A Simplifed Introduction to the NEURON Simulator

—NEURON を使い回す—

第 3 版 (Python 版)

Keiji Imoto
National Institute for Physiological Sciences (NIPS)
National Institutes of Natural Sciences (NINS)
Okazaki 444-8585, Japan

25-Dec-2017

25-Dec-2017 Update (Version 3)

13-Jan-2012: Python version (Version 2)

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

12-April-2010: minor changes

目次

第 I 部

NEURON の概要

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。これまでにおよそ 20 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、常微分方程式

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

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 には、その他の関連する機能も付け足されており、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。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。[*1] 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。第 2 版を作成するにあたっては、主言語を oc から Python に変更した。さらに今回第 3 版を作成するにあたっては、誤りを訂正するとともに、これまでに欠けていた部分、特にネットワークのシミュレーションを補った。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 3.6 を使用している[*2]。

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、Python 並びに数値計算についての基本的な知識があれば、更に

[*1] oc の歴史は古く、UNIX の開発で有名な Brian Kernighan が、Rob Pike との共著である “The Unix Programming Environment” (1984) で用いた hoc (High-Order Calculator) に由来している。その後、NEURON のために hoc はオブジェクト指向プログラミング (OOP) の要素を取り入れて oc となった。

[*2] この解説書では version の差が問題となることはない。Version 2 では `print` a と表すが、Version 3 では `print(`a`)` と表す。

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常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（Windows では 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。
4. 世界中の研究者に使用されて、多くの蓄積がある。これだけ多くの蓄積がある神経細胞・神経細胞ネットワークに関するシミュレータは他にない。

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、単純化したモデル神経細胞を使用する必要がある。

Python を使用する大きな利点は、シミュレーションで得られたデータを利用する際に、oc よりも Python の方が便利だからである。一方、NEURON での Python 使用はあくまでも oc のシステムをラップしたものであり、pure python ではない。またこれまでに蓄積されてきた NEURON の様々なスクリプトの多くは oc で書かれていることを考え合わせると、細胞形態、イオンメカニズムの部分は従来の oc のまま利用し、Python を利用する部分は、得られたシミュレーションデータの処理に限るようにするのがよいのかもしれない。

Python 部分も含めて、NEURON は何かとトラブルが起きやすく、それらを回避するには、トリック的なプログラミングが必要な場合が少なからずある。ただそのようなトラブルは世界中で経験されていることが多く、ネット検索により解決 (回避) 法を見つけられることが多い。

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）。

1.3 NEURON の学び方

NEURON に限らずプログラミングを学ぶには、実際自分で簡単なプログラムを走らせて、それを少しずつ自分の目的に合わせていくのが効果的な方法の様に思う。とにかくプログラムを走らせることが大切である。その上で、ソースコードを眺め、少しソースコードを変えてまた走らせてみる、ということを繰り返えすのがいちばんのトレーニングとなる。プログラムのソースコードにはいろいろな技（裏技も含めて）が込められており、それを初めから理解するのは難しく、少なくとも初めは、全てを理解しようとしないという態度も必要のようだ。

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、

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

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 では神経細胞を円柱体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　常微分法的式の計算方法

常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、Crank-Nicolson 法 (Nicholson ではなく Nicolson が正しい) が default の方法である。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 SUNDIALS (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。

2.3 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。Python は人工知能研究の分野で主要なプログラム言語となっている。Python の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。
- 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。
- 変数は宣言することなく使用することができる。
- 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。
- よく用いられるライブラリは、`numpy`、`scipy`、`matplotlib` などである。
- スクリプト言語の性質として、処理速度は遅い。この欠点を補うために、PyPy という Python コンパイラが開発されている。ただし使用範囲は限られている。
- C/C++ 言語との融合を図った cython は、python から派生した言語であり、高速化のために用いられる。


hoc ファイルを python に移植する場合、気をつけなくてはならないことがある。

- oc/hoc では、数値は基本的にすべて倍精度実数であるらしい。1/3 は、0.333 となる。しかし Python では、整数と実数の区別がなされており、1/3 は分子分母ともに整数と解釈され、計算結果を整数でだすので 0 となる。もちろん 1.0/3.0 であれば、0.333 となる。
- hoc プログラム内で作成された変数 a は、python プログラム内では、`h.a` でアクセスできるが、hoc の中では整数的に扱われていても、python 側では実数である。整数として用いるには変換が必要で、`int(h.a)` として用いる。

2.4 C/C++ 言語

mod ファイルは一旦 C ファイルに変換されて C/C++ コンパイラよりコンパイルされる。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。`NEURON` を使うにあたって、通常 C/C++ を必要とすることはほとんどない。

