関数、クラス）以外の部分が順番に実行される。
- 組込み関数がある（表 1)。例えば、**printf()** は、標準出力に文字列を出力する。
- プログラムの流れを制御するステートメントとして、**if**、**else**、**break**、**for**、**while** 等が使用できる。
- **forall**、**forsec** 等が便利。

```
//-----------------------------------------------
// n01.hoc:   a C-like hoc program
```

[*1] クラスとは、鋳型もしくはひな形のことを指す。ひな形を元に実際に作られたもの（ここでは変数）がオブジェクト（インスタンスとも言う）である。

```
    strdef s
    s = "Hello, World!"
    for t = 0, 3 {
        printf("%d %s\n", t,  s)
    }
//-----------------------------------------------
```

 //で始まる行はコメント行。//が行の途中で来れば、それ以降がコメント。また C 言語と同様に、/* */を使用できる。文字列の変数を使用する場合には、変数の宣言として **strdef** が必要。**for** にはいく通りかの使用方法があるが、この使い方が最もよく用いられている様である。この場合、後の数（ここでは 3）が含まれていることに注意。**printf** は、C 言語の場合とほぼ同じ。変数が double の場合でも、%f でよく %lf とする必要はない様である。

ファイルへの出力の場合は、次のように行なう。

```
//-----------------------------------------------
// n02.hoc:       a C-like hoc program   with text file output
    strdef s
    s = "Hello, World!"
    wopen("a.txt")    // open for write
    for t = 0, 3 {
        fprint("%d %s\n", t, s)
    }
    wopen()
//-----------------------------------------------
```

 ファイル IO（Input-Output）には、テキストファイルが用いられることが多い。書き込み用にファイルを開く場合は、**wopen(***filename***)** を用い、閉じる場合は引数なしで **wopen()** を使用する。C 言語で用いられる **fprintf()** ではなく **fprint()** であることに注意。ファイル IO には、File クラスを用いる方法もあり、この場合はバイナリの読み書きがサポートされている。

実際に数値をファイルに書き出し、それをグラフを描くプログラム gc を用いてプロットしてみる。gc はホームメイドのプログラムで、コマンドで使用可。Windows 版と MacOS 版を用意してある。MacOS 版はバイナリファイルのみ使用可。

```
//-----------------------------------------------
// n03.hoc:   drowing a small graph
    wopen("a.txt")   // open a file for "write"
    tstop = 200
    for t=0, tstop {
        fprint("%f %f\n", t, sin(0.1*t))
    }
    wopen()  // close the output file
```

```
    WinExec("gc a.txt")
//------------------------------------------------
```

プログラム中からコマンドを使ったり、他のプログラムを起動するには、**system()** を利用する。Windows/Cygwin 環境では、**WinExcec()** を使用する。

binary ファイルの場合は下記のようになる。binary を用いる利点は、ファイルサイズがコンパクトで、読み書きが速いことであるが、テキストエディタで内容がわからない欠点がある。

```
//------------------------------------------------
// n04.hoc:   drowing a small graph (binary file version)
    objref fp
    fp = new File()
    fp.wopen("a.dat")
    double a[2]

    tstop = 200
    for t=0, tstop {
        a[0] = t
        a[1] = sin(0.1 * t)
        fp.vwrite(2, &a)
    }
    fp.close()
    WinExec("gc a.dat")
//------------------------------------------------
```

File クラスを利用している。クラスを利用してオブジェクトを作成するには、まず **objref** *aObject* と宣言をして、*aObject* = **new *Class*()** という構文でオブジェクトを作成する。入力にファイルを開く場合は **.ropen()**、出力にファイルを開く場合は**.wopen()** を用いる。binary ファイルの読み書きには、**.vread()**、**.vwrite()** を用いる。また、C 言語でおなじみの**.seek()**、**.tell()** も利用できる。

2.2 Section

いよいよ神経細胞のコンパートメント（section）を定義する。これには **create** というステートメントが用いられる。内部的にはコンパートメントのオブジェクトが作成されるのだと推測される。先ず初めに 1 コンパートメントを作成し、そこに Hodgkin-Huxley タイプの Na^+ チャネル、K^+ チャネル、leak チャネルのパッケージである hh を挿入してみる。

```
//------------------------------------------------
//  n05.hoc:  a simple hh cell
    create soma
    soma insert hh
```

```
    tstop = 200
    dt = 0.01
    v_init = -65
    wopen("a.txt")
    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        fprint("%f %f\n", t, soma.v(0.5))
    }
    wopen()
    WinExec("gc a.txt &")
//------------------------------------------------
```

 hoc ファイルでの操作は、いずれかの section に対して行なう場合が多い。一般的に、*section statement* という形式が用いられる。**finitialize()**、**fcurrent()** は、シミュレーション計算の初期化を行なう。**fadvance()** は、1 ステップ計算を行なう。global 変数である時間 **t** は、**fadvance()** により **dt**（これも global 変数）だけ増加されるので、**t** を増加させる命令は不必要である。

出力ファイルを binary file にしたバージョン。

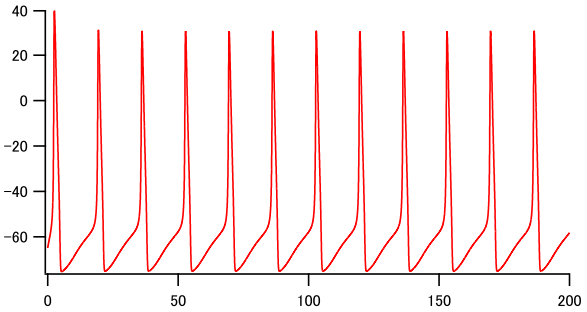
```
//------------------------------------------------
// n06.hoc:  a simple hh cell (binary version)
    create soma
    soma insert hh

    tstop = 200
    dt = 0.01
    v_init = -65
    objref fp
    fp = new File()
    fp.wopen("a.dat")
    double a[2]

    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
```

```
        fp.vwrite(2,&a)
    }
    fp.close()

    WinExec("gc a.dat")
//------------------------------------------------
```

n05.hoc と n06.hoc では、soma の電位 v をプロットしているが、予想通り何もおきない。

これではつまらないし、うまくプログラムが動作しているかもわからないため、leak 電流の性質を変更してみる。n05.hoc と n06.hoc では、soma や hh のパラメータは、既定値を用いており、hh の leak 電流の平衡電位は、default の場合-54.3 mV であるが、この値を-30 mV に変更してみる。細胞はこの leak 電流のために脱分極し、Na^+ チャネルの閾値に達して action potential を発生し、K^+ チャネルにより再分極する。まさしく HH モデルの世界！なお hh の詳細は hh.mod で定義されている。

```
//------------------------------------------------
// n07.hoc:  a simple hh cell, with increased leak
    create soma
    soma insert hh
    soma el_hh = -30    // added
    tstop = 200
    dt = 0.01
    v_init = -65
    objref fp
    fp = new File()
    fp.wopen("a.dat")
    double a[2]

    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        fp.vwrite(2,&a)
    }
    fp.close()
    WinExec("gc a.dat")
//------------------------------------------------
```



ここではシステムが動くことを確認するために hh.mod を用いているが、hh.mod は squid axon のモデルであり、6.3 ℃という低温での実験に合わせたパラメータを用いている。温度は global 変数 **celsius** で設定できる。**celsius** は default で 6.3 に設定されているようである。温度は $Q10$ を通してチャネル等のキネティックスに反映される。n07.hoc の場合、**celsius** = 37 とすると、action potential は発生しなくなってしまう。なお、*soma* **gnabar_hh** *= 3 として、Na チャネルの density を 3 倍に増加させると、action potential は再び現れる。神経細胞用には、hh2.mod などが作られている。実際の計算にはそれらを用いた方がよいと思われるが、mod ファイルをシステムに組込む作業が必要なので、当面は既に組込まれている hh.mod を用いることとする。
(http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=3817)

2.3 Connecting sections

次に、soma に apical dendrite を付け加える。**create** は同じ。**connect** でつなぐ。先にも述べたが、一般的に hoc の命令文（statement）は、いずれかの section に対して行なわれ、形式的には、*section statement* という形となる。複数の statement を {} で囲むことができる。ただし *section* と { の間に改行を入れないようにする。section の情報は必ずしも明示的に示される訳ではない。**access** *section* により section の既定値を設定できる。

```
//------------------------------------------------
// n08.hoc:   soma + dendrite

    create soma, ap_dend
    soma {
        L = 30
        diam = 30
        nseg = 1
        insert hh
        el_hh = -30
    }
    ap_dend {
        L = 500
        diam = 2
```

```
        nseg = 23
    }
    connect ap_dend(0), soma(1)

    tstop = 200
    dt = 0.01
    v_init = -65

    objref fp
    double a[4]
    fp = new File()
    fp.wopen("a.dat")
    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        a[2] = ap_dend.v(0.1)
        a[3] = ap_dend.v(0.9)
        fp.vwrite(4, &a)
    }
    fp.close()
    WinExec("gc a.dat")
//-----------------------------------------------
```

 soma と ap_dend の 2 つの section より成り立っている。**nseg** で section 内の segment の数を示す。この値は奇数でなくてはならない。soma および ap_dend の近位部から測って 0.1 と 0.9 の部分の電位変化をプロットしている。活動電位が soma から ap_dend に伝わっていることがわかる。

