A Simplifed Introduction to NEURON Simulator
— (私的) NEURON 事始め—
第 2 版 (Python 版)

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

13-Jan-2012

13-Jan-2012: Python version

11-June-2010: correction of bugs in n12.hoc

12-April-2010: minor changes

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。これまでに 10 年以上の実用の歴史があり、多くの論文で利用されている。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 はそのような数値計算を行う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” (Cambridge 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。わたし自身、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。

（私的）事始め (第 1 版) 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。確かに oc はそれなりに便利なインタープリタではあるが、oc を他の状況で使うことはない。そのような理由のためか、NEURON を同じくインタープリタである Python から使用できるように開発が進められて来た。Python は科学計算用に広く用いられており、多くのライブラリが使用できることも大きな魅力である。今回、第 2 版を作成するにあたっては、これまでの誤りに訂正を加えるとともに、主言語を oc から Python に変更した。Python には Version 2 の系列と Version 3 の系列があり、Version 3 には後方互換性が保たれていない。この解説書では、Python version 2.7 を使用している。

1.1 説明書の対象者

この説明書は、ある程度プログラミングに経験のある人をターゲットにしている。例題を多く含めたので、初心者でも書いてある内容を再現できるが、C/C++ 言語並びに数値計算についての基本的な知識があれば、更に発展した NEURON の使用ができると期待している。

1.2 NEURON シミュレータの特徴

NEURON シミュレータの特徴として 3 つの点があげられる。

1. NEURON の使用により、非常に数多くの連立微分方程式の数値解を比較的簡単な操作で求めることができる。

2. イオンチャネル等のコンポーネントを新たに作製し、mknrnmodl（システムによっては mknrndll）というツールによりプログラムに組み込むことができる。
3. 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となる。

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等の検討。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 PC 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　シミュレーションの基本

2.1　膜電位

NEURON シミュレータは、神経細胞の電位変化のシミュレーションを得意とするプログラムである。細胞をコンデンサと考え、それに蓄えられる電荷の量により膜電位が生じている、という考え方が基本となっている。蓄えられている電荷量を Q、静電容量（キャパシタンス）を C とすると、電位 V との関係は、

$$Q = CV$$

で表される。C を定数として扱い、この式の両辺を時間 t で微分すると、

$$\frac{dQ}{dt} = C\frac{dV}{dt}$$

となる。左辺は、細胞に出入りする電荷量の時間的変化量、すなわち電流 I である。電気生理学の習わしとして、細胞内に流れる電流を負の値として示すので、

$$I = -\frac{dQ}{dt}$$

である。従って、電流 I と電位 V の関係は、

$$\frac{dV}{dt} = -\frac{1}{C}I$$
$$V(t) = V(0) - \frac{1}{C}\int_{\tau=0}^{\tau=t} I d\tau$$

となる。すなわちこの式は、細胞の膜電位の初期値と、細胞に出入りする電流がわかれば、膜電位の時間的変化は計算できることを示している。

神経細胞は複雑な形をしている。特に樹状突起は長く伸びており、神経細胞を単なる球体のように扱うことは出来ない。このため NEURON では神経細胞を球体、円柱体などの集まりとして考え、それぞれのコンパートメントについて電位の微分方程式を解くという手法をとっている。またそれぞれの構成コンパートメントに入出する電流にはさまざまな種類の電流が含まれる。電位依存性イオンチャネルを介する電流、神経伝達物質依存性イオンチャネルを介する電流（シナプス電流がこれに相当する）、leak 電流（常に開いているイオンチャネルを流れる電流）、近接のコンパートメントとの電流等が考えられる。神経細胞には多種類のイオンチャネルが存在しているので、電流の種類数はかなりの数になると予想される。しかし現実には、それぞれのイオンチャネルの分布などの情報はほとんどないことから、数種類のイオンチャネルでシミュレーションを行うことが普通である。i コンパートメント、j 電流の考えを含めると、冒頭で示した式となる。

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

2.2　イオンチャネル

2.3　微分法的式の計算方法

微分方程式の数値的解法は、コンピュータの歴史の中でも長年にわたっていろいろな手法が開発されて来ている。計算のステップサイズが一定である fixed step の方法としては、backward Euler が default の方法

であり、Crank-Nicolson を指定することもできる (Nicholson ではなく Nicolson が正しい)。また adaptive step の方法 (計算のステップサイズが状況に応じて調節される) である CVODE や DASPK も使用できる。それぞれの手法の特徴は、

- Forward Euler: simple, inaccurate and unstable
- Backward Euler: inaccurate but stable
- Crank-Nicolson: stable and more accurate
- Adaptive integration: fast or accurate, occasionally both

と表現されている。

CVODE は、アメリカ Lawrence Livermore National Laboratory で開発された微分方程式解法ライブラリ SUNDIAL (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) の一部である。古典的なライブラリ (例えば LSODE や VODE を含む ODEPACK) が Fortran で書かれているのに対して、このライブラリは C/C++ で書かれている。

今後は、CVODE を使用することが主流となると予想されるが、外部刺激を加える場合等では、CVODE が adaptive integrator であることに注意しなくてはならない。例えば、0.1 ms の刺激を入れたいのだが、安定した状態では integration の step size がそれより長くて、刺激入力を無視するということが起こりうる (この問題は、max step を設定することにより回避可能)。また CVODE は、Voltage-clamp シミュレーションには使用できない。

2.4 C/C++ 言語

`NEURON` を使うにあたって、通常 C/C++ を直接必要とすることはほとんどない。mod ファイルを書く場合に、C プログラムを埋め込む形で書く場合がある。mod ファイルは一旦 C ファイルに変換されて gcc によりコンパイルされる。

2.5 Python

Python に関しては莫大な量の情報がネット上に掲載されているので (ほとんど英語ではあるが)、それらを参照されたい。基本的な特徴として、次の様な点があげられる。

- interactive に使うことも、スクリプトファイルの実行もできる。`NEURON` を使う場合、基本的にはスクリプトファイルを実行することになる。
- スクリプトファイルは、テキストファイルであり、エディタで編集可能。文字コードには utf-8 を使用する。
- Python ファイルでは、インデントがプログラム制御に用いられている。インデントには半角スペース 4 文字を使用し、Tab は使用しない。全角の空白を用いてはならない。
- 基本的なデータタイプは、整数 `int` と実数 `float` である。`float` は単精度 (4 bytes) ではなく倍精度 (8 bytes) が用いられる。
- list がよく用いられる。一種の配列であるが、要素の種類が異なっていてもよい。
- Object-oriented programming をよく使用する。
- クラス変数、クラス関数は原則的に public であり、C++ の場合のように private といったアクセス制

限を付すことは出来ない。

3 NEURON を動かす

Python 環境として NEURON を立ち上げるには、neuron -python もしくは nrniv -python というように、-python のオプションを付ける。スクリプトの内容にもよるのであろうが、この解説書の例題の場合、mod を追加したものでなければ、python xx.py でも問題ない様である。

はじめの数個の例は、NEURON とあまり関係がない。主にファイル IO を学ぶ。

```
#-----------------------------------------------
# n01.py:   a C-like python program
s = "Hello, World!"
for i in range(3):
print i, 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。また複数行にわたるコメントは、"""と"""で囲む。文字列の変数を使用する場合には、変数の宣言は不要。文字列は、" "もしくは’ ’で囲む。Python の制御はインデントに大きく依存している。インデントは space 4 文字で行い、tab は使用しない。この例題の場合、i は 0 から 2 まで変化するので、Hello, World!の表示は 3 回繰り返される。

ファイルへの出力の場合は、次のように行なう。