第 II 部

NEURON によるシミュレーション

3 NEURON による計算の流れ

NEURON では、1 つの神経細胞を多数のコンパートメントに区分し、それぞれのコンパートメント i の電位変化を、常微分方程式

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

で計算する。NEURON の内部では、多数の常微分方程式を設定し、選択された常微分方程式の計算方法（例えば、Crank-Nicolson 法や CVode 法）に従って数値計算を行う。計算量はかなりの量であるが、計算速度は結構速い。

神経細胞のコンパートメントには、2 段階がある。神経細胞は、まず Section に区分される。Section には、soma や短い dendrite、長い dendrite の一部分などがある。計算上、Section は、Segment に分けられる。dendrite や axon の場合、どの程度に区分すればよいかだいたいの目安が知られている（segment の長さの項、参照）。また Segment の数は奇数であることが望ましい。

NEURON のプログラム（スクリプト）は、多くの場合次のように構成されている。

1. 関係ファイル（環境設定ファイル、ライブラリファイルなど）のファイルの読み込み
2. 神経細胞の構造、性質の設定
3. 神経細胞間の結合、外部刺激、シナプス入力などの設定
4. 記録する変数の指定、実行に関係する数値の設定
5. === シミュレーション計算の実行===
6. 計算結果のプロット、ファイルへの出力など

シミュレーション時間 (tstop)、数値積分のステップサイズ (dt)、電位の初期値 (v_init)、細胞数などの数値の設定は、プログラムの始めの部分にまとめておくと、メンテナンスが容易になる。

4 Section

n01.py では、いよいよ 1 コンパートメント（section）のシミュレーションを行ってみる。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。

まず、Python 環境で NEURON を使うには、使用するライブラリを指定しなくてはならない。ほとんどが、neuron のなかの h であるが、何故か run() は neuron にある。その他には、数値計算用に numpy（通常 np の名前で用いる）、プロットのために matplotlib.pyplot（通常 plt の名前で用いる）が必要な場合が多い。

```
from neuron import h
import neuron
import numpy as np
import matplotlib.pyplot as plt
```

soma という Section クラスのインスタンスを作成する。h は、hoc で定義された関数であることを示している。default で作成される Section は、直径 diam が 500 μm、長さ L が 100 μm という値であるので、現実的な値（いずれも 30 μm）に設定する。そして、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 を挿入する。

```
soma = h.Section()
soma.diam = 30.0
soma.L = 30.0
soma.insert("hh")
```

微分方程式の数値計算には、CVode を用いる。誤差は、通常、絶対誤差で 10^{-5} にしておけば問題ない。

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

記録したいデータは、Vector クラスのインスタンスを作成しておき、その record() 関数で記録する変数を指定する。この関数は、シミュレーション時の記録のリストに変数を登録するらしい。Vector を *vec*、Section を *sec*、部位を *loc*、記録する変数を z としたとき、*vec*.record(sec(*loc*)._ref_*z*) と書く。

```
vv = h.Vector()
tv = h.Vector()
vv.record(soma(0.5)._ref_v)
tv.record(h._ref_t)
```

シミュレーションを走らせるためには、いろいろな部分の初期化が必要らしい。h.finitialize() と h.fcurrent() を入れておくのが無難。

シミュレーションの時間、電位の初期設定などをおこなって、シミュレーションを走らせる。

```
tstop = 200.0
v_init = -65.0
h.finitialize(v_init)
h.fcurrent()
neuron.run(tstop)
```


シミュレーションを走らせるには、関数 run() を用いるが、h.run() ではなく neuron.run() であることに注意。

シミュレーションが終わると、データは、Vector に蓄えられている。これをどのように表示するかについては、いろいろな方法がある（Appendix 参照）。ここでは、Python の matplotlib を用いている。as_numpy()

は、hoc の Vector を numpy の array に変換する関数。

```
ax = plt.subplot()
ax.plot(tv.as_numpy(), vv.as_numpy())
plt.show()
```

スクリプトファイル **n01.py** を走らせるには、ターミナルから、

```
> nrniv -python n01.py
```

もしくは

```
> nrniv n01.py
```

 何かの理由で（例えばある時間のところで条件を変えるなど）、1 ステップ毎に進めたい場合は、neuron.run(tstop) の代わりに、CVode を使っている場合であれば、

```
while h.t < tstop:
    cvode.solve()
```

もしくは、CVode 以外の ODE solver を使用している場合は、

```
while h.t < tstop:
    h.fadvance()
```

を用いる。また、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。一定のステップ毎に結果を知りたい場合には、neuron.run(tstop) の代わりに、

```
h.dt = 0.01  # for example
while h.t < tstop:
    cvode.fixed_step()
```