2.4 Point Process

2.4.1 AlphaSynapse

section とならんで重要な要素は、Point Process と呼ばれるものであり、シナプス入力（AlphaSynapse、Exp2Syn)、current clamp (IClamp)、voltage clamp (VClapm) 等がこれにあたる。これらの Point Process はクラスとして定義されており、section に加える場合は、**insert** ではなく、次の方法で行なう。

objref *pp*

section pp = **new** *PointProcess*(x)

x は、section での位置を示す。

dendrite にシナプス入力を入れる。soma の hh の leak 電流の平衡電位は、default の値に戻した。また

dendrite に soma の 1/10 の density で hh を加えた。

```
//------------------------------------------------
// n09.hoc
    create soma, ap_dend
    soma {
        L = 30
        diam = 30
        nseg = 1
        insert hh
    }
    ap_dend {
        L = 500
        diam = 2
        nseg = 23
        insert hh
        gnabar_hh = 0.012
        gkbar_hh  = 0.0036
        gl_hh = 0.00003
    }
    connect ap_dend(0), soma(1)

// synaptic input
    objref syn
    ap_dend syn = new AlphaSynapse(0.5)
    syn.onset = 5
    syn.tau = 0.1
    syn.gmax = 0.05

    tstop = 20
    dt = 0.01
    v_init = -65

    objref fp
    double a[4]
    fp = new File()
    fp.wopen("a.dat")

    finitialize(v_init)
    fcurrent()
```

```
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        a[2] = ap_dend.v(0.1)
        a[3] = ap_dend.v(0.9)
        fp.vwrite(4, &a)
    }
    fp.close()

    WinExec("gc a.dat")
//------------------------------------------------
```

2.5 クラス

同じような神経細胞をいちいち定義するのは手間がかかる。クラス（テンプレート；鋳型）を定義し、クラスからオブジェクト（インスタンス）を作成すると、手間が省ける。

クラスの定義は、**begintemplate** *class_name* と **endtemplate** *class_name* で囲んだ部分。変数を明示的に初期化する関数 **init()** を用意することが必要である。**public** として宣言された変数、関数が外からアクセスすることが出来る。

2.6 NetCon、NetStim

神経細胞間の情報伝達は、神経軸索をつたわる活動電位を計算することによりシミュレーションすることが可能だが、いちいち計算しなくても伝達にかかる時間を delay として取り扱えば、計算量を大幅に減らすことができる。NetCon は delay を考慮に入れたシグナル伝達を扱う pipe メカニズムである。

objref *nc*

nc = **new NetCon(***src_pp*, *target_pp*, *threshould*, *delay*, *weight***)**

もしくは、

src_section nc = **new NetCon(**&*src_range_value*, *target_pp*, *threshould*, *delay*, *weight***)**

という形で定義される。ここで*_pp* は point process を示す。*src_rannge_value* は、&v(0.5) の様に表す。&soma.v(0.5) という表記はエラーとなる。

NetCon の情報は **List()** を用いて扱うと便利である。

objref *nclist*

nclist = **new List()**

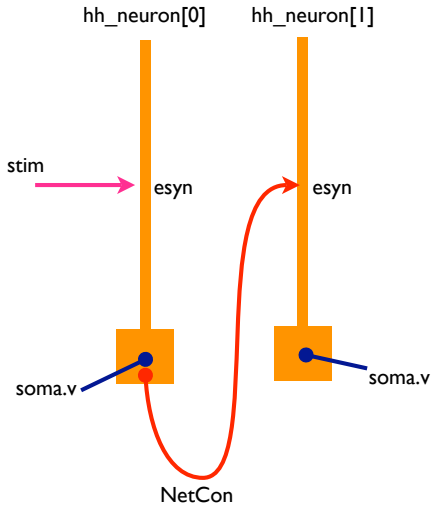
nclist.**append(new NetCon(***src_pp*, *target_pp*, *threshould*, *delay*, *weight***))**

もしくは、

src_section nclist.**append(new NetCon(**&*src_range_value*, *target_pp*, *threshould*, *delay*, *weight***))**

外部刺激に相当する Point Process メカニズムとして、**NetStim** クラスが用意されている。**NetStim** は、**NetCon** の source としても target としても利用できる。パラメータとしては、**interval**、**number**、**start**、

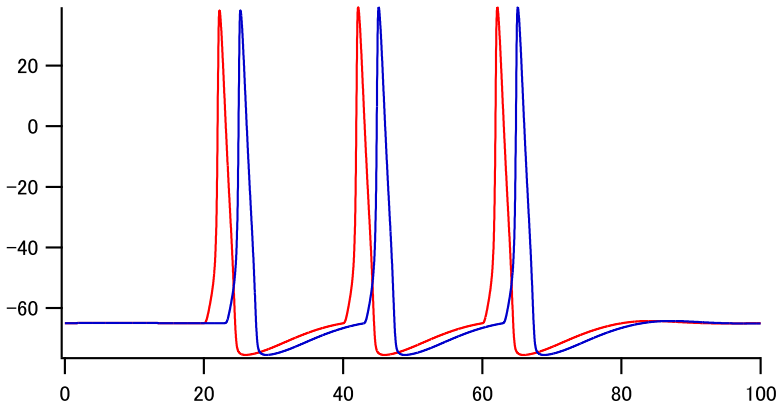
noise などがある。



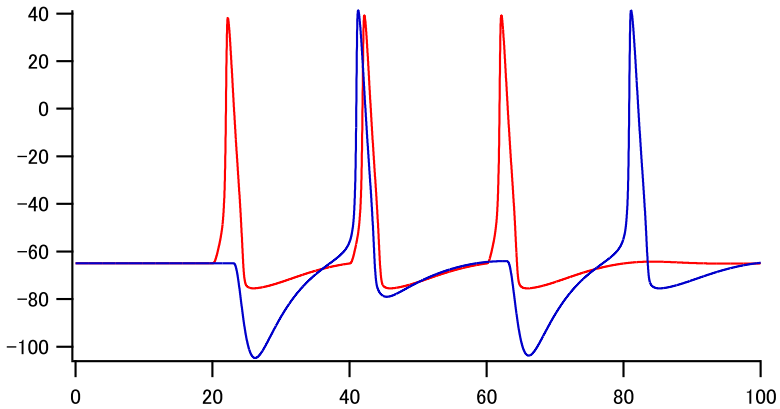

```
//---------------------------------------
// n10.hoc
//---------------------------------------
//     class definition
    begintemplate HHneuron
        public soma, ap_dend, esyn
        create soma, ap_dend
        objref esyn
        proc init(){
            soma {
                L = 30
                soma.diam = 30
                soma.nseg = 1
                soma insert hh
            }
            ap_dend {
                L = 500
                diam = 2
                nseg = 23
                insert hh
                gnabar_hh = 0.012
                gkbar_hh  = 0.0036
                gl_hh = 0.00003
            }
```

```
            connect ap_dend(0), soma(1)
            ap_dend esyn = new Exp2Syn(0.5)
            esyn.tau1 = 0.5
            esyn.tau2 = 1.0
            esyn.e = 0
        }
    endtemplate HHneuron
//---------------------------------------

// cells
    objref hh_neuron[2]
    hh_neuron[0] = new HHneuron()
    hh_neuron[1] = new HHneuron()

// synapse
    objref stim
    stim = new NetStim(0.5)
    stim.interval = 20
    stim.number = 3
    stim.start = 20
    stim.noise = 0

// connections
    objref nclist
    nclist = new List()
    nclist.append( new NetCon(stim, hh_neuron[0].esyn, 0.0, 0, 0.02))
    hh_neuron[0].soma nclist.append( new NetCon( &v(0.5), \
                                    hh_neuron[1].esyn, 10, 1, 0.02))
//---------------------------------------
    tstop = 100
    dt = 0.01
    v_init = -65

    objref fp
    double a[3]
    fp = new File()
    fp.wopen("a.dat")
    finitialize(v_init)
    fcurrent()
    while(t<tstop){
```

```
        fadvance()
        a[0] = t
        a[1] = hh_neuron[0].soma.v(0.5)
        a[2] = hh_neuron[1].soma.v(0.5)
        fp.vwrite(3, &a)
    }
    fp.close()
    WinExec("gc a.dat")
//--------------------------------------
```



NetCon で weight のパラメータを 0.02 から-0.02 に変えてみると、次の結果が得られた。Rebound firing を示している。ただしこの神経細胞は Na^+ チャネル、K^+ チャネル、leak チャネルを持っているだけで、low-threshold タイプの Ca^{2+} チャネルを有しているわけではない。