```
#-----------------------------------------------
# n02.py: a C-like python program with text file output
s = ’Hello, World!’
fs = open(’a.txt’, ’w’)    # open for text write
for t in range(3):
    s1 = str(t) + ’ ’ + s + ’\n’
    fs.write(s1)
fs.close()
#-----------------------------------------------
```

 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。ファイルを開く場合は、open(*filename*, *mode*) を用い、閉じる場合は引数なしで close() を使用する。*mode* は、書き込みの場合は"w"、読み込みの場合は"r"、テキストの場合は"t"、バイナリの場合は"b"となる。たとえばバイナリの書き込みの場合、"wb"となる。default ではテキストなので、"t"は普通使用しない。C 言語の場合と異なり、read()、write() は引数が 1 つのみである。従って出力の場合、文字列を作り上げてから書き出す。数値変数の文字列への変換には str() を使用する。文字列をつなぐためには、+ を使用する。’\n’ は改行。

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[*1]。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.py:   drowing a small graph
import os
import math

fs = open('a.txt','w')    # open a file for "text write"
tstop = 200
for t in range(tstop):
    s = str(t) + ' ' + str(math.sin(0.1*t)) + '\n'
    fs.write(s)
fs.close()
os.system('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、os.system() を利用する。

binary ファイルを使用する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。一方、テキストエディタでは内容がわからない欠点がある。

```
#-----------------------------------------------
# n04.py:   drowing a small graph (binary file version)
import os
import math
import array

fs = open('a.dat','wb')    # open a file for "binary write"
tstop = 200
a=array.array('d', [0.0]*2) # array of double
for t in range(tstop):
    a[0] = t
    a[1] = math.sin(0.1*t)
    fs.write(a)
fs.close()
os.system('gc a.dat')
#-----------------------------------------------
```


書き込み用のバッファとして a を array ライブラリを使用して作っている。

[*1] http://www.nips.ac.jp/huinfo/documents/index.htm

 binary の読み込みは、C/C++ の場合と多少とも異なる。例えば、倍精度実数（8 バイト）が 20 個並んでいるデータを読み込む場合、struct ライブラリの unpack を用いて次の様にすればよい。a は List であり、先頭の値は a[0] となる。

```
a = struct.unpack('20d', fs.read(20*8))
```

3.1 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 h.Section() というステートメントが用いられる。内部的には Section クラスのインスタンスが作成される。先ず初めに 1 コンパートメントのモデルを作成し、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。この例題は簡単なスクリプトであるが、プログラムの観点からはこの例題は基本的ひな形である。

```
#-----------------------------------------------
#  n05.py:  a simple hh cell
#
import os
import array
from neuron import h

soma = h.Section()
soma.insert('hh')

cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)

fs = open('a.dat', 'wb')
a = array.array('d',[0.0]*2)

tstop = 200
v_init = -65

h.finitialize(v_init)
while h.t < tstop:
    cvode.solve()
    a[0] = h.t
    a[1] = soma(0.5).v
    fs.write(a)
```

```
fs.close()
os.system('gc a.dat 1 &')
#-----------------------------------------------
```

NEURON 関係の関数などは h を付けることによりアクセスできる。integrator には CVODE を使用し、atol( ) で絶対的な誤差範囲を指定している。相対的な誤差範囲を使用する場合は、rtol() を使用する。ある種の変数のみ誤差範囲を変える場合は、atolscale() などを用いる。CVODE を使用した場合、ステップサイズは誤差範囲の設定に基づいて自動的に設定される。決まったステップ毎に結果を知りたい場合には、while ループを下記のようにすればよい。

```
dt = 0.01  # for example
while h.t < tstop:
    cvode.solve(h.t+dt)
    a[0] = h.t
    a[1] = soma(0.5).v
    fs.write(a)
```

n05.py 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、やや非現実的ではあるが leak 電流の性質を変更してみる。n05.py 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。

```
#-----------------------------------------------
# n06.py:  a simple hh cell, with increased leak
#
import os
import array
from neuron import h

soma = h.Section()
soma.insert('hh')
soma.el_hh = -30.0
tstop = 200.0

dt = 0.01
v_init = -65.0

cvode = h.CVode()
```

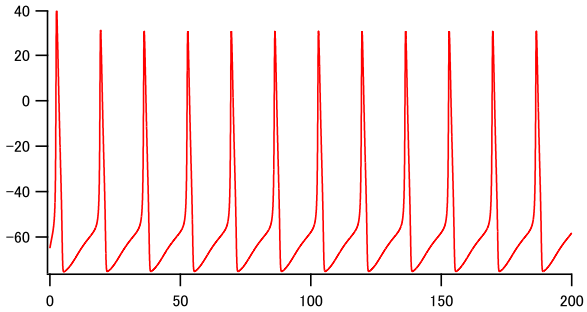


Fig.1　n06.py: a simple hh-type neuron

```
cvode.active(1)
cvode.atol(1.0e-5)

fp = open('a.dat', 'wb')
a = array.array('d', [0.0]*2)

h.finitialize(v_init)
while h.t < tstop:
    cvode.solve()
    a[0] = h.t
    a[1] = soma(0.5).v
    fp.write(a)

fp.close()
os.system('gc a.dat 1 &')
#-----------------------------------------------
```

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 `h.celsius` で設定できる。`h.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、`h.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)

3.2 Connecting sections

次に、soma に apical dendrite を付け加える。section は `connect` でつなぐが、*child*.`connect`(*parent*, *parent_x*, *child_x*) という表記の仕方をする。

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

import os
import array
from neuron import h

soma = h.Section()
ap_dend = h.Section()

soma.L = 30.0
soma.diam = 30.0
soma.nseg = 1
soma.insert('hh')
soma.el_hh = -30.0

ap_dend.L = 500.0
ap_dend.diam = 2
ap_dend.nseg = 23

ap_dend.connect(soma, 1.0, 0)

cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)

tstop = 200.0
dt = 0.01
v_init = -65.0

a = array.array('d',[0.0]*4)
fp = open('a.dat', 'wb')

h.finitialize(v_init)
```

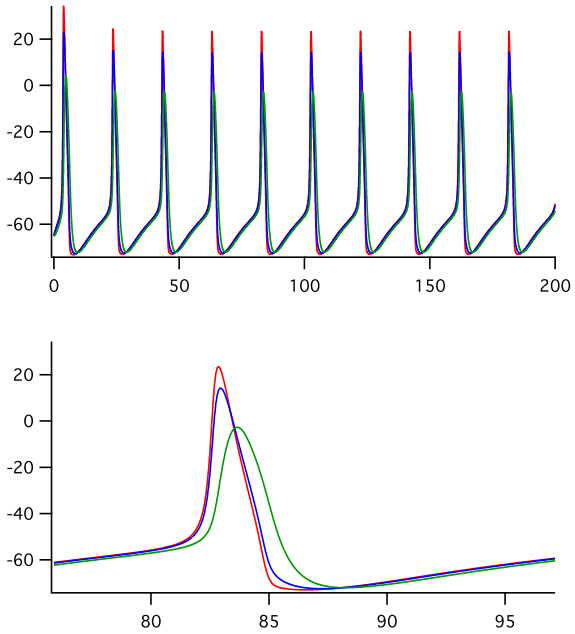


Fig.2　n08.hoc: Potentials of soma (red), proximal dendrite(0.1) (blue), and distal dendrite(0.9) (green). The lower panel is a magnification.

```
while h.t<tstop:
    cvode.solve()
    a[0] = h.t
    a[1] = soma(0.5).v
    a[2] = ap_dend(0.1).v
    a[3] = ap_dend(0.9).v
    fp.write(a)

fp.close()
os.system('gc a.dat 3 &')
#-----------------------------------------------
```

soma と ap_dend の 2 つの section より成り立っている。`nseg` で section 内の segment の数を指定する。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 apical dendrite に伝わっていることがわかる。