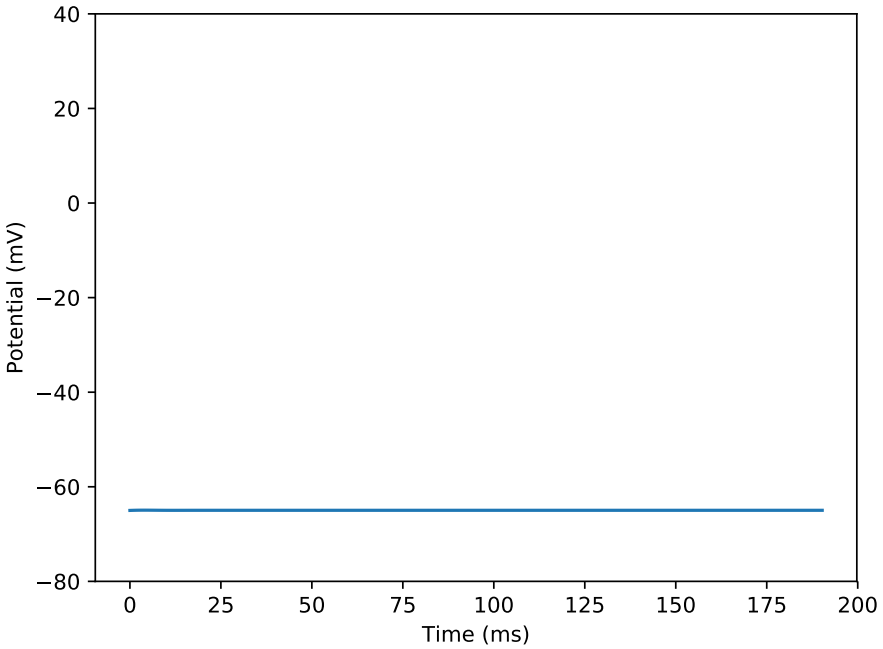


Fig 1　n01.py: HH neuron

n01.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 (Fig. 1)。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。**n02.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。

```
soma.el_hh = -30.0
```

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 (Fig. 2)。なお `hh` の詳細は `hh.mod` で定義されている。

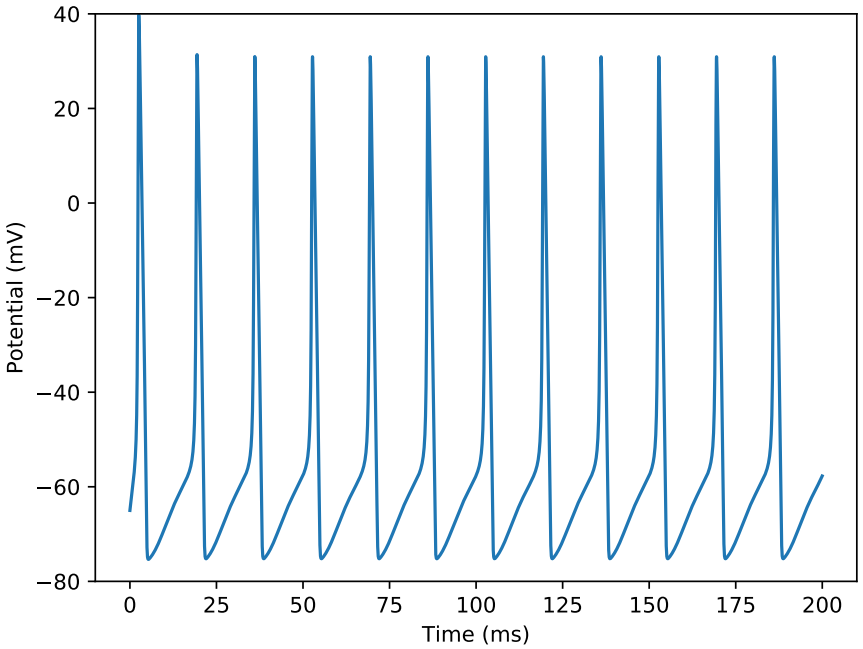


Fig 2　n02.py: HH neuron with a depolarized leak equilibrium potential

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。

神経細胞用には、`hh2.mod`[*3]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 `NEURON` シミュレータのシステムに組込む作業が必要なので（p. 28 参照）、当面は既に組込まれている `hh.mod` を用いることとする。

5　PointProcess

`Section` とならんで重要な要素は、`PointProcess` 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 `PointProcess` はそれぞれ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。x は、section での位置を示す。