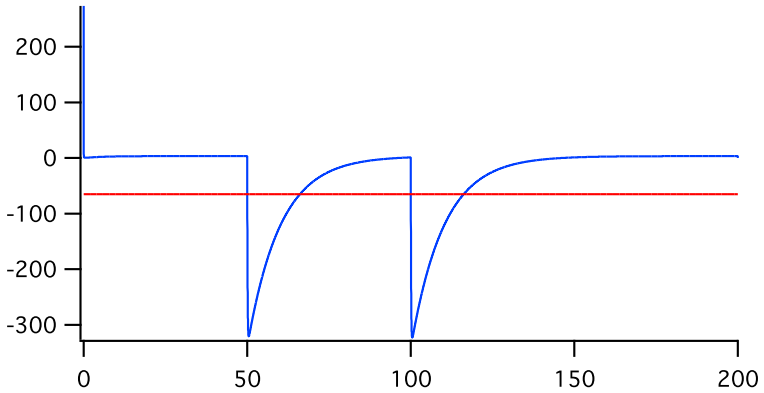


2.7 Voltage clamp

単一コンパートメント（soma のみ）にシナプス入力がある場合に、voltage clamp で測定する場合のシミュレーション。

```
//---------------------------------------
//  n11.hoc
//
    create soma
    objref esyn
    objref vcl

    soma {
        L = 30
        diam = 30
        nseg = 1
        esyn = new Exp2Syn(0.5)
        insert hh
        vcl = new VClamp(0.5)
    }

// stimulation
    objref stim
    stim = new NetStim(0.5)
    stim.interval = 50
    stim.number = 2
    stim.start = 50
    stim.noise = 0

// synaptic connections
    objref nclist
    nclist = new List()
    nclist.append( new NetCon(stim, esyn, 0.0, 0, 0.005))
//---------------------------------------
    tstop = 200
    dt = 0.01
    v_init = -65

    vcl.dur[0] = 10
```

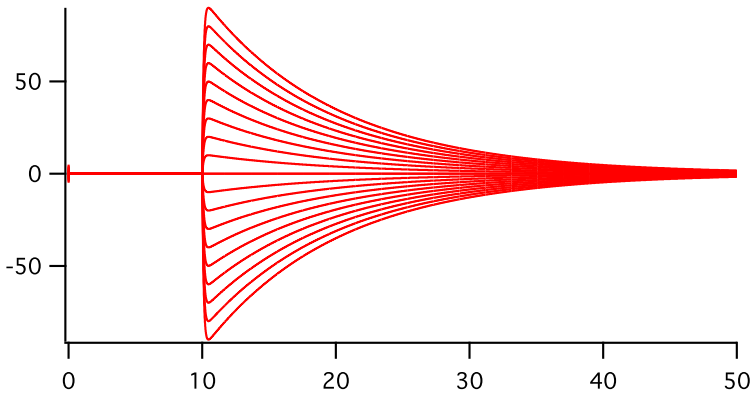
```
    vcl.dur[1] = 10
    vcl.dur[2] = 180
    vcl.amp[0] = v_init
    vcl.amp[1] = v_init
    vcl.amp[2] = v_init
    vcl.gain = 1000
    vcl.tau1 = 0.01
    vcl.tau2 = 0.01

    objref fp
    double a[3]
    fp = new File()
    fp.wopen("a.dat")
    finitialize(v_init)
    fcurrent()

    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        a[2] = 1000 * vcl.i
        fp.vwrite(3, &a)
    }
    fp.close()
    system("gc a.dat")
//--------------------------------------
```



Voltage clamp で、シナプス電流の i-v relation を求める。基本的には上記のプログラムと同じ。holding potential を変化させて繰り返しを行なっている。

```
//--------------------------------------
//  n12.hoc
// iv in voltage-clamp mode
    tstop = 50
    dt = 0.01
    vstart = -100
    vstep = 10     // voltage step of holding potential
    ntrace = 20   // number of traces
    npnt   = tstop/dt
    double a[npnt][ntrace+1]

    objref vc, syn, ns, nc
    create soma
    soma {
        diam = 30
        L = 30
        nseg = 1
        insert pas
        vc = new VClamp(0.5)
        vc.dur[0] = 10
        vc.dur[1] = 10
        vc.dur[2] = 30
        syn = new Exp2Syn(0.5)
        syn.e = 0
    }
    ns = new NetStim()
    ns.number = 1
    ns.start = 10
    ns.noise = 0
    nc = new List()
    nc.append(new NetCon(ns,syn, 0, 0, 0.001))   // 1 nS

    for i=0, ntrace-1 {
        v_init = vstart + vstep * i
        vc.amp[0]=v_init
        vc.amp[1]=v_init
        vc.amp[2]=v_init
        e_pas = v_init
        finitialize(v_init)
        fcurrent()
```

```
        for j = 0, npnt-1 {
            fadvance()
            a[j][0] = t
            a[j][i] = 1000 * vc.i    // nA -> pA
        }
    }

    objref fp
    fp=new File()
    fp.wopen("a.dat")
    for j = 0, npnt-1 {
        fp.vwrite(ntrace+1, &a[j][0])
    }
    fp.close()
    system("gc a.dat")
//---------------------------------------
```



シナプス入力には Ex2Syn() を用いた。leak 電流には pas を用い、holding current を消すために、leak 電流の平衡電位が holding potential と同じであるとして計算している。2 次元の配列にデータを記録し、まとめてファイルに書き込んでいる。

soma より 10 本の dendrites がでており、soma で voltage clamp を行なった場合のシミュレーションは次のようになる。

```
//---------------------------------------
//  n13.hoc
//
    create soma, dend[10]
    objref esyn
    objref vcl
```

```
    soma {
        L = 30
        Ra = 100
        nseg = 1
        diam = 30
        insert hh
        vcl = new VClamp(0.5)
    }
    for i=0,9 {
        dend[i] {
            L = 300
            Ra = 100
            nseg = 21
            diam=2
            insert hh
            gl_hh *= 2.3
        }
    }
    dend[0] esyn = new Exp2Syn(0.75)
    for i=0,9{
        connect dend[i](0), soma(1)
    }

// stimulation
    objref stim
    stim = new NetStim(0.5)
    stim.interval = 50
    stim.number = 2
    stim.start = 50
    stim.noise = 0

// synaptic connections
    objref nclist
    nclist = new List()
    nclist.append( new NetCon(stim, esyn, 0.0, 0, 0.001))

//---------------------------------------
    tstop = 200
    dt = 0.01
```

```
    v_init = -65

    vcl.dur[0] = 10
    vcl.dur[1] = 10
    vcl.dur[2] = 180
    vcl.amp[0] = v_init
    vcl.amp[1] = v_init
    vcl.amp[2] = v_init
    vcl.gain = 1000
    vcl.tau1 = 0.1
    vcl.tau2 = 0.1

    objref fp
    double a[4]
    fp = new File()
    fp.wopen("a.dat")
    finitialize(v_init)
    fcurrent()

    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        a[2] = dend[0].v(0.75)
        a[3] = 1000 * vcl.i
        fp.vwrite(4, &a)
    }
    fp.close()
    WinExec("gc a.dat")

//---------------------------------------
```

3 Morphology

3.1 section

NEURON では、神経細胞の形態を円柱もしくは錐体の集まりとして表現している。それぞれのパーツは、section と呼ばれる。表面として計算されるのは、円柱の側面に当たる部分であり、断面に当たる部分は考えに入れられていない。soma も、球体・楕円体ではなく円柱として考えられる。半径 r の球の表面積は、$4\pi r^2$ で表されるが、長さ $2r$、半径 r の円柱の側面の面積は、$(2r) * (2\pi r)$ なので、球の場合と同じになる。

長い円柱の場合、cable property を考慮に入れなくてはならない。NEURON では section を segment に分割して計算する機能を備えている。nseg は分割の値であり、演算の技術的な理由で、この値は奇数でなくてはならない。円柱の位置を示すには、分割された部分の番号ではなく、0 と 1 の間の値で示される Normalized distance が用いられる。このために、nseg の値を変更しても、場所を示す値を変える必要はない。

nseg をどのような値にするかにより、計算の結果は異なってくる。通常は nseg = 3 程度でよいが、形態学的なデータに基づく枝分かれのあるモデルの場合は、nseg $\geq$ 9 が必要であるとされている。

section のパラメータとしては、次のものがある。

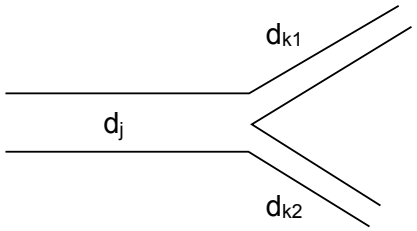
- **L** // Length [μm]
- **Ra** // cytoplasmic registivity [Ωcm]
- **nseg** // discretization parameter

それぞれの segment でのパラメータである Range variable には次のようなものがある。

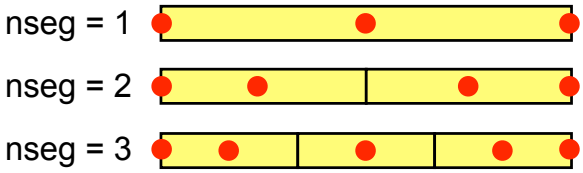
- **diam** // diameter
- **cm** // specific membrane capacitance [μf/cm^2]
- **v** // membrane potential [mV]
- **nai** // internal sodium concentration [mM]

range variable が、distance に対して linear に変化する場合、dend01.diam(0:1) = 1.5:1.0 というように書くことができる。このような値の設定は、各 segment 毎の値を入れているので、当然のことながら、nseg が先に与えられなくてはならない。