3.3　Point Process

3.3.1　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（`VClamp`）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 に加える場合には `insert()` ではなく、次の方法で行なう。

pp = h.*PointProcess*(*x*, sec= *section*)

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.py

import os
import array
from neuron import h

soma = h.Section()
ap_dend = h.Section()

soma.L = 30.0
soma.diam = 30.0
soma.nseg = 1
soma.insert('hh')

ap_dend.L = 500.0
ap_dend.diam = 2
ap_dend.nseg = 23
ap_dend.insert('hh')

ap_dend.gnabar_hh = 0.012
ap_dend.gkbar_hh = 0.0036
ap_dend.gl_hh = 0.00003

ap_dend.connect(soma, 1.0, 0)

# synaptic input

syn = h.AlphaSynapse(0.5, sec=ap_dend)
syn.onset = 5.0
syn.tau = 0.1
syn.gmax = 0.05

cvode = h.CVode()
```

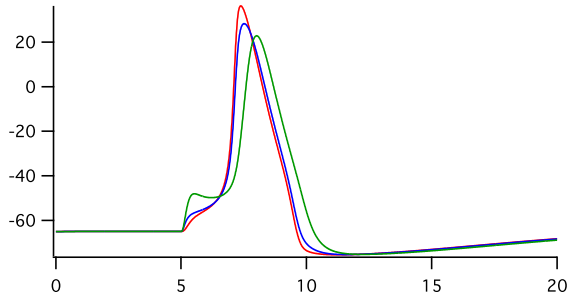


Fig.3　n09.hoc: Synaptically evoked action potential. Synaptic input at dendrite(0.5). Potentials at soma (red), proximal dendrite(0.1) (blue), and distal dendrite(0.9) (green)

```
cvode.active(1)
cvode.atol(1.0e-5)

tstop = 20.0
v_init = -65.0
a = array.array('d', [0.0]*4)
fp = open('a.dat', 'wb')

h.finitialize(v_init)
while h.t<tstop:
    cvode.solve()
    a[0] = h.t
    a[1] = soma(0.5).v
    a[2] = ap_dend(0.1).v
    a[3] = ap_dend(0.9).v
    fp.write(a)
fp.close()
os.system("gc a.dat 3 &")
#-----------------------------------------------
```

3.4　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、`class` *class_name* で行う。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。`self.` が前に付けられていることにより、クラス変数、クラス関数であることがわかる。Python では変数、関数は原則的に public である。

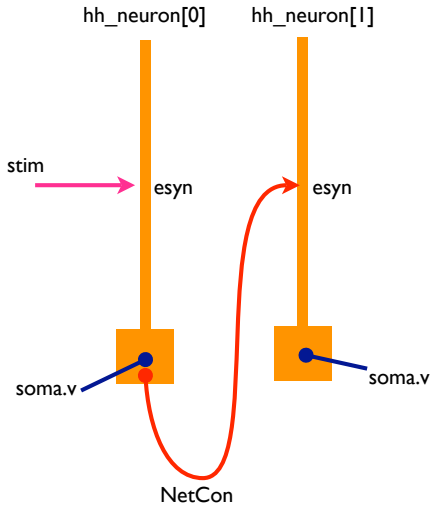


Fig.4　Two synaptically connected neurons.

3.5　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。

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

もしくは、

nc = h.NetCon(*section*._ref_v, *target_pp*, *threshould*, *delay*, *weight*)

という形で定義される。

NetCon の情報は list を用いて扱うと便利である。

nclist = []

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

もしくは、

nclist.append(h.NetCon(*section*._ref_v, *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.py
#--------------------------------------
```

```
import os
import array
from neuron import h

#    class definition
class HHneuron():
    def __init__(self):
        self.soma = h.Section()
        self.ap_dend = h.Section()

        self.soma.L = 30.0
        self.soma.diam = 30.0
        self.soma.nseg = 1
        self.soma.insert('hh')

        self.ap_dend.L = 500.0
        self.ap_dend.diam = 2.0
        self.ap_dend.nseg = 23
        self.ap_dend.insert('hh')
        self.ap_dend.gnabar_hh = 0.012
        self.ap_dend.gkbar_hh  = 0.0036
    self.ap_dend.gl_hh = 0.00003

    self.ap_dend.connect(self.soma, 1.0, 0)
    self.esyn = h.Exp2Syn(0.5, sec=self.ap_dend)
    self.esyn.tau1 = 0.5
    self.esyn.tau2 = 1.0
    self.esyn.e = 0
# end of class HHneuron

# cells
hh_neuron = [HHneuron() for i in range(2)]

# synapse
stim = h.NetStim(0.5)
stim.interval = 20.0
stim.number = 3
stim.start = 20.0
stim.noise = 0
```

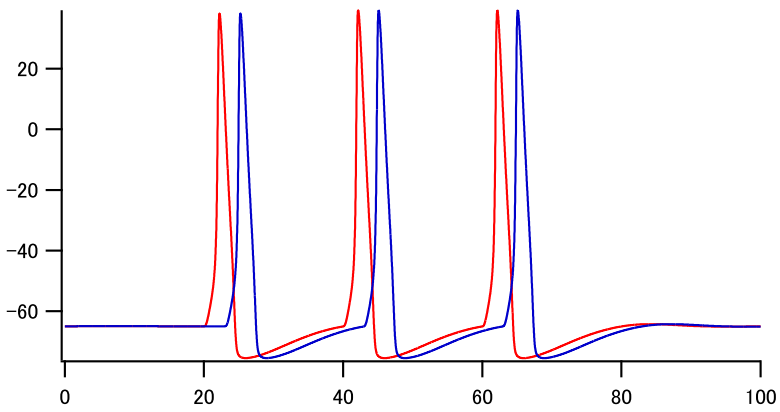


Fig.5　n10.hoc: Synaptically connected neurons

```
# connections
nclist = []
nclist.append(h.NetCon(stim, hh_neuron[0].esyn, 0.0, 0, 0.02))
nclist.append(h.NetCon(hh_neuron[0].soma(0.5)._ref_v, hh_neuron[1].esyn, 10, 1, 0.02))

tstop = 100
dt = 0.01
v_init = -65

cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)
a = array.array('d',[0.0]*3)
fp = open('a.dat', 'wb')

h.finitialize(v_init)
while h.t < tstop:
    cvode.solve(h.t + dt)
    a[0] = h.t
    a[1] = hh_neuron[0].soma(0.5).v
    a[2] = hh_neuron[1].soma(0.5).v
    fp.write(a)
fp.close()
os.system('gc a.dat 2 &')
#---------------------------------------
```