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

[*3] http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=3817

n03.py では、current clamp の `PointProcess` である `IClamp` クラスを用いて電流注入を行なっている (Fig 3)。`IClamp` では、`delay`、`dur` (duration)、`amp` (amplitude) を設定する。ここでは、`amp` = 0.2 nA (200 pA) で、現実の実験で用いられる値に近いものである（グラフでは 100 倍の値を表示)。

hoc では、`delay` ではなく `del`。

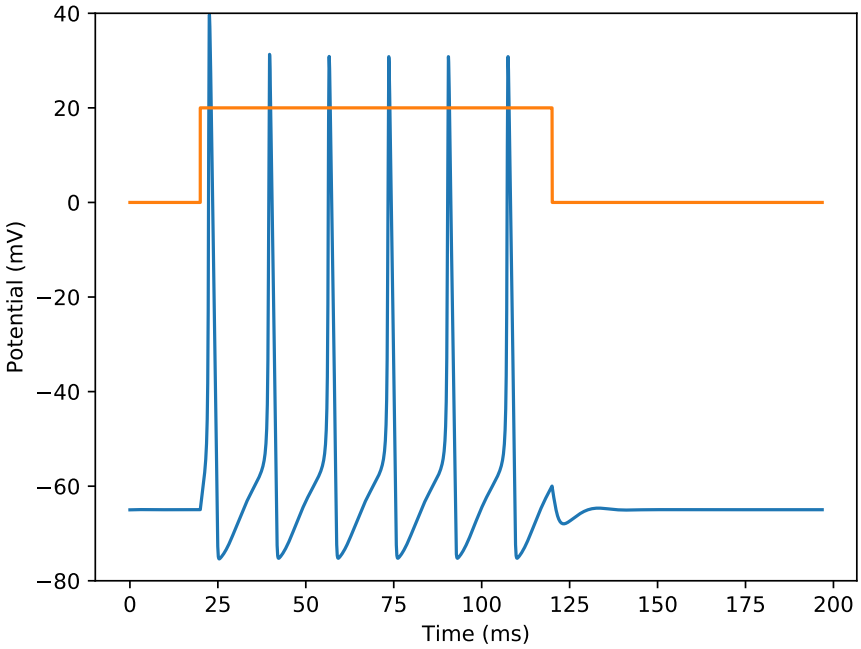


Fig 3　n03.py: Action potentials (blue) evoked by current injection (red, 0.2 nA, scaled 100 times).

6　Connecting sections

次に **n04.py** では、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。**n04.py** では、dendrite に soma の 1/10 の density で `hh` を加えた。

soma に current injection (0.5 nA、10 倍に拡大して表示）を行なって action potential を発生させ、soma および ap_dend のら 0.1（近位部）と 0.9（遠位部）の部分の電位変化をプロットした (Fig 4)。活動電位が soma から apical dendrite に伝わっていることがわかる。

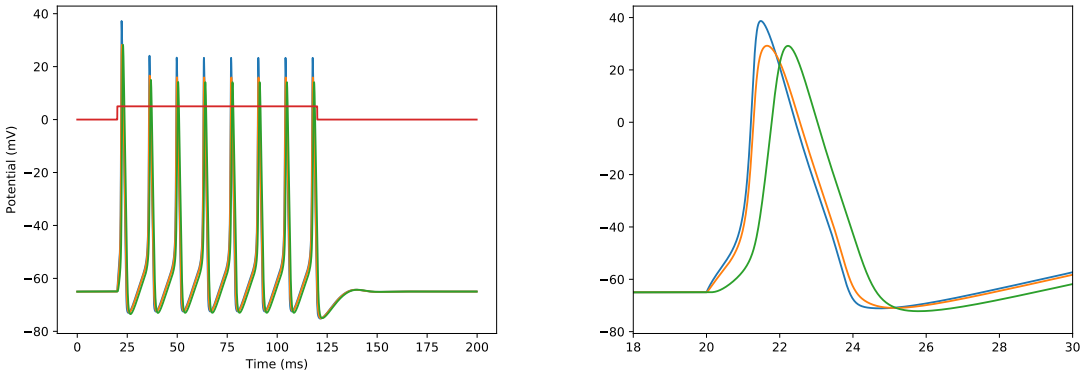


Fig 4　n04.py: Potentials of soma (blue), proximal dendrite(0.1) (red), and distal dendrite (0.9) (green). The right panel is a magnification.