Dendrite に分岐がある場合にそれらを単一の dendrite として計算を行なう Rall の Cable 理論では、$(d_j)^{2/3} = (d_{k1})^{2/3} + (d_{k2})^{2/3}$ という条件を満たしている必要があった。



しかし NEURON では、演算をおこなう点（node）を section のつながりの部分にも置いており、section 間の条件は緩和されている。



赤丸が演算を行なう node。

3.2 geometory

神経細胞を模したモデルを作るには、soma、dendrite、axon などの section を create し、それらをつなげばよい。section をつなぐには、connect *child*(0 *or* 1), *parent* (*x*) を用いる。これは、*parent* connect *child*(0 *or* 1), *x* と書いても同じである。

この操作を簡便にするために、Menu → Build → Cell Builder が用意されている。より複雑な形態をした神経細胞のデータ入力には、ファイルからの読み込で行なわれる。3 次元の座標（*x*, *y*, *z*）と半径 *diam* が与えられている場合、pt3dadd() を用いる。

入力したあるいは読み込んだデータの確認には、

- *section_name* **psection()** // parameters of a section
- **forall psection()** // parameters for all sections
- **topology()** // section connections
- Menu → Graph → Shape plot

が便利である。

実際に神経細胞の形態のデータを読み込むには、先ず dendrite を配列として **create** し、データを読み込み、そして section を **connect** する。下記のような例を見ると参考になる。

- http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=279 の tc200.geo、

これらの例では、ファイルからの読み込みの場合、hoc ファイルにプログラムコードとデータの数値が同一のファイルに入っているという、普通のプログラムから考えると奇妙な成り立ちになっている。これは、比較的初期の NEURON では、ファイルの入出力に制限（入力用にファイルが 1 つしか開けない）があったために、苦肉の策としてこのような形になったのではないかと思われる。しかしながら、これらのファイルは単に読み込むだけで形態のデータを取り込めるので非常に便利である。

通常これらのデータは、section の長さ（**L**）、直径（**diam**）および Connection を定義しているが、**nseg** や **Ra** 等を含めて他の定義は行なっていないことに注意する必要がある。

```
load_file("tc200.geo")
```

なお、ソースコードとデータを分け、テキスト形式のデータファイルからデータを読み込むには、File クラスを用いて、次のようなコードを適宜書き換えて使用すればよい。古典的な **ropen()** と **fscan()** の組み合わせは、動かない。（クラス関数の**.ropen()** と **ropen()** は別物らしい。）

```
//---------------------------------------
    objref fp
    fp = new File()
    fp.ropen("datafile.txt")   // open for read
    while(1){
        a = fp.scanvar()           // fscan() does not work
        printf ("a = %f\n", a)
    }
    fp.close()
//---------------------------------------
```

3.3 segment の長さ

例を見ていると、100 μm を越えるような細長い section でも **nseg**=1 となっている。これでは具合が悪いであろうということは想像に難くない。シグナルの減弱は距離と周波数に関係しており、e-fold の減弱が生じる長さは、

$$\lambda_f \approx \frac{1}{2}\sqrt{\frac{d}{\pi f R_a C_m}}$$

で表される。diam = 1 μm、R_a = 180 Ωcm、C_m = 1 μf/cm^2、R_m = 16,000 Ωcm^2 とすると、$\lambda_{100} \sim 225$ μm となる。segment の長さ（**L/nseg**）の λ_f に対する比率パラメータを、d_lambda と呼ぶ。d_lambda の既定値は 0.1 である。（ただし、膜の時定数 τ_m が 8 ms 以下の場合はより小さい値を用いる必要がある。）目安として、**diam** が 1 μm の場合、1 segment の長さは 20 μm 程度にするということになる。

自動的に **nseg** を決めるには、下のコードを用いる。これらの関数は stdlib.hoc に含まれている。

```
//---------------------------------------
    func lambda_f(){
            return 1e5 * sqrt(diam/(4 * PI * $1 * Ra * cm))
    }
    proc geom_nseg(){
        soma area (0.5)
        nseg = int ((L/(0.1*lambda_f(100)) + 0.9)/2) * 2 + 1
    }
//---------------------------------------
```

読み込んだ形態データを確認するために、いくつかの手段が用意されている。

- **psection()** コマンドで section の情報を示すことができる。ある特定の section の情報が知りたい場合は、*section* **psection()**、全ての section の情報が知りたい場合は、**forall psection()** を用いる。SectionList を利用する場合は、**forsec** *section_list* **psection()**。

- **topology()** コマンドで、section がどのように **connect** されているかを示す。
- 図で示す。図で示す方法には用途に応じていくつかのやり方があるようであるが、一番基本的な方法は、Shape クラスを用いる方法である。
  objref *sh*
  sh = **new Shape**(*mode*)
  引数は mode で、0 の場合 diam、1 の場合 centroid、2 の場合直線で示される。mode は図の menu で変更可能。特定の section の色を変える方法は、*section shape*.**color**(*color*)。*color* は、2 が赤、3 が青である。 Point process をマークするには、*shape*.**point_mark**(*point_process*, *color*) が便利。

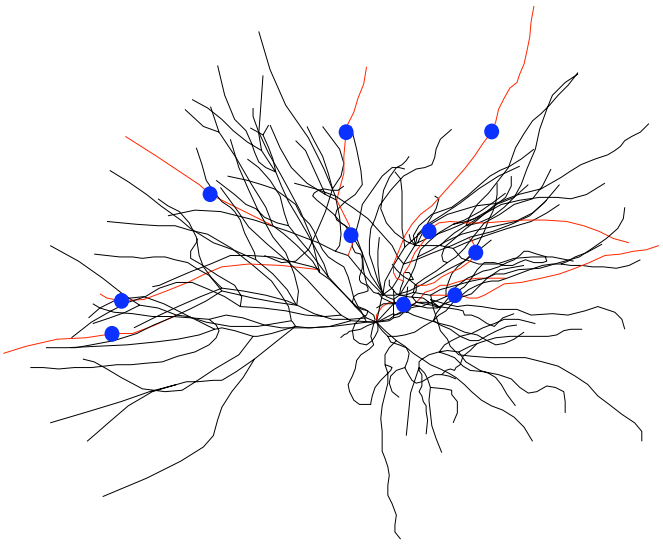
tc200 のモデルを読み込み、dendrite に random に 30 個のシナプスを作る例。

```
//--------------------------------
//  tc01.hoc
//
    load_file("nrngui.hoc")
    load_file("stdlib.hoc")
    load_file("tc200geo.hoc")

//--------------------------------
    nSyn=30
    tstop = 200
    dt = 0.01
    v_init = -65

//---------------------------------------
    objref dendritic, dendritic_only
    objref sh
    objref r, locvec, tx
    objref esyn[nSyn]
    objref sref
    objref nclist
    strdef s

//--------------------------------
//  class definition
//
    begintemplate TimeClock
        public getSec
        double jx[3]
        strdef datetime
        func getSec(){  // returns time in sec
```

```
            system("date '+%H %M %S'", datetime)
            sscanf(datetime,"%d%d%d", &jx[0], &jx[1], &jx[2])
            return (jx[0]*60 + jx[1])*60 + jx[2]
        }
    endtemplate TimeClock

//------------------------------
//  utilities
//
    func measureDist(){local dx localobj sr
        dx = 0
        sr = new SectionRef()
        while(sr.has_parent){
            dx += L
            access sr.parent
            sr = new SectionRef()
        }
        return dx
    }

//------------------------------
// Properties of dendrites
//
    dendritic = new SectionList()
    forall {
        insert hh
        Ra = 100
        dendritic.append()
    }
    soma dendritic.remove()

    forsec dendritic {
        d = lambda_f(100)
        nseg = int((L/(0.1*d))+0.9)/2*2+1
    }

// total dendritic length
    total_length=0
    forsec dendritic {
        total_length += L
```

```
    }
    print "Tolal dendrite length = ", total_length

//------------------------------
// Shape Plot
    sh = new Shape(1)
    sh.size(-150, 150, -150, 150)

//------------------------------
// stimulation
    objref stim
    stim = new NetStim(0.5)
    stim.interval = 50
    stim.number = 2
    stim.start = 50
    stim.noise = 0

    nclist = new List()

//------------------------------
// random generator
    tx = new TimeClock()
    r = new Random(tx.getSec())
    r.uniform(0, total_length)
    locvec = new Vector(nSyn)
    locvec.setrand(r)

    for k = 0, nSyn-1 {
        l=0
        l1 = 0
        found = 0
        forsec dendritic {
            if(found==0){
                l1 += L
                if(l1>locvec.x(k)){
                    lx = (locvec.x(k)-l)/L
                    sref = new SectionRef()
                    print secname(), measureDist()
                    sref.sec {
                        esyn[k] = new Exp2Syn(lx)
```

```
                        sh.color(2)
                    }
                    sh.point_mark(esyn[k],3)
                    nclist.append( new NetCon(stim, esyn[k], 0.0, 0, 0.001))
                    found = 1
                    break
                }
                l = l1
            }
        }
    }
    sh.flush()
    doEvents()

//-------------------------------
// main loop

    objref fp
    fp = new File()
    fp.wopen("a.dat")

    double a[2]

    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        fp.vwrite(2, &a)
    }
    fp.close()
    system("~/bin/gc a.dat")
//-----------------------------------------------
```