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 (図 3.5)。Rebound

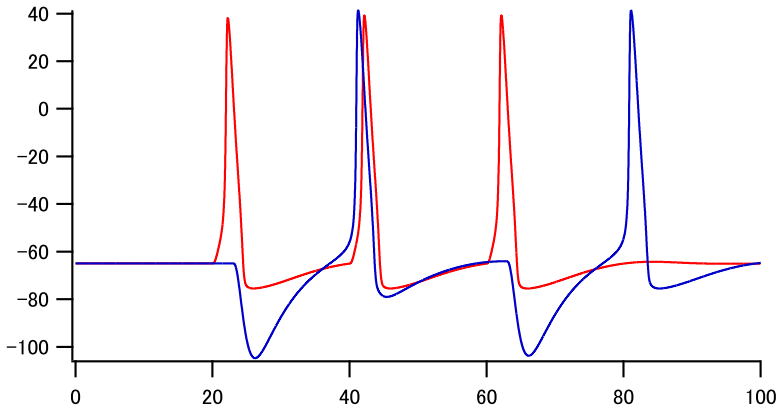


Fig.6　Inhibitory connection

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。

3.6　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.py
#
import os
import array
from neuron import h

soma = h.Section()
vcl = h.VClamp()

soma.L = 30.0
soma.diam = 30.0
soma.nseg = 1

soma.insert('hh')

esyn = h.Exp2Syn(0.5, sec=soma)
vcl = h.VClamp(0.5, sec=soma)
```

```
stim = h.NetStim(0.5)
stim.interval = 50.0
stim.number = 2
stim.start = 50.0
stim.noise = 0

nclist = []
nclist.append( h.NetCon(stim, esyn, 0.0, 0, 0.005))

tstop = 200.0
h.dt = 0.01
v_init = -65.0

vcl.dur[0] = 10.0
vcl.dur[1] = 10.0
vcl.dur[2] = 180.0
vcl.amp[0] = v_init
vcl.amp[1] = v_init
vcl.amp[2] = v_init
vcl.gain = 1000.0
vcl.tau1 = 0.01
vcl.tau2 = 0.01

a=array.array('d',[0.0]*3)
fp = open('a.dat', 'wb')

h.finitialize(v_init)
while h.t < tstop:
    h.fadvance()
    a[0] = h.t
    a[1] = soma(0.5).v
    a[2] = 1000.0 * vcl.i
    fp.write(a)
fp.close()
os.system("gc a.dat 2 &")
#---------------------------------------
```

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.py
```

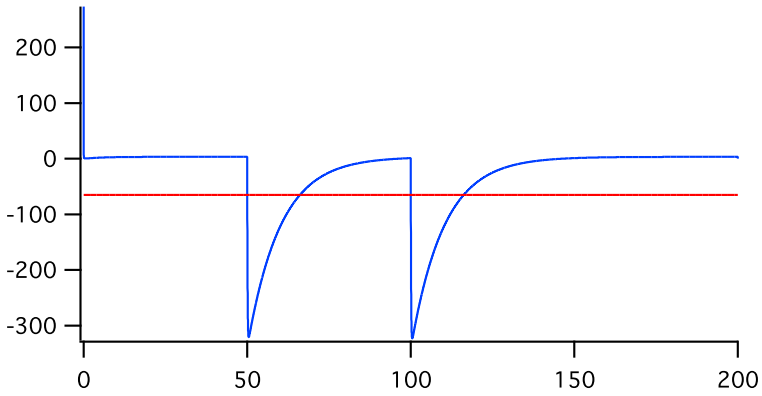


Fig.7 n11.hoc: Synaptic current measured in the voltage-clamp mode

```
# iv in voltage-clamp mode

import os
import array
from neuron import h

tstop = 50.0
h.dt = 0.01
vstart = -100.0
vstep = 10.0
ntrace = 20
npnt   = int(tstop/h.dt)

a = [ array.array('d', [0.0]*(ntrace+1)) for i in range(npnt)]

soma = h.Section()
soma.diam = 30.0
soma.L = 30.0
soma.nseg = 1
soma.insert('pas')

vc = h.VClamp(0.5, sec=soma)
vc.dur[0] = 10.0
vc.dur[1] = 10.0
vc.dur[2] = 30.0
```

```
syn = h.Exp2Syn(0.5, sec=soma)
syn.e = 0

ns = h.NetStim()
ns.number = 1
ns.start = 10.0
ns.noise = 0

nc = []
nc.append(h.NetCon(ns,syn, 0, 0, 0.001))   # 1 nS

for i in range(ntrace):
    v_init = vstart + vstep * i
    vc.amp[0]=v_init
    vc.amp[1]=v_init
    vc.amp[2]=v_init
    soma.e_pas = v_init
    h.finitialize(v_init)
    for j in range(npnt):
        h.fadvance()
        if i==0:
            a[j][0] = h.t
        a[j][i+1] = 1000 * vc.i    # nA -> pA

fp = open('a.dat', 'wb')
for j in range(npnt):
    fp.write(a[j])
fp.close()
os.system('gc a.dat 20 &')
```


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.py
#
import os
```

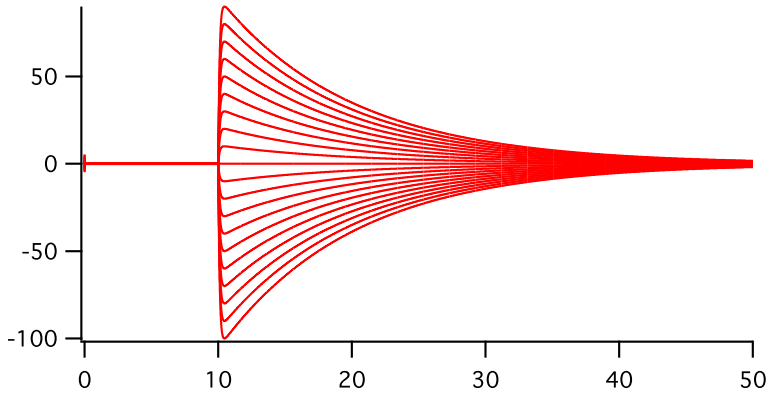


Fig.8 n12.hoc: A group of synaptic current traces in the voltage-clamp mode

```
import array
from neuron import h

soma = h.Section()
dend = [h.Section() for i in range(10)]

soma.L = 30.0
soma.Ra = 100.0
soma.nseg = 1
soma.diam = 30.0
soma.insert('hh')
vcl = h.VClamp(0.5, sec=soma)

for i in range(10):
    dend[i].L = 300.0
    dend[i].Ra = 100.0
    dend[i].nseg = 21
    dend[i].diam = 2.0
    dend[i].insert('hh')
    dend[i].gl_hh *= 2.3

esyn = h.Exp2Syn(0.75, sec=dend[0])
for i in range(10):
    dend[i].connect(soma, 1.0, 0.0)

# stimulation
```

```
stim = h.NetStim(0.5)
stim.interval = 50.0
stim.number = 2
stim.start = 50.0
stim.noise = 0