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。この値は奇数であることが好ましい (p. 24 参照)。

`h.allsec()` は、細胞のすべての `section` を返す関数であり、すべての `section` の section variable を設定する場合などに使うことができる。例えば、内部抵抗 `Ra` を設定するには、

```
for sec in h.allsec():
    sec.Ra = 100.0
```

またこの応用として、すべての `segment` の range variable を設定することができる。range variable とは、`segment` によって異なりうる値であり、`diam` (deameter)、`v` (membrane potential)、`g` (specific conductance) などが含まれる。

```
for sec in h.allsec():
    for seg in sec:
        seg.pas.g = 0.0002
```

7　Synaptic input

Synapse 入力には、`PointProcess` のなかの `AlphaSynapse` クラスや `ExpSyn` クラスが用いられる。`AlphaSynapse` クラスはコンダクタンスを変化させる場合に、`ExpSyn` クラスは電流量を変化させる場合に用いる。

n05.py では `AlphaSynapse` クラスを用いて、soma に興奮性シナプス入力を入れている (Fig 5)。この場合は、soma や proximal dendrite だけでなく、distal dendrite でも action potential を発生している (Fig 6)。HH の sodium channel の density を下げていくに従って、dendrite の action potential が変化する様子をみることができる。

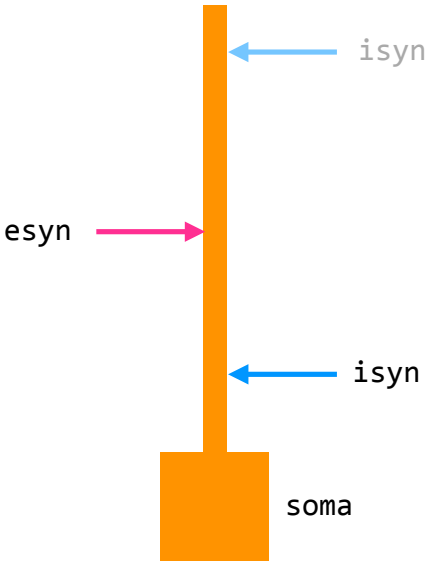


Fig 5　Location of excitatory and inhibitory inputs

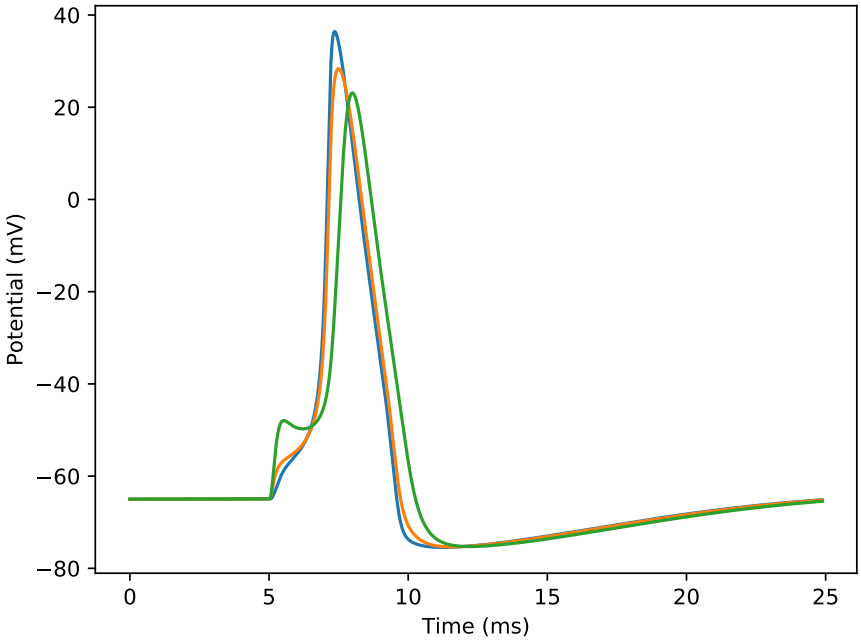


Fig 6　n05.py: Synaptically evoked action potential. Synaptic input at dendrite(0.5). Potentials at soma (blue), proximal dendrite(0.1) (red), and distal dendrite(0.9) (green)

次に **n06.py** では、興奮性シナプス入力に先立って抑制性シナプス入力を加える。抑制性シナプス電流の平衡電位を-70 mV に設定している。従って抑制性シナプス入力により大きく過分極することはないが、抑制

性シナプス入力の位置が興奮性入力の位置より soma よりである場合、soma での脱分極を抑える (shunting) (Fig 7)。

`matplotlib` を用いて、一つのグラフに複数のトレースをプロットする場合、x 軸の値は共通でなくてはならない。このため、`CVode` の `fixed_step()` 関数を用いている。