Windows/Cygwin の場合、**system(***cmd***)** は動くが、**system(***cmd*, *string***)** にバグがあるようだ。そのバグを回避するために、次のコードを使用する。

```
begintemplate TimeClock
    public getSec
    double jx[3]
    strdef datetime
    func getSec(){  // returns time in sec
        system("date '+%H %M %S'  > tmp001.txt")    // changed
        ropen("tmp001.txt")                  // added
        getstr(datetime)                     // added
        ropen()                              // added
        system("rm tmp001.txt")              // added
        sscanf(datetime,"%d%d%d", &jx[0], &jx[1], &jx[2])
        return (jx[0]*60 + jx[1])*60 + jx[2]
    }
endtemplate TimeClock
```

次に、n 次以上の dendrite 分岐に一様に synapse 入力がある場合を考える。mesureDis()、countBranch() を呼ぶことにより **access** している section が変化しないように改良。シナプスの数は 100 個。synaptic delay は正規分布の乱数を用い、平均 10 ms、標準偏差 1 ms としている。synaptic delay は負の数になるとエラーを起こすことに注意。

```
//------------------------------
//  tc02.hoc
//
    load_file("nrngui.hoc")
```

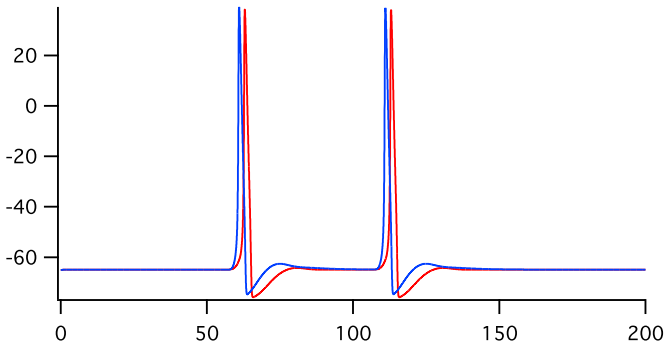
```
    load_file("tc200geo.hoc")

//------------------------------
    tstop = 200
    dt = 0.01
    v_init = -65

    nSyn=100       // number of synapses
    mdel = 10     // mean of synaptic delay
    sdel = 1      // sd of synaptic delay
    w = 0.001     // synaptic weight

    br = 5

//---------------------------------------
    objref dendritic, dendriticN
    objref sh
    objref r, vloc, vdel, tx
    objref esyn[nSyn]
    objref sref
    objref nc
    strdef s


//------------------------------
//  class definition
//
    begintemplate TimeClock
        public getSec
        double jx[3]
        strdef datetime
        func getSec(){  // returns time in sec
            system("date '+%H %M %S'", datetime)
            sscanf(datetime,"%d%d%d", &jx[0], &jx[1], &jx[2])
            return (jx[0]*60 + jx[1])*60 + jx[2]
        }
    endtemplate TimeClock

//------------------------------
//  utilities
```

```
//
    func measureDist(){local dx localobj sr, srsave
        dx = 0
        srsave = new SectionRef()
        sr = new SectionRef()
        while(sr.has_parent){
            dx += L
            access sr.parent
            sr = new SectionRef()
        }
        access srsave.sec
        return dx
    }

    func countBranch(){local bx localobj sr, srsave
        bx = 0
        srsave = new SectionRef()
        sr = new SectionRef()
        while(sr.has_parent){
            if(sr.nchild==2){
                bx += 1
            }
            access sr.parent
            sr = new SectionRef()
        }
        access srsave.sec
        return bx
    }

//------------------------------
// Properties of dendrites
//
    dendritic = new SectionList()
    forall {
        insert hh
        Ra = 100
        dendritic.append()
    }
    soma dendritic.remove()
```

```
    forsec dendritic {
        d = lambda_f(100)
        nseg = int((L/(0.1*d))+0.9)/2*2+1
    }

    dendriticN = new SectionList()
    total_length=0
    forall {
        if(countBranch()>=br){
            dendriticN.append()
            total_length += L
        }
    }

    print "total length of \">=", br, "-branch\" dendrites = ", total_length

//------------------------------
// Shape Plot
    sh = new Shape(1)

//------------------------------
// stimulation
    objref stim
    stim = new NetStim(0.5)
    stim.interval = 50
    stim.number = 2
    stim.start = 50
    stim.noise = 0

//------------------------------
// random generator

    tx = new TimeClock()
    r = new Random(tx.getSec())
    r.uniform(0, total_length)   // synaptic location
    vloc = new Vector(nSyn)
    vloc.setrand(r)

    r.normal(mdel,sdel*sdel)     // synaptic delay
    vdel = new Vector(nSyn)
```

```
    vdel.setrand(r)

    nc = new List()

    for k = 0, nSyn-1 {
        l=0
        l1 = 0
        found = 0
        forsec dendriticN {
            if(found==0){
                l1 += L
                if(l1>vloc.x(k)){
                    lx = (vloc.x(k)-l)/L
                    sref = new SectionRef()
                    print secname(), ", dist=", measureDist(), \
                          ", br=", countBranch(), ", delay=", vdel.x(k)
                    sref.sec {
                        esyn[k] = new Exp2Syn(lx)
                        sh.color(2)
                    }
                    sh.point_mark(esyn[k],3)
                    nc.append(new NetCon(stim, esyn[k], 0.0, vdel.x(k), w))
                    found = 1
                    break
                }
                l = l1
            }
        }
    }
    sh.flush()
    doEvents()

//------------------------------
// main loop

    objref fp
    double a[2]
    fp = new File()
    fp.wopen("a.dat")
```

```
    print "Sim Start"

    finitialize(v_init)
    fcurrent()
    while(t<tstop){
        fadvance()
        a[0] = t
        a[1] = soma.v(0.5)
        fp.vwrite(2, &a)
    }
    fp.close()
    system("~/bin/gc a.dat &")
// -- end of file
```



赤：tc01.hoc のトレース、青：tc02.hoc のトレース。

4 Spikes

4.1 NetCon

神経細胞集団の活動場合、膜電位 v の変化よりも活動電位あるいはスパイクのタイミング、数の方が解析の対象となってくる。スパイクの記録は NetCon の.record() を用いて行なうことができる。神経細胞が List クラスのオブジェクト cells であり、それぞれの神経細胞にあるポイントプロセス pp がスパイクを表すとする。

```
//------------------------------
    objref spikes
    objref netcon, vec
    spikes = new List()
    for i=0, cells.count()-1 {
        vec = new Vector()
        netcon = new NetCon(cells.object(i).pp, nill)
        netcon.record(vec)
        spikes.append(vec)
    }
    objref netcon, vec
//------------------------------
```

vec には、event が起きた実時間が記録される。スパイクのデータをファイルに記録するには、**Vector** クラスの関数**.vwrite(***fp***)** が便利である。［データ数: 4-byte integer］[4: 4-byte integer］［データ 0: 8-byte double］［データ 1: 8-byte double］.....［データ N: 8-byte double］という binary の形式でファイルに書かれる。2 番目の値 4 は、データが double であることを示す。

```
//------------------------------
    objref fp
    fp.wopen("a.nrb")
    for i=0, spikes.count()-1 {
        vec.vwrite(fp)
    }
    fp.close()
//------------------------------
```

4.2 Raster Plot の描き方

ここではラスタープロットを描く関数を示す。local な変数はできるだけ local に宣言し、他の関数との混乱を防ぐようにしている。使い方は、あらかじめ Graph オブジェクトを用意しておき、データファイル（たとえば a.nrb）を読みこむ。

objref *gr*

$gr =$ **new Graph()**
plotRaster(gr, "$a.nrb$")

```
//------------------------------
//  plotRaster(Graph, String)

    proc plotRaster(){ local nn, ts \
                    localobj fp, spikes, vec, spikey
        fp = new File()
        fp.ropen($s2)
        if(!fp.isopen()){ return }
        nn = 0
        ts = 0
        spikes = new List()
        while(!fp.eof()){
            vec = new Vector()
            vec.vread(fp)
            spikes.append(vec)
            nn += 1
            xx = vec.x(vec.size()-1)
            if(ts < xx){ ts = xx }
        }
        fp.close()
        $o1.view(0, 0, ts, nn, 100, 100, 400, 200)
        $o1.erase_all()
        for i=0, nn-1{
            spikey = spikes.object(i).c
            spikey.fill(i+1)
            spikey.mark($o1, spikes.object(i), "|", 6)
        }
    }
//------------------------------
```