# synaptic connections
nclist = []
nclist.append( h.NetCon(stim, esyn, 0.0, 0, 0.001))

tstop = 200
h.dt = 0.01
v_init = -65

vcl.dur[0] = 10.0
vcl.dur[1] = 10.0
vcl.dur[2] = 180.0
vcl.amp[0] = v_init
vcl.amp[1] = v_init
vcl.amp[2] = v_init
vcl.gain = 1000.0
vcl.tau1 = 0.1
vcl.tau2 = 0.1

a = array.array('d', [0.0]*4)
fp = open('a.dat', 'wb')

h.finitialize(v_init)
while h.t<tstop:
    h.fadvance()
    a[0] = h.t
    a[1] = soma(0.5).v
    a[2] = dend[0](0.75).v
    a[3] = 100 * vcl.i
    fp.write(a)
fp.close()
os.system('gc a.dat &')
#--------------------------------------
```

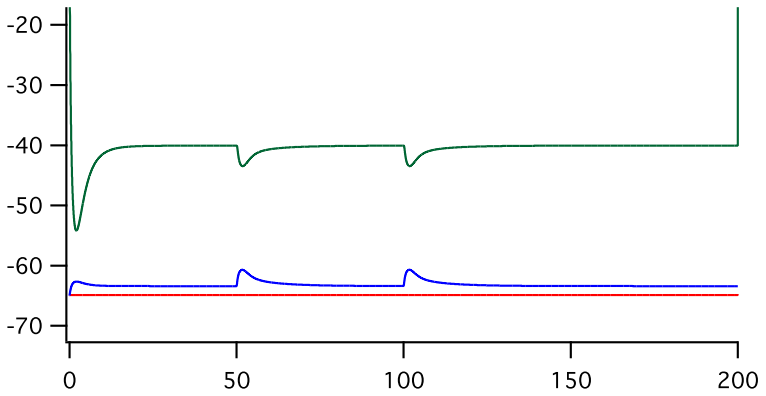


Fig.9 n13.hoc: Synaptic current measured in the voltage-clamp mode (green). Synaptic input at dendrite[0](0.75). The voltage-clamp electrode is positioned at soma. Potentials at soma (red) and dendrite[0](0.75) (blue). Currents are in nA, and magnified by a factor of 100.

4 Morphology

4.1 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、hoc ファイルであれば `dend01.diam(0:1) = 1.5:1.0` と書くことができたが、Python ではこの書き方はエラーとなってしまう。下の様な関数を作成して使う。

```
import numpy
from neuron import h
from itertools import izip

def taper_diam(sec, a, b):
    dx = 1.0/sec.nseg
    for (seg, x) in izip(sec, numpy.arange(dx/2, 1, dx)):
        seg.diam=(b-a)*x+zero_bound

# Test
dend = h.Section(name='dend')
```

```
dend.L = 100.0
dend.nseg = 5
taper_diam(dend, 2.0, 3.0)
for seg in dend:
    print seg.diam
```

*2

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。

4.2 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、

- `h.psection`(*section_name*) `#` parameters of a section

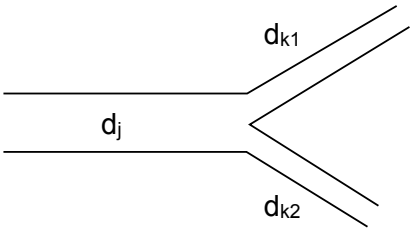


Fig.10 Branching of dendrites

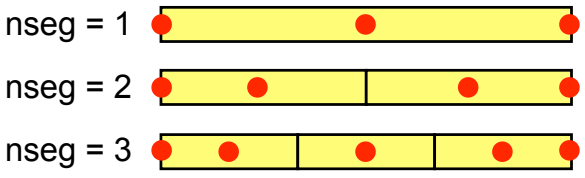


Fig.11 Calculation is performed at thenodes with red markings

*2 http://www.neuron.yale.edu/phpbb/viewtopic.php?f=2&t=2131

- h.psection を for sec in h.allsec():と組み合わせることにより、すべての section の情報を読み取れる。
- h.topology() # section connections

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 Section のインスタンスの配列として作り、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。Python で動かしているスクリプトでも、h.load_file() を使用することが出来る。この場合、各セクションは、名前の前に h. を付けることによりアクセスできる。

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

4.3 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、

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

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 に含まれており、関数名の前に h. を付けることによりアクセスできる。

```
//----------------------------------------
    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、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。

sh = `h.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。

なお、Graph が表示されている状態で、NEURON Main Menu →Window →Print File Window Manager →Print →PostScript という操作により、図を PostScript（ps）ファイルに保存することができる。以後の処理は、外枠などの余分な図形があるので、ps ファイルを import して GUI で操作編集できるソフトが便利。ps ファイルまたは eps（encapsulated postscript）ファイルを読み込むことができるソフトは比較的限られている。例えば、Adobe Illustrator（Windows, Mac OS）、Corel Draw（Windows）、Scribus[*3]（Linux, Mac OS, Windows; Open Source）など。eps ファイフは bitmap データを含んでおり、PostScript の解釈をして図を描くのではなく、この bitmap 情報を使用するソフト（Microsoft Office など）では、編集はできない。

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。

```
#------------------------------
#  n14.py
#
import os
import math
import array
import random
from neuron import h

h.load_file("nrngui.hoc")
h.load_file("stdlib.hoc")
h.load_file("tc200.geo")

nSyn=30
tstop = 200.0
dt = 0.01
v_init = -65.0

# Properties of dendrites
dendritic = []
```

[*3] http://www.scribus.net/

```
for sec in h.allsec():
    sec.insert('hh')
    sec.Ra = 100
#    h('printf("' + sec.name() + ': %f\\n", ' +  sec.name() + '.diam)')
    h('tmp = ' +  sec.name() + '.diam')
    if sec.name() !=  'soma' >= 0:
        dendritic.append(sec)

print 'number of dend =', len(dendritic)

for sec in dendritic:
    d = 1.0e5 * math.sqrt(sec.diam/(4.0*h.PI*100.0*sec.Ra*sec.cm))
    sec.nseg = int((sec.L/(0.1*d))+0.9)/2*2+1

# total dendritic length
total_length=0
for sec in dendritic:
    total_length += sec.L

print 'Tolal dendrite length = ', total_length

# Shape Plot
sh = h.Shape(1)
sh.size(-150, 150, -150, 150)

# stimulation
stim = h.NetStim(0.5)
stim.interval = 50.0
stim.number = 2
stim.start = 50.0
stim.noise = 0

nclist = []

locvec = [random.uniform(0, total_length) for i in range(nSyn)]
esyn = []
nclist = []
for k in range(nSyn):
    l = 0
    l1 = 0
```

```
    found = 0
    for sec in dendritic:
        if found==0:
            l1 += sec.L
            if l1 > locvec[k]:
                lx = (locvec[k]-l)/sec.L
                e = h.Exp2Syn(lx, sec=sec)
                esyn.append(e)
                sh.color(2, sec=sec)
                sh.point_mark(esyn[k],3)
                nclist.append( h.NetCon(stim, esyn[k], 0.0, 0, 0.001))
                found = 1
                break
            l = l1
sh.flush()
h.doEvents()

# main loop
fp = open('a.dat', 'wb')
a = array.array('d', [0.0]*2)

cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)

h.finitialize(v_init)
while h.t < tstop:
    cvode.solve(h.t + dt)
    a[0] = h.t
    a[1] = h.soma(0.5).v
    fp.write(a)

fp.close()
os.system('gc a.dat 1 &')
```

Morphology のデータファイルはそのまま。`h.load_file('tc200.geo')` で読み込むことが出来る。この hoc ファイルで、細胞の soma 等の Section が作られる。理由はわからないが、例えば h.dend1[0].diam というように section の diam(直径) にアクセスすると、500 という謝った値が帰ってくる。一度、hoc 環境で計算すると正しい値が帰ってくるようになる。Python を使用することにより乱数に関係する部分が簡素化されている。