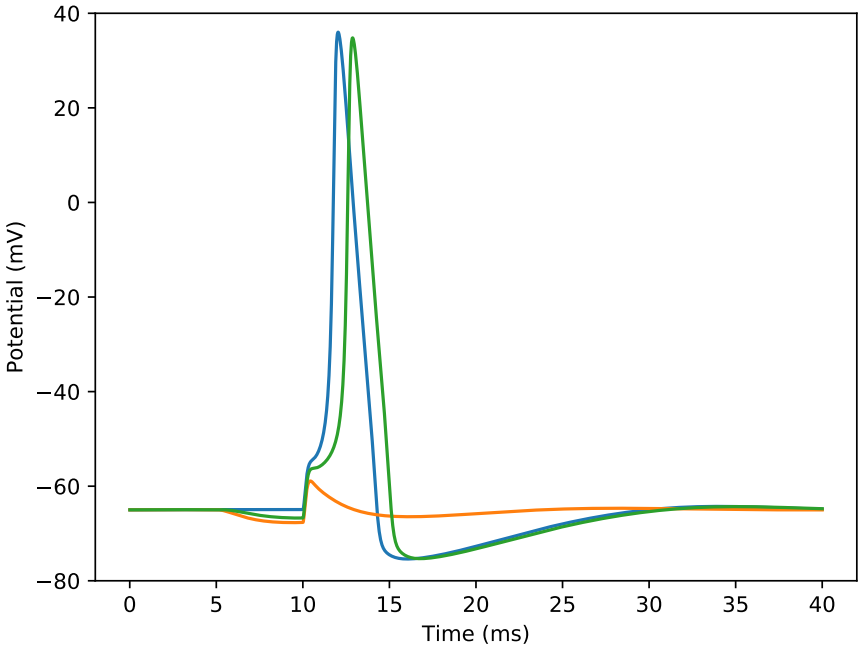


Fig 7　n06.py: Potentials in soma. Excitatory input (loc=0.5) only (blue). Additional inhibitory input at proximal dendrite (0.3) (red) or at distal dendrite (0.9) (green).

n07.py では、よりシステマティックに抑制性シナプス入力の位置を変化させてみた。（より正確に表現するなら、10 個のシナプスを作り、そのうち一つだけ、コンダクタンスの値をゼロ以外にしている。）(Fig 8)