この関数について特にコメントをすることはないが、NEURON における **Vector** の扱い方の参考になる。

5 NMODL

NMODL は NEURON 版の MODL (MOdel Description Language) である。MODL は NEURON だけでなく Genesis などの他のシステムでも用いられており、NMODL で書かれたファイル（mod ファイル）は、原則的には他のシステムでも利用可能である。mod ファイルの内部は、PARAMETER、STATE、ASSIGNED 等のいくつかのブロックに分かれている。ブロックの種類の中で NEURON ブロックは、NEURON に特有のものである。

mod ファイルは、nocmodl により C 言語のファイルに変換されて、その後 gcc（C コンパイラ）によりコンパイルされる。Windows の場合、nrnmech.dll が作成されて実行時に oc に組み込まれる。従って新たな mod ファイルを使用する場合にも、oc インタープリタ本体のコンパイルをする必要はない。MacOS と Linux の場合、special という名前の実行形式のスクリプトが生成される。このスクリプトは新たに作成されたダイナミックライブラリ（umac/.libs/libnrnmech.so）を読み込む。

実際の操作は、Windows の場合、mknrndll のアイコンをクリックし、mod ファイルがあるディレクトリを選択する。コマンドラインから操作を行う場合、mod ファイルが置かれているディレクトリに移動して、$NEURONHOME/bin にある mknrndll という shell script を使えばよい。ただしこの scirpt では NEURONHOME を示す変数 N が定義されていないので、

```
N=/cygdrive/c/nrn62
export N
```

の 2 行を付け加えなくてはならない。

Mac の場合は、mod ファイルのあるディレクトリ（フォルダ）のアイコンを mknrndll のアイコンに重ねる。コマンドラインからの場合は、mod ファイルが置かれているディレクトリに移動し、open -a mknrndll . とする（"."は言うまでもなく current directory の意味)。そのディレクトリ内に umac（Mac Universal BInary の意で、ppc と intel 系の両方の CPU で稼働できる）という名前のディレクトリが作成され、その中に special という実行可能なファイルが作られる。

Mac の場合でも、ソースコードからコンパイルした場合には、アイコンや application bundle が作成されない場合がある。そのような場合は、nrniv などが含まれる bin ディレクトリにある nrnivmodl という名前のシェルスクリプトを使用して、nrnivmodl . とする。"."はあってもなくても構わないようだ。

5.1 Blocks

mod ファイルでコメントは、COMMENT と ENDCOMMENT で挟まれた行、あるいは":"で始まる行である。また VERBATIM と ENDVERBATIM に挟まれた行は、nocmodl で処理されることなくそのまま C 言語ファイルになる。

5.1.1 NEURON block

SUFFIX でモジュールの名前を定義する。RANGE で外からアクセスできる変数を示す。

5.1.2 ASSIGNED block

mod ファイル外で値をいれる変数、あるいは mod ファイル内で式の左側に来る変数。

5.1.3 STATE block

微分法的式などで用いられる変数。変数は ASSIGNED と STATE の両方で宣言することは出来ない。

5.1.4 INITIAL block

初期化ブロック。

5.1.5 BREAKPOINT block

実際の計算の場所。微分方程式を解く場合は SOLVE を用いる。方法としては、cnexp、runge、euler、derivimplicit などが使用可能。通常は cnexp。runge は求められる以上の精度（時間がかかる）。

5.1.6 DERIVATIVE block

ここには常微分方程式がくる。

5.1.7 NET_RECEIVE block

NetCon のために拡張された部分らしい。event が起きた時に何をするかを記述する部分。

5.2 常微分方程式を解いてみる

試しに簡単な常微分方程式の数値解を求めるのに NEURON を使用してみる。まず簡単な常微分方程式で試してみる。

$$z'' = -z$$

初期値は、$z(0) = 0, z'(0) = 1$ とします。$z1 = z'$ と置くと、

$$z' = z1$$
$$z1' = -z$$

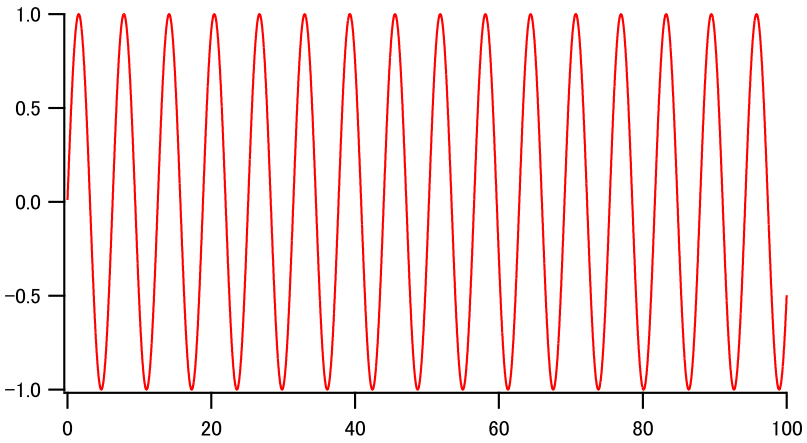
連立の微分方程式として表すことが出来る。

```
:----------------------------------------
: m01.mod:  a simple ODE   z(t)'' = -z(t)
  NEURON{
      SUFFIX m01
      RANGE z
  }
  STATE {z z1}
  INITIAL{
      z = 0
      z1 = 1
```

```
  }
  BREAKPOINT{
      SOLVE zstates METHOD cnexp
  }
  DERIVATIVE zstates {
      z' = z1
      z1' = -z
  }
:---------------------------------------
```

この mod ファイルを試すための hoc ファイル。

```
//---------------------------------------
// m01.hoc:   test file for mo1.mod

    create soma
    soma insert m01

    tstop = 100
    dt = 0.01
    v_init = -65

    objref fp
    fp = new File()
    fp.wopen("a.dat")
    double a[2]

    finitialize(v_init)
    fcurrent()

    while(t<tstop){
        fadvance()
        a[0] = t
        soma a[1] = z_m01
        fp.vwrite(2,&a)
    }
    fp.close()
    WinExec("gc a.dat")
//---------------------------------------
```



5.3 IntervalFire

NEURON Book に掲載されている Interval File を掲載する。人工細胞であり、変数 m は、微分方程式

$$\frac{dm}{dt} = (m_\infty - m)/\tau$$

に従い、その値が 1 を越えると fire して $m = 0$ に戻る。 外からに入力があると、m の値が w だけ変化する。この式は解析的に解けて、$m(t = 0) = 0$ とすれば、

$$m = m_\infty(1 - \exp(-\frac{t}{\tau}))$$

で示される。$t = invl$ の時、$m = 1$ であるから、

$$m_\infty = \frac{1}{1 - \exp(-\frac{invl}{\tau})}$$

である。

```
: ------------------------------------------------------------------------
:  The NEURON Book pp. 309- 310
NEURON {
    ARTIFICIAL_CELL IntervalFire  : name of the module
    RANGE tau, m, invl            : accessible variables
}
PARAMETER {
    tau = 5 (ms)   <1e-9,1e9>
    invl = 10 (ms) <1e-9,1e9>
}
ASSIGNED {                        : non-accessible variables
    m
    minf
```

```
    t0 (ms)
}
INITIAL {
    minf = 1/(1 - exp(-invl/tau)) : so natural spike interval is invl
    m = 0
    t0 = 0
    net_send(firetime(), 1)       : set a self-event
                                  : after time period of firetime()
}
NET_RECEIVE (w) {
    m = M()
    t0 = t
    if (flag == 0) {              : *** event triggered by others ***
        m = m + w
        if (m > 1) {
            m = 0
            net_event(t)          : issue event
        }
        net_move(t+firetime())    : move the next event to t + firetime()
    }else{                        : *** self-triggered event ***
        net_event(t)              : issue event
        m = 0
        net_send(firetime(), 1)   : next self-event
    }
}
FUNCTION firetime()(ms) {          : m < 1 and minf > 1
    firetime = tau*log((minf-m)/(minf - 1))
}
FUNCTION M() {                     : to monitor m-value
    M = minf + (m - minf)*exp(-(t - t0)/tau)
}
: ------------------------------------------------------------------------
```

 NetCon に関しては十分な Reference がないので、この例は NetCon をどのように使用すればよいかを理解するために、とても参考となる例である。ここで使用されている NetCon に関係する関数は下記の通り。

- **net_event**(t_1) 時間 t_1 に event を発生させる。event は NetCon で定義された相手全てに伝えられる。
- **net_send**($t_2, flag$) 現時点 t より t_2 後に event を発生させる。$flag = 0$ の場合は他に、$flag = 1$ の場合は self へ event が送られる。
- **net_move**(t_3) 詳細は不明。次に起きる予定の self event を t_3 へと移動させるらしい。