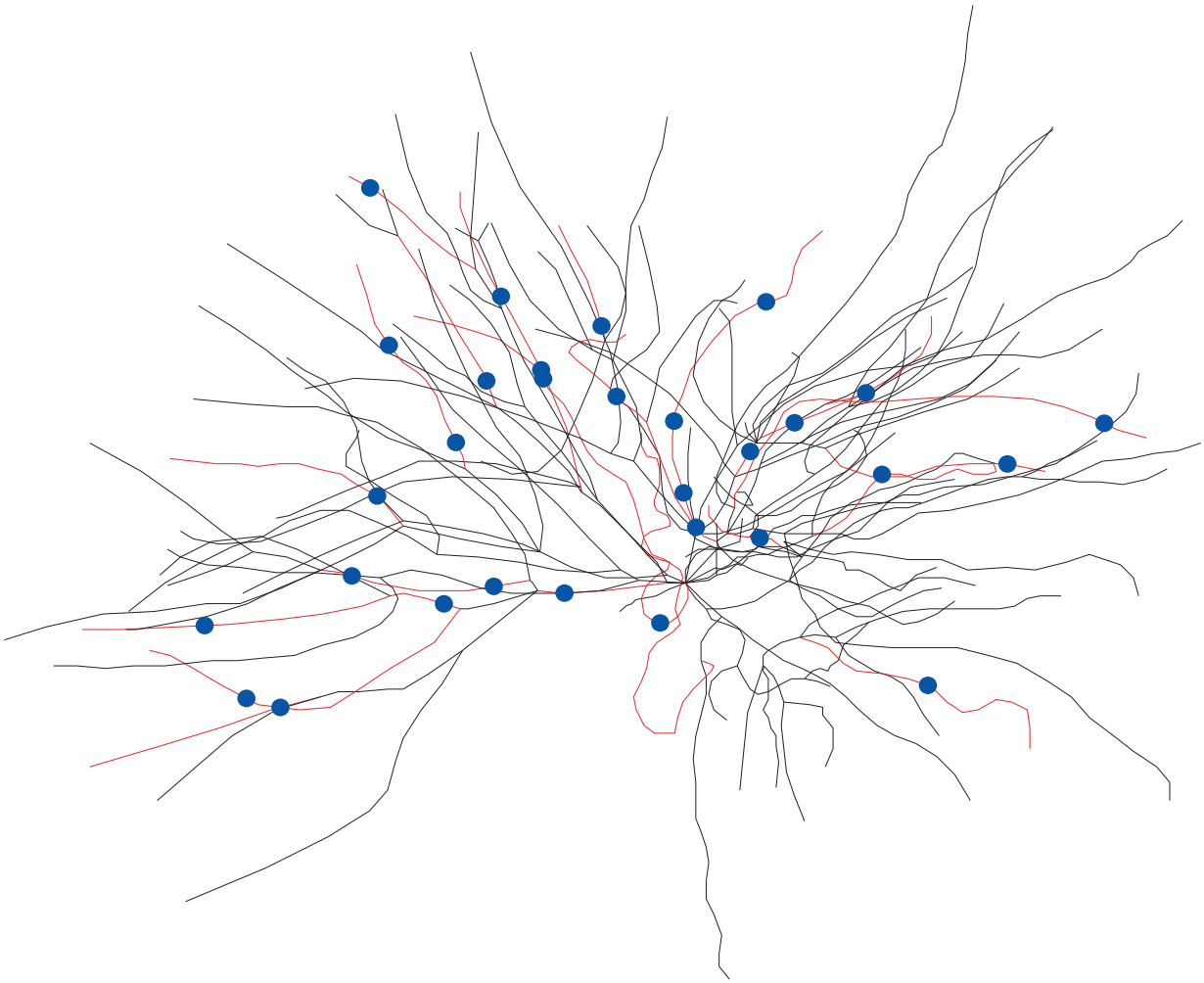


Fig.12　n14.py: tc neuron and random synapses

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。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。

```
#------------------------------
#  n15.hoc
import os
import math
import numpy
import random
from neuron import h

h.load_file('nrngui.hoc')
h.load_file('tc200.geo')
```

```
tstop = 200.0
dt = 0.01
v_init = -65.0

nSyn=100      # number of synapses
mdel = 10     # mean of synaptic delay
sdel = 1      # sd of synaptic delay
w = 0.001     # synaptic weight
br = 5

def measureDist(sec):
    dx = 0.0
    sec1 = sec
    while True:
        sr = h.SectionRef(sec=sec1)
        if sr.has_parent() < 1.0:
            break
        dx += sec1.L
        sec1 = h.SectionRef(sec=sec1).parent
    return dx

def countBranch(sec):
    bx = 0
    sec1 = sec
    while True:
        sr = h.SectionRef(sec=sec1)
        if sr.has_parent() < 1.0:
            break
        bx += 1
        print sec1.name()
        sec1 = h.SectionRef(sec=sec1).parent
    return bx

#-------------------------------
# Properties of dendrites

dendritic = []
for sec in h.allsec():
    sec.insert('hh')
```

```
    sec.Ra = 100
    if sec.name() != 'soma':
        dendritic.append(sec)

for sec in dendritic:
    d = h.lambda_f(100,sec=sec)
    sec.nseg = int((sec.L/(0.1*d))+0.9)/2*2+1

dendriticN = []
total_length=0
for sec in dendritic:
    if countBranch(sec) >= br:
        dendriticN.append(sec)
        total_length += sec.L

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

# Shape Plot
sh = h.Shape(1)

# stimulation
stim = h.NetStim(0.5)
stim.interval = 50.0
stim.number = 2
stim.start = 50.0
stim.noise = 0

# random generator
locvec = [random.uniform(0, total_length) for i in range(nSyn)]
delvec = [random.gauss(mdel, sdel) for i in range(nSyn)]
nc = []
esyn = []
for k in range(nSyn-1):
    l = 0
    l1 = 0
    found = 0
    for sec in dendriticN:
    if found == 0:
        l1 += sec.L
        if l1 > locvec[k]:
```

```
            lx = (locvec[k]-l)/sec.L
            e = h.Exp2Syn(lx, sec=sec)
            esyn.append(e)
            sh.color(2, sec=sec)
            sh.point_mark(esyn[k],3)
            nc.append(h.NetCon(stim, esyn[k], 0.0, delvec[k], w))
            found = 1
            break
        l = l1
sh.flush()
h.doEvents()

# main loop
a = numpy.zeros(2, float)
fp = open('a.dat', 'wb')
print 'Sim Start'

h.finitialize(v_init)
cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)

while h.t < tstop:
    cvode.solve(h.t+dt)
    a[0] = h.t
    a[1] = h.soma(0.5).v
    fp.write(a)

fp.close()
os.system('gc a.dat &')
```