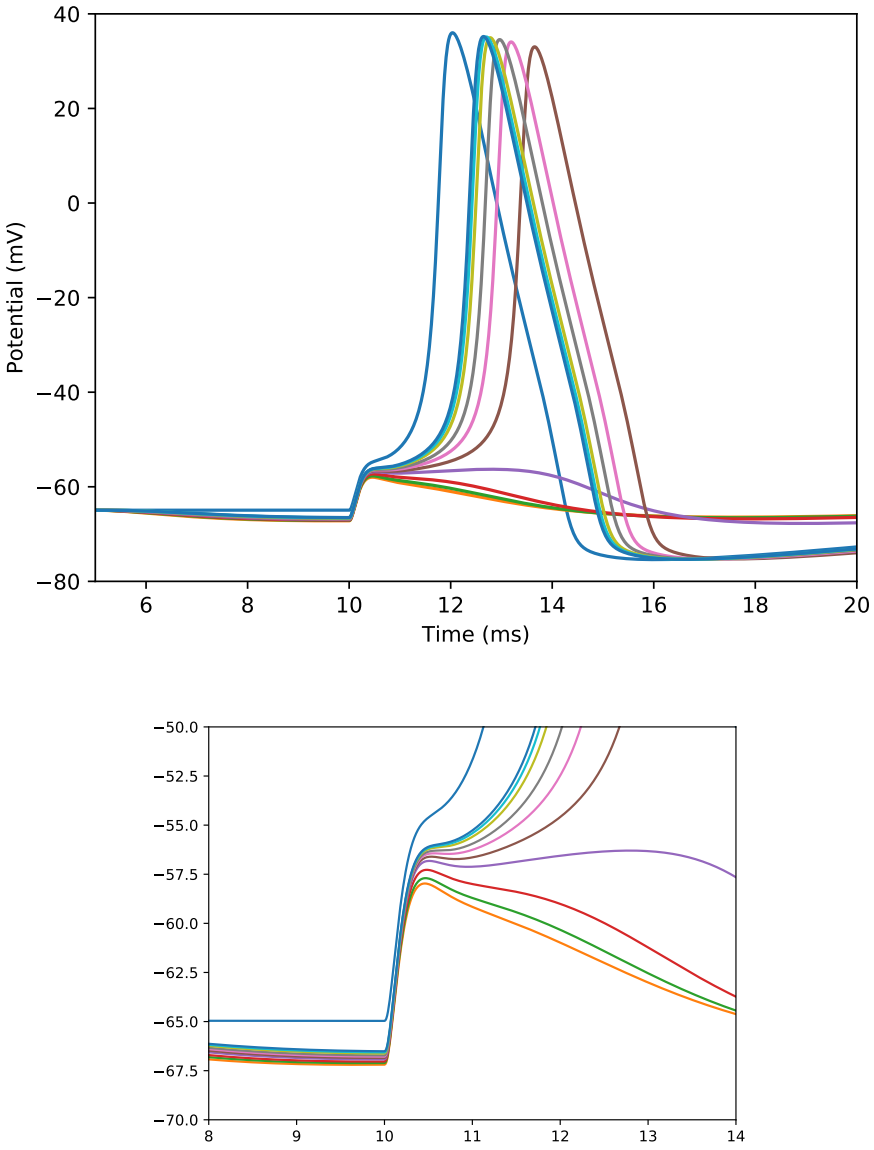


Fig 8 n07.py: Location effect of inhibitory synaptic input

8 クラス class

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。class（テンプレート；鋳型）を定義し、Class 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。この手法は、Object-oriented programming (OOP) と呼ばれ、同様の機能は多くのプログラミング言語で用いられている。

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（たとえ中身が空でも）。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。

クラスが別の Python ファイルで定義されている場合は、スクリプトの先頭で、

```
import class_file
```

として取り込む。extension の.py は不要。hoc ファイルの場合は、

```
h.load_file("class_file.hoc")
```

とする。


Python の使用が増加してきているとは言え、NEURON のモデルは、hoc ファイルの場合がほとんどである。神経細胞の特性の定義を hoc ファイルで行い、それを Python スクリプトに読み込むという方針が効率的である。

9 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*, *threshold*, *delay*, *weight*)

もしくは、

nc = h.NetCon(*section*._ref_v, *target_pp*, *threshold*, *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。

n08.py では、HHneuron クラスを定義し、2 つの HHneuron インスタンスを作成した。そして興奮性シナプス結合を加え、外部より刺激した（Fig. 9）。

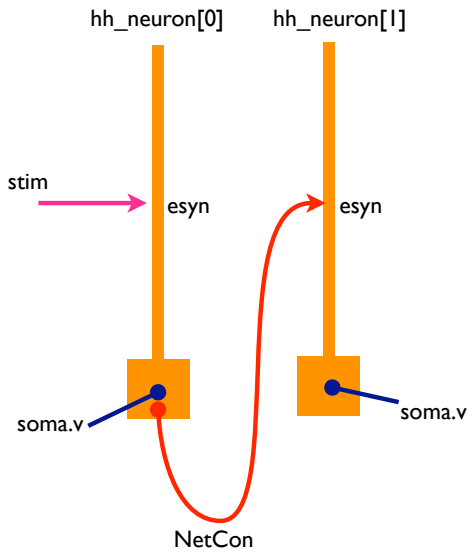


Fig 9　Two synaptically connected neurons.

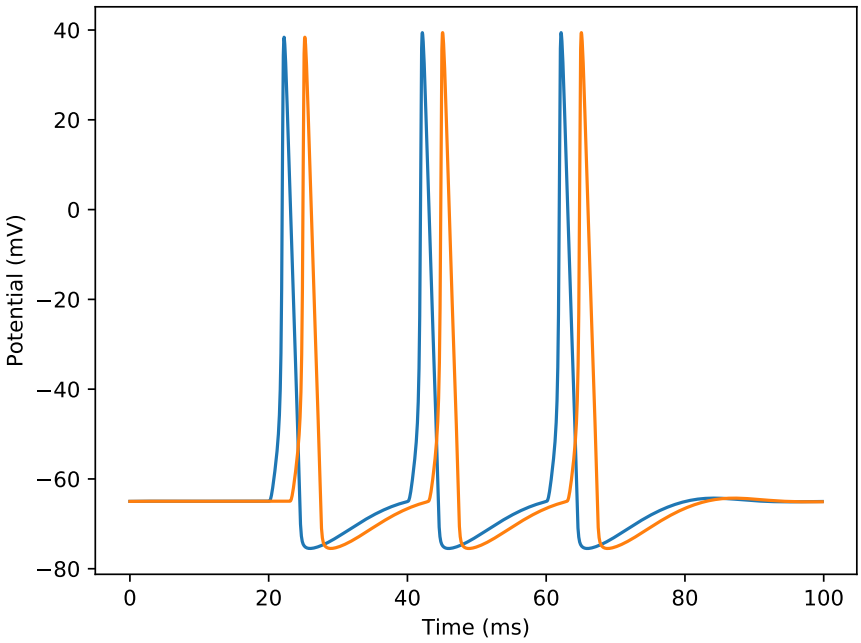


Fig 10　n08.py: Synaptically connected neurons

10 Voltage clamp

n09.py は、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。

Voltage clamp の場合、数値積分の方法として `CVode` は用いられず、default では Cranck-Nicolson 法が用いられる。そのためステップサイズ `h.dt` の値を設定しておく必要がある。`NEURON` が扱うシミュレーションでは、ステップサイズは 10 μs から 100 μ ぐらいであることが多い。