INITIAL block では、m_∞ の値を求めるとともに、m の値を初期化している。t_0 は event が発生からの時間を示す。さらに INITIAL block で **net_send()** を用いて次の event が起きるように設定している。firetime() は次の event が起きるまでの時間を計算する関数。ここでは $invl$ でもよい。

event が起きた場合、NET_RECEIVE block が実行される。引数 w は NetCon の weight であり、正負の値を取ることができる。何を実行するはか、m の値に依存するが、m の値は常々計算されている訳ではないので、まずは明示的に m の値を（場合によっては計算して）入れる。event が起きた時刻として t_0 をアップデートする。

$flag$ は event が自己由来か他由来かを示すフラグで、0 の場合は他、1 の場合は自己であることを示す。他からの event を受け取った場合、新しい m の値は $m+w$ となる。もしそれが 1 を越えていたなら、**net_event()** で event を発生させる。m の値が変化したので、次の event 予定をキャンセルして、**net_move()** を用いて firetime()+t にセットし直す。

自己から発せられた event の場合は、event を発生させ、次の自己宛の event を **net_send()** でセットする。

m の値は NET_RECEIVE block でしか計算されない。m の挙動をしらべるため、関数 M を定義している。M の値はアクセスされるたびに更新されている。関数 X の返す値は、X() ではなく X で示すことができる。

5.4 Synaptic transmission

先ず標準の Point process である **exp2syn** の内容を検討する。event が起きたた場合に、2 つの指数関数の和で表されるコンダクタンスの変化を示す。

```
: ----------------------------------------------------------------------------
:   nrn-6.1src/nrnoc/exp2syn.mod

NEURON {
    POINT_PROCESS Exp2Syn
    RANGE tau1, tau2, e, i
    NONSPECIFIC_CURRENT i
    RANGE g
    GLOBAL total
}
UNITS {
    (nA) = (nanoamp)
    (mV) = (millivolt)
    (uS) = (microsiemens)
}
PARAMETER {
    tau1= 0.1 (ms) <1e-9,1e9>
    tau2 = 10 (ms) <1e-9,1e9>
    e=0 (mV)
```

```
}

ASSIGNED {
    v (mV)
    i (nA)
    g (uS)
    factor
    total (uS)
}
STATE {
    A (uS)
    B (uS)
}
INITIAL {
    LOCAL tp
    total = 0
    if (tau1/tau2 > .9999) {  : avoid tau1==tau2
        tau1 = .9999*tau2
    }
    A = 0
    B = 0
    tp = (tau1*tau2)/(tau2 - tau1) * log(tau2/tau1)
    factor = -exp(-tp/tau1) + exp(-tp/tau2)
    factor = 1/factor
}

BREAKPOINT {
    SOLVE state METHOD cnexp
    g = B - A
    i = g*(v - e)
}
DERIVATIVE state {
    A' = -A/tau1
    B' = -B/tau2
}
NET_RECEIVE(weight (uS)) {
    A = A + weight*factor
    B = B + weight*factor
    total = total+weight
}
```

```
: ----------------------------------------------------------------------
```

 $t = 0$ の時に入力があったとすると、コンダクタンス g の時間的な変化は、

$$g = factor\left(\exp(-\frac{t}{tau2}) - \exp(-\frac{t}{tau1})\right)$$

で表される。$factor$ は、g の最大値が 1 となるようにするための係数である。$factor$ の値を求めるために、g が極値をとる t の値 t_p を求める。

$$\frac{dg}{dt} = factor\left(-\frac{1}{tau2}\exp(-\frac{t}{tau2}) + \frac{1}{tau1}\exp(-\frac{t}{tau1})\right)$$

$t = t_p$ の時 $dg/dt = 0$ であるから、

$$\begin{aligned}
\frac{1}{tau2}\exp(-\frac{t_p}{tau2}) &= \frac{1}{tau1}\exp(-\frac{t_p}{tau1}) \\
tau1\exp(-\frac{t_p}{tau2}) &= tau2\exp(-\frac{t_p}{tau1}) \\
\log(tau1) - \frac{t_p}{tau2} &= \log(tau2) - \frac{t_p}{tau1} \\
\left(\frac{1}{tau1} - \frac{1}{tau2}\right)t_p &= \log(\frac{tau2}{tau1}) \\
t_p &= \frac{tau1 * tau2}{tau2 - tau1}\log(\frac{tau2}{tau1})
\end{aligned}$$

従って、

$$factor = 1/\left(\exp(-\frac{t_p}{tau2}) - \exp(-\frac{t_p}{tau1})\right)$$

 NET_RECEIVE block で行なっていることは、状態の設定し直し（一種の初期化とも言える）であり、各 time step での計算は、BREAKPOINT block（実態は DERIVATIVE block）で行なわれている。

 元のソースコードでは、A = A + weight*factor の部分が、state_discontinuity(A, A+weight*factor) と書かれている。この state_discontinuity() という関数は、微分方程式で変数が abrupt に変化した場合のトラブルを回避するためのものであるが、NET_RECEIVE block の改良により用いる必要はなくなっている。

par

5.5 Synaptic plasticity

これも NEURON Book からの転載であるが、Use-dependent synaptic plasticity の例を示す。このコードは、nrn-6.1/share/examples/niniv/netcon/gsyn.mod と同じもの。上記の例から推測されるように、NET_RECEIVE block で、event が起きればその時刻を記録しておき、次の event が起きた時に前の event からの時間によってシナプス結合の強度を調節するようにすればよい。変数の記憶には、**NetCon** の機能が用いられる。

```
: ----------------------------------------------------------------------
:  The NEURON Book pp. 281-282
: ----------------------------------------------------------------------
```

```
:  The NEURON Book pp. 281-282
NEURON {
    POINT_PROCESS GSyn
    RANGE tau1, tau2, e, i
    RANGE Gtau1, Gtau2, Ginc
    NONSPECIFIC_CURRENT i
    RANGE g
}
UNITS {
    (nA)    =  (nanoamp)
    (mV)    =  (millivolt)
    (umho)  =  (micromho)
}
PARAMETER {
    tau1    = 1    (ms)
    tau2    = 1.05 (ms)
    Gtau1   = 20   (ms)
    Gtau2   = 21   (ms)
    Ginc    = 1
    e       = 0    (mV)
}
ASSIGNED {
    v   (mv)
    i   (nA)
    g   (umho)
    factor
    Gfactor
}
STATE {
    A (umho)
    B (umho)
}
INITIAL {
    LOCAL tp
    A = 0
    B = 0
    tp = (tau1*tau2)/(tau2-tau1) * log(tau2/tau1)
    factor = -exp(-tp/tau1) + exp(-tp/tau2)
    factor = 1/factor
    tp = (Gtau1*Gtau2)/(Gtau2-Gtau1) * log(Gtau2/Gtau1)
```

```
    Gfactor = -exp(-tp/Gtau1) + exp(-tp/Gtau2)
    Gfactor = 1/Gfactor
}
BREAKPOINT {
    SOLVE state METHOD cnexp
    g = B - A
    i = g * (v - e)
}
DERIVATIVE state {
    A' = -A/tau1
    B' = -B/tau2
}
NET_RECEIVE (weight (umho), w, G1, G2, t0 (ms)){
    G1 = G1*exp(-(t-t0)/Gtau1)
    G2 = G2*exp(-(t-t0)/Gtau2)
    G1 = G1 + Ginc * Gfactor
    G2 = G2 + Ginc * Gfactor
    t0 = t
    w = weight * (1 + G2 - G1)
    A = A + w*factor
    B = B + w*factor
}
: ------------------------------------------------------------------------
```

 NET_RECEIVE block の引数。引数の数が 1 の場合、**NetCon** の weight が渡される。引数が $n+1$ 個の場合、最初の引数は、**NetCon** の weight であり、残りの引数はこの mod の変数を **NetCon** で記憶しておくために用いられる。これらの引数は、通常の"call by value"ではなく、"call by reference"で渡されるので、NET_RECEIVE block で変更された値は **NetCon** で保存される。

5.6 NMDA receptor channel

NMDA receptor channel は Mg^{2+} block により電位依存性と活動依存性を示す。下記の mod ファイルは、Gasparini et al, J Neurosci 24:11046-11056, 2004 による。state_discontinuity() を取除く等の改変を行なっている。

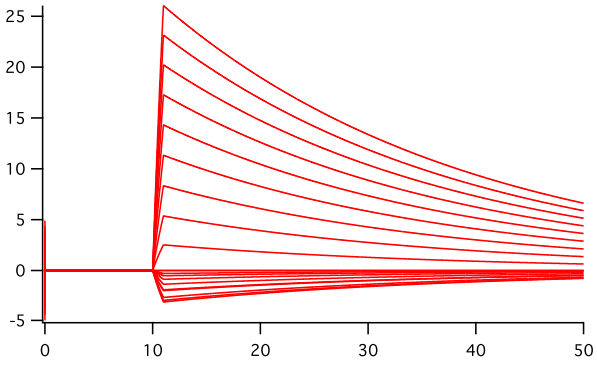
http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=44050