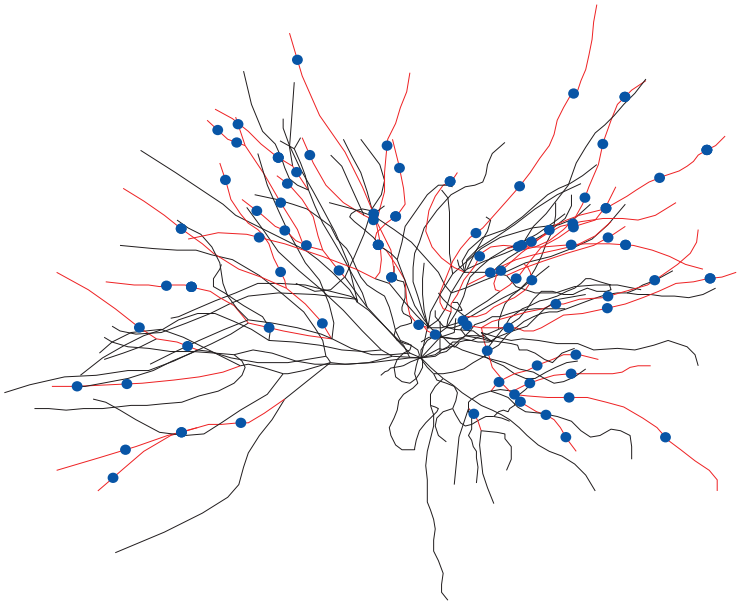


Fig.13　n15.hoc; 100 random synapses

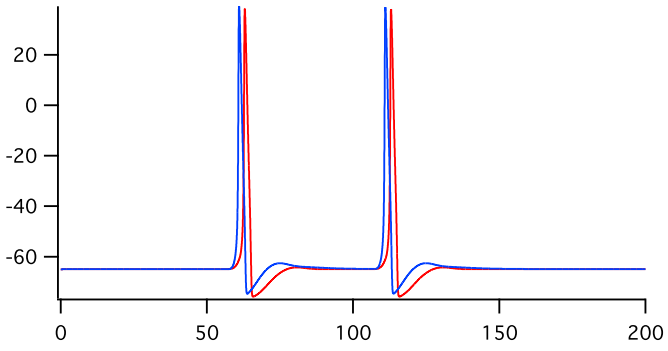


Fig.14　赤：n14.hoc のトレース、青：n15.hoc のトレース

5 Spikes

この Spikes のセクションの全体、Python ではまだ未確認。

5.1 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()
//------------------------------
```

5.2 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。

6 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。oc の代わりに Python を使用する場合も、全く同じ方法でよい。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, i686}/.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。

6.1 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。

6.1.1 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。

6.1.2 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。

6.1.3 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。

6.1.4 INITIAL block

初期化ブロック。関数 finitialize(v_init) から各 module の INITITAL block が使われる。

6.1.5 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（Crank-Nicolson 法)、runge（Runge-Kutta 法)、euler（Euler 法)、derivimplicit などが使用可能。これらの方法は、いずれも一定の dt を用いる fixed step method である。通常は cnexp。runge は求められる以上の精度のため使用されない（時間がかかる)。

状況によって積分の細かさを変化させる adaptive integrator の方法としては、CVODE 法が利用可能である。

6.1.6 DERIVATIVE block

ここには常微分方程式がくる。

6.1.7 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。

6.2 例：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、$z(0) = 0, z'(0) = 1$ とする。$z1 = z'$ 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 py ファイル。

```
#----------------------------------------
# m01.hoc:   test file for mo1.mod
import os
import array
from neuron import h

soma = h.Section()
soma.insert('m01')

tstop = 100.0
dt = 0.01
v_init = -65.0

cvode = h.CVode()
cvode.active(1)
cvode.atol(1.0e-5)

fp = open('a.dat','wb')
a = array.array('d',[0.0]*2)

h.finitialize(v_init)
while h.t < tstop:
    cvode.solve(h.t+dt)
    a[0] = h.t
```

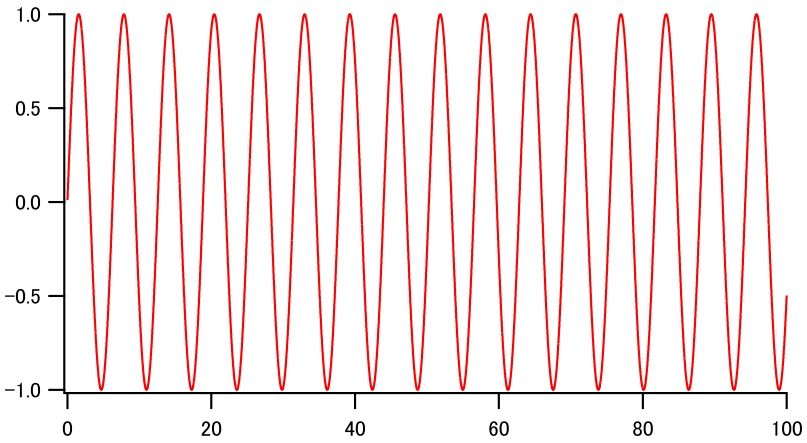


Fig.15　m01.hoc: m10.mod 内で cnexp を用いて数値計算。

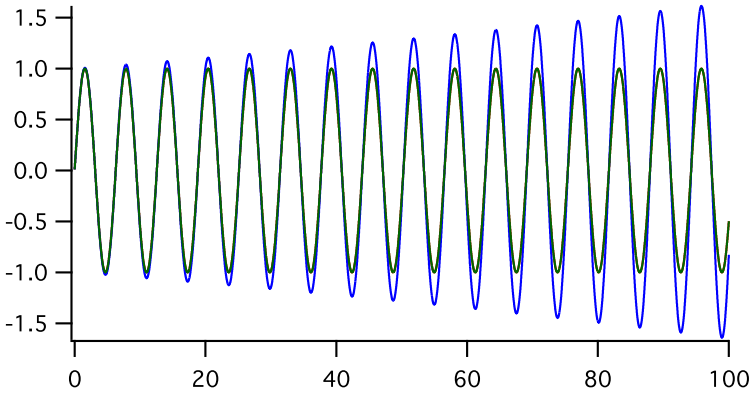


Fig.16　m01.hoc: m10.mod 内で異なる数値計算法を用いた。cnexp（赤; 緑に重なっている）、euler（青）、runge（緑）。

```
    a[1] = soma.z_m01
    fp.write(a)
fp.close()
os.system('gc a.dat 1 &')
#---------------------------------------
```

6.3　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。

6.4 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、

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

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

6.5 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。

6.6 NMDA receptor channel

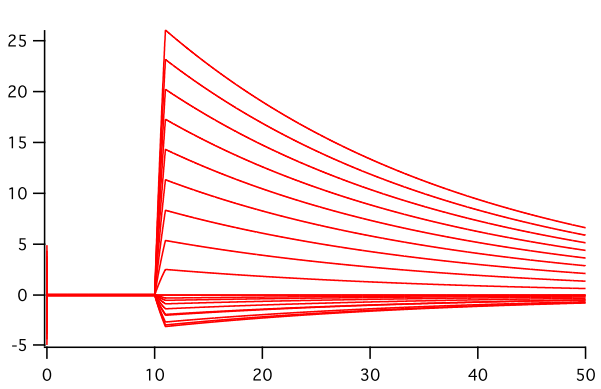
NMDA receptor channel は Mg^{2+} block により電位依存性と活動依存性を示す。下記の mod ファイルは、Gasparini et al, J Neurosci 24:11046-11056, 2004 による。state_discontinuity() を取除く等の改変を行なっている。
http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=44050

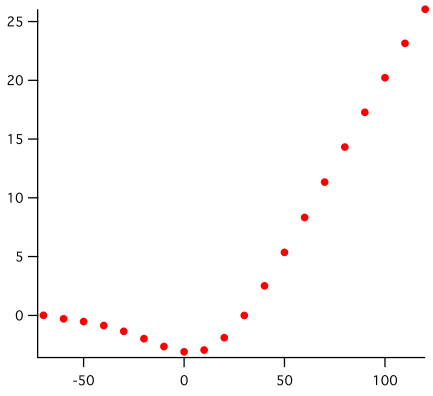
```
: ------------------------------------------------------------------------
NEURON {
    POINT_PROCESS nmdanet
    RANGE R, g, mg
    NONSPECIFIC_CURRENT i
    GLOBAL Cdur, Alpha, Beta, Erev, Rinf, Rtau
}
UNITS {
    (nA) = (nanoamp)
    (mV) = (millivolt)
    (umho) = (micromho)
    (mM) = (milli/liter)
}
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。