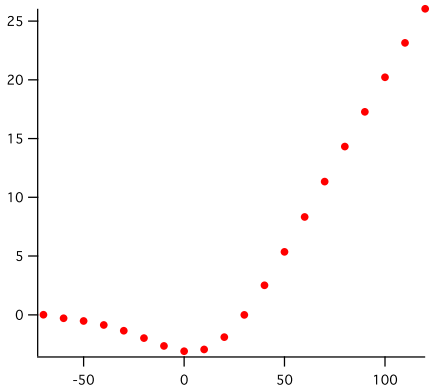
```
: ------------------------------------------------------------------------
NEURON {
    POINT_PROCESS nmdanet
    RANGE R, g, mg
    NONSPECIFIC_CURRENT i
```

```
    GLOBAL Cdur, Alpha, Beta, Erev, Rinf, Rtau
}
UNITS {
    (nA) = (nanoamp)
    (mV) = (millivolt)
    (umho) = (micromho)
    (mM) = (milli/liter)
}
PARAMETER {
    Cdur   = 1  (ms)     : transmitter duration (rising phase)
    Alpha  = 0.35 (/ms)  : forward (binding) rate
    Beta   = 0.035 (/ms) : backward (unbinding) rate
    Erev   = 0 (mV)      : reversal potential
    mg     = 1 (mM)      : external magnesium concentration
}
ASSIGNED {
    v      (mV)          : postsynaptic voltage
    i     (nA)          : current = g*(v - Erev)
    g     (umho)        : conductance
    Rinf                 : steady state channels open
    Rtau   (ms)          : time constant of channel binding
    synon
}
STATE {Ron Roff}
INITIAL {
    Rinf = Alpha / (Alpha + Beta)
    Rtau = 1 / (Alpha + Beta)
    synon = 0
}

BREAKPOINT {
    SOLVE release METHOD cnexp
    g = mgblock(v)*(Ron + Roff)*1(umho)
    i = g*(v - Erev)
}

DERIVATIVE release {
    Ron' = (synon*Rinf - Ron)/Rtau
    Roff' = -Beta*Roff
}
```

```
FUNCTION mgblock(v(mV)) {
    TABLE
    DEPEND mg
    FROM -140 TO 80 WITH 1000
    mgblock = 1 / (1 + exp(0.062 (/mV) * -v) * (mg / 3.57 (mM)))
}
NET_RECEIVE(weight, on, nspike, r0, t0 (ms)) {
    if (flag == 0) { : a spike, so turn on if not already in a Cdur pulse
        nspike = nspike + 1
        if (!on) {
            r0 = r0*exp(-Beta*(t - t0))
            t0 = t
            on = 1
            synon = synon + weight
            Ron =  Ron + r0
            Roff = Roff - r0
        }
        net_send(Cdur, nspike)
    }
    if (flag == nspike) { : if this associated with last spike then turn off
        r0 = weight*Rinf + (r0 - weight*Rinf)*exp(-(t - t0)/Rtau)
        t0 = t
        synon = synon - weight
        Ron = Ron - r0
        Roff = Roff + r0
        on = 0
    }
}
: ------------------------------------------------------------------------
```

n12.hoc を NMDA receptor channel に変更して Voltage-clamp mode で I-V relationship を得ると、Ma^{2+} block の効果がわかる。


25
20
15
10
5
0
-5
0
10
20
30
40
50


25
20
15
10
5
0
-50
0
50
100

6 NEURON の内部

6.1 初期化

finitialize()

6.2 Integration の方法

6.3 Graph

付録 A　インストールの仕方

現在の最新バージョンは 7.0（2008 年 9 月から）。以前のバージョンと比較して、マルチコア対応になりバグが修正されている模様である。

インストールの方法は、http://www.neuron.yale.edu/neuron/install/install.html に記載されている。当然のことながら OS によって異なる。Unix/Linux で開発されてきたプログラムなので、OS によってうまく働かない機能があるようだ。

A.1　Windows の場合

Windows 用とは言うものの、Unix/Linux に似た環境を提供する Cygwin のサブセットを用いている。nrn-*n.n*.setup.exe をダウンロードする。そのプログラムを起動しインストールを行う（この作業は管理者権限が必要)。インストールされた先を環境変数 PATH に加える。（コントロールパネル → システム → 詳細設定 → 環境変数で、ユーザー環境変数に、例えば、PATH=%PATH%;C:¥nrn70¥bin と設定する。%PATH% はシステムとして設定された PATH)。

注意すべき点として、cygwin では PATH の中に空白文字があるとトラブルの原因となるので、インストール先は、C:¥Program Files の中に nrn70 を置くのではなく、C:¥nrn70 とか C:¥WIN32APP¥nrn70 といった所にするのがよい。

cygwin のサブセットがインストールされるので、C 言語のプログラムをコンパイルし走らせる事が出来る。gcc.exe は含まれているが、g++.exe は含まれていないので、C++ 言語のプログラムの処理は出来ない。

A.2　Mac OS の場合

Mac OS の場合は、dmg ファイルをダウンロードして解凍し、NEURON-*n.n* のフォルダを Applications フォルダにコピーするだけで完了。

Mac OS の場合、開発環境がインストールしてあれば、ソースコードから簡単にコンパイルすることが出来る。開発環境は OS の DVD に含まれているが、デフォールトではインストールされていない。NEURON のコンパイルの方法は、上記のウェブサイトに書かれており、その通りすれば出来るはずである。Intel CPU のマシンであれば、–enable-carbon のオプションは使用しない。トラブルがなければ 20 分程度で完了できる。–enable-carbon のオプションをつけると umac ディレクトリが作成され、–enable-carbon のオプションをつけないと i686 ディレクトリが作成されるようである。

.bash_profile ファイルで、PATH を/Applications/NEURON-*n.n*/nrn/umac/bin に通しておくと便利である。あるいは、$HOME に bin ディレクトリをつくって PATH を通しておき、そこに nrniv 等の symbolic link を置く方法もある。

```
ln -s /Applications/NEURON-7.0/nrn/i686/bin/nrniv nrniv
ln -s /Applications/NEURON-7.0/nrn/i686/bin/nrngui nrngui
```

A.3 Linux の場合

インストールの方法は、Mac OS とほぼ同じ。

A.4 計算速度

計算速度に関しては正確に測定したことはないが、CPU のスペックが同じ程度であれば MacOS よりも Windows の方が速いようである。MacOS の場合は、異なる CPU に使用できるようにしているために遅くなっているのかと想像される。実際、ソースコードを–enable-carbon のオプションなしでコンパイルすると、Intel CPU 専用のプログラムが作成され、実行速度は 1.5〜2 倍となった（MacBook Pro の場合)。また Core 2 Duo や Xeon などでは 64-bit で計算を行うようにコンパイルするのがよい（どのように設定する？)。

NEURON のコンパイルには、Windows/MacOS/Linux のいずれでも GNU C/C++ コンパイラを使用している。 Intel コンパイラを使用すれば速度面での改善が予測される。iv（InterViews）なしではコンパイル可能であるとの報告あり[*2]。

付録 B hoc ファイル用のテキストエディタ

hoc ファイルの作成・編集にはテキストエディタを用いる。notepad（Windows）や TextEdit（Mac OS）などのテキストエディタでよいが、行番号がわかりやすい方がよく、また定義が色でハイライト（syntax highlight）されると便利である。

B.1 Emacs 系テキストエディタ

プログラマ好みのエディタである emacs は、様々な機能を有しており、また機能拡張も可能である。Windows の場合、meadow と呼ばれている[*3]。Mac OS の場合は、Carbon Emacs を用いるのが簡単[*4]。hoc ファイルの syntax highlight を設定をするには、nrnhoc.el がある[*5]。

なお、Windows には Emacs の簡易版に相当する xyzzy.exe というプログラムがある。nrnhoc.el をそのまま試してみたが、エラーとなり動かなかった。

B.2 Smultron

MacOS には Smultron というフリーのテキストエディタがあり[*6]、hoc ファイルの定義が色でハイライトされる。文字コードが日本語の使用も可（UTF-8)。

[*2] http://www.neuron.yale.edu/phpBB/viewtopic.php?=6&t=116
[*3] http://www.meadowy.org/meadow/
[*4] http://homepage.mac.com/zenitani/emacs-j.html
[*5] http://homepages.inf.ed.ac.uk/sterratt/progs/neuron
[*6] http://www.tuppis.com/smultron/

付録 C　References

- The NEURON Book. Carnevale NT & Hines ML, Cambridge University Press, 2006.
  NEURON の開発者による説明書であり、NEURON の説明書としてはもっともまとまった本。ただしこの本だけでは情報が十分でないところもある。
- Programmer's Reference。命令、関数等の詳細を調べるために便利。少ないながら example もある。zip ファイルをダウンロードして使用することも可能。
  http://www.neuron.yale.edu/neuron/docs/help/contents.html
- The NEURON Forum。NEURON に関する質問サイト。多くの質問に Carnevale 自身が回答している。プログラムのちょっとした工夫等が書かれている。
  http://www.neuron.yale.edu/phpBB/
- ModelDB。データ、プログラムのデータベース。大部分が NEURON 用のもの。
  http://senselab.med.yale.edu/modeldb/
- Hines ML & Carnevale NT (2004) Discrete event simulation in the NEURON environment. Neurocomputing Volumes 58-60, Pages1117-1122.