7 NEURON の内部

7.1 初期化

finitialize()

7.2 Integration の方法

7.3 Graph

付録 A　インストールの仕方

現在の最新バージョンは 7.2。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 で NEURON をまともに動かすのは容易ではない。それは NEURON が本来、UNIX/Linux の環境で開発され、開発に使用されたツール類が通常の Windows の開発ツールと互換性を持たないためである。実際 Windows で走る NEURON は、一般的に用いられている Microsoft Visual Studio C++ コンパイラで build したものではなく、Windows 上で UNIX/Linux に似た環境をつくる cygwin を用い gcc で build されている。

Windows で NEURON を Python を用いて動かすには、NEURON と Python が同一のコンパイラで build されていなくてはならない。上に記したように NEURON は cygwin 環境の gcc で build されており、他のコンパイラで build することは困難な様だ。幸い cygwin でも Python が動くので、現在の選択肢としては cygwin を用いるのが妥当の様だ。(問題点としては、numpy はライブラリとして用意されているが、scipy は部品をあつめて build しなくてはならない。)

cygwin 以外の環境としては、MinGW を用いるということも考えられる。Enthoughtpython の組み合わせがうまく動けば、かなり便利かもしれない。

NEURON の Windows 用バイナリパッケージには、python2.6 が含まれているが、これ自体では動かない (?)、またどのように拡張すればよいのかわからない、という問題点がある。

ここでは方針として、

1. cygwin のインストール
   setup.exe を使用する。
   (a) setup.exe をダウンロード、実行
   (b) Install from Internet
   (c) Root install directory は default のままで → C:\cygwin
   (d) Install for All Users
   (e) Local Package Directory → 例えば C:\cygwin\packages
   (f) Select Packages の画面が出てくる。default では Base システムがインストールされる。
   Base システムだけでは不十分で、gcc-core、gcc-g++、gcc-g77、make、libncurses、python、python-numpy, zip、unzip を選択する。依存性を自動的にチェックし、必要なものを追加してくれる。
2. NEURON のバイナリパッケージのインストール。インストールの場所をここでは C:/nrn72 としておく。
3. 新たにソースコードから NEURON を build。
   方法は、http://www.neuron.yale.edu/neuron/download/compile mswin に記載されている通り。

ソースコードのファイルをダウンロードして解凍。
フォルダの名前 iv-17 と nrn-7.2 は、それぞれ iv と nrn に変更しておく。
cygwin のターミナルで作業をすると、cygwin の python がある/usr/bin に PATH が通っているので、configure の時に cygwin の Python が認識されるはずです。確認するためは./configure で作成される Makefile を見ればよい。

```
$ cd iv
$ ./configure --prefix= `pwd `
$ make
$ make install
$ cd ../nrn
$ ./configure --prefix= `pwd ` --with-nrnpython
$ make
```

build されるファイルで必要なファイルは、次の 2 つのみ。これらをバイナリ版 NEURON のファイルと差し換える。

- nrniv.dll
- cygIVhines-3.dll

cygwin がインストールしてある環境では、NEURON を動かすために必要なファイルは比較的少なく、(多分) 次の通り。

- C:/nrn72/bin/nrniv.exe, nrniv.dll, cygIVhines-3.dll, mknrndll
- C:/nrn72/lib/nrn.def, mknrndl2.sh, mknrndll.mak, nrnunits.lib
- C:/nrn72/lib/python/neuron 以下のファイル ・ C:/nrn72/src 以下のファイル

バイナリ版の他のファイルはかえって邪魔をする可能性があるので、新しくフォルダをつくって必要なファイルだけで動かす方がよいかもしれない。(元のフォルダの名前を nrn72 から、例えば nrn72_binary に変更し、新しく nrn72 というフォルダを作成して、そこに必要なファイルをコピーする。)
NEURON を Python モードで動かすには、cygwin のターミナル (mintty.exe) から、

```
$ nrniv -python xx.py
```

とすればよい。
NEURON を Python の module として作成し、python そのものから NEURON を立ち上げることも可能なはずであるが、私自身はまだ成功していない。python setup.py install のところでエラーを起こしてしまう。

A.2 Mac OS の場合 (Snow Leopard)

現在の NEURON のバイナリ版は、すでに python 利用機能が組み込まれている。ターミナル (コマンドプロンプト) から nrniv を立ち上げるときに、`nrniv -python` とすれば python mode となっている。プロンプトが oc mode の時は `oc >`であるのに対し、python mode の場合は >>> となり、python モードであることがわかる。

この場合にどの python が使用されているかは、

```
>>> import sys
>>> sys.version
```

で知る事ができる。NEURON-7.2 の binary distribution の場合は、python 2.6 が使用されている。python 2.6 は、Snow leopard で MacOS に含まれているバージョンである。

Python の他のライブラリも使用することを考えると、NEURON そのものもソースコードからコンパイルして build するのがよい。この場合、iv については通常と同じ。nrn については、./configure の時に—with-nrnpython PYLIB='-framework Python' PYLIBLINK='-framework Python' を加える。複数の python がある場合に、どの python が使用されるかは、コマンドで python とした時に起動するものが使用される。なお MacOS では、PATH の設定を/etc/paths で行う事ができ、先に現れる方が優先度が高くなる。

余談であるが、MacOS ではいろいろなサービスやユティリティに python が用いられている。OS に付属の python(python 2.6) を動かなくすると、これらのユティリティプログラムの動作に支障を来たす (例えば Automator)。Linux でも同様の事が起きる。

ソースコードからの build の手順はつぎのとおり。

```
$ cd (source code directory)
$ IDIR = /Applications/NEUORN-7.2
$ cd iv-17
$ ./configure --prefix=$IDIR/iv
$ make
# make install
$ cd ..
$ cd nrn-7.2
$./configure --prefix=$IDIR/nrn --with-iv=$IDIR/iv --with-nrnpython \
    PYLIB='-framework Python' PYLIBLINK='-framework Python'
$ make
# make install
```

続いて、通常の方法で NEURON モジュールの作成を行う。

```
$ cd neuron/nrn-7.2/src/nrnpython
# python setup.py install
```

これで python を立ち上げ、`import neuron` とすると、 _environ が定義されていないとエラーになってしまう。

環境変数を得る関数 environ は、executable にはリンクされるが、dynamic lib のためには environ と称する関数が用意されていないため生じるエラーであり、MacOS のバグとのことである。

nrn-7.2/src/oc/system.c と nrn-7.2/src/oc/plot.c に、

```
#define environ (*_NSGetEnviron())
```

を加え、また nrn-7.2/src/ivoc/ivocmain.cpp に、

```
#include </usr/include/crt_externs.h>
```

```
#define environ (*_NSGetEnviron())
```
を加えた。コンパイルしてインストールしたところ問題は解決したようである。

A.3 Linux の場合

インストールの方法は、MacOS の場合とほぼ同じ。

付録 B References

- Hines ML, Davison AP, Muller E (2009) NEURON and Python. Front Neuroinfom 3:1. NEURON を python で使うために必読の解説。多少とも不規則な書き方も多く解説されている。
- 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.