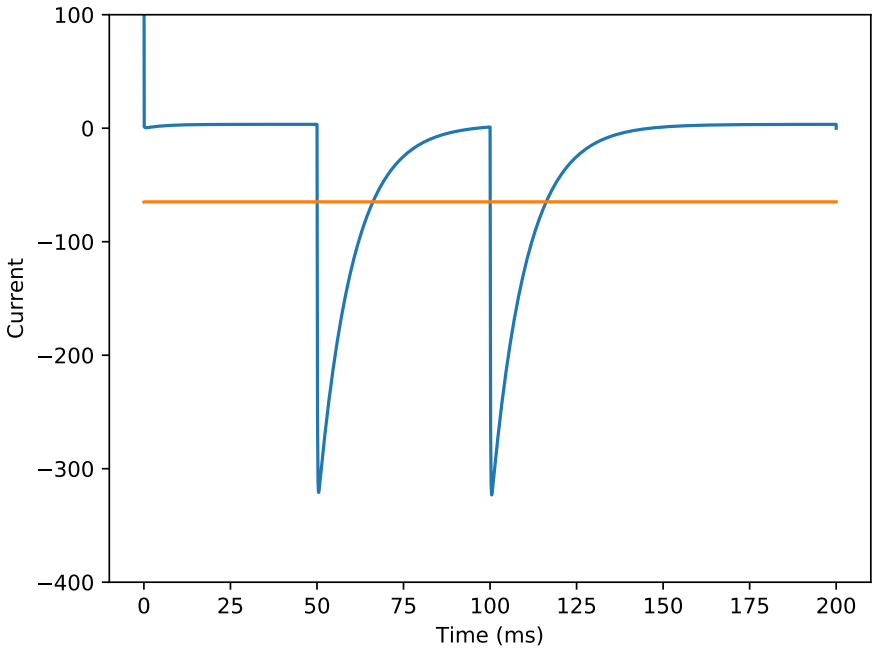


Fig 11 n09.py: Synaptic current (blue) measured in the voltage-clamp mode. Voltage(red) is clamped constant.

n10.py では、Voltage clamp でシナプス電流の i-v relation を求めている。基本的には上記の **n09.py** 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。

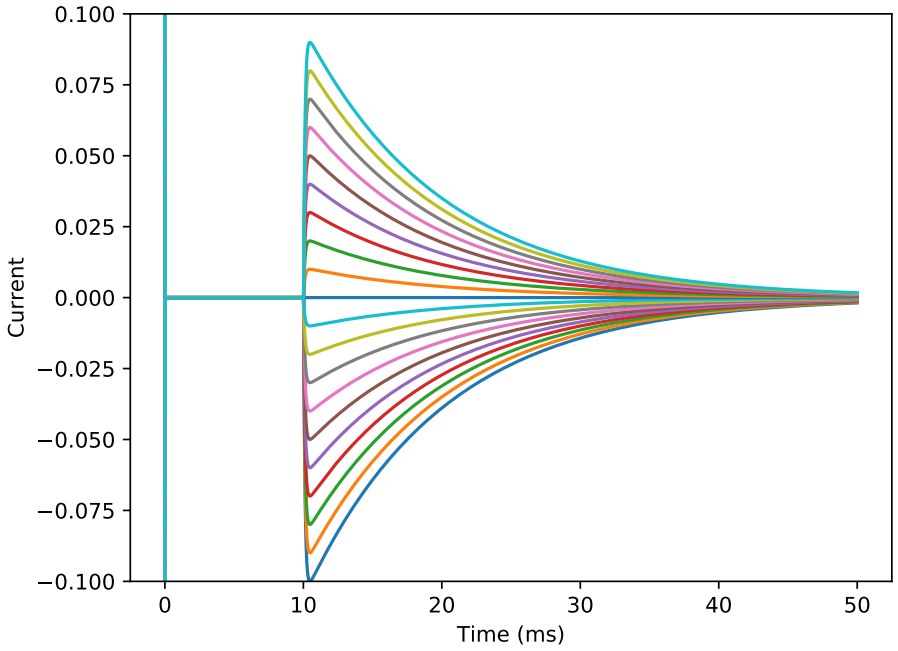


Fig 12　n10.py: A group of synaptic current traces in the voltage-clamp mode


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。

`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。`dendrite` の leak コンダクタンスを `soma` の 2.3 倍に設定している。

`soma` では、電位は一定であるが、シナプス入力があった `dendrite` の distal 部分（$x = 0.75$）では、数 mV の電位の変化が見られる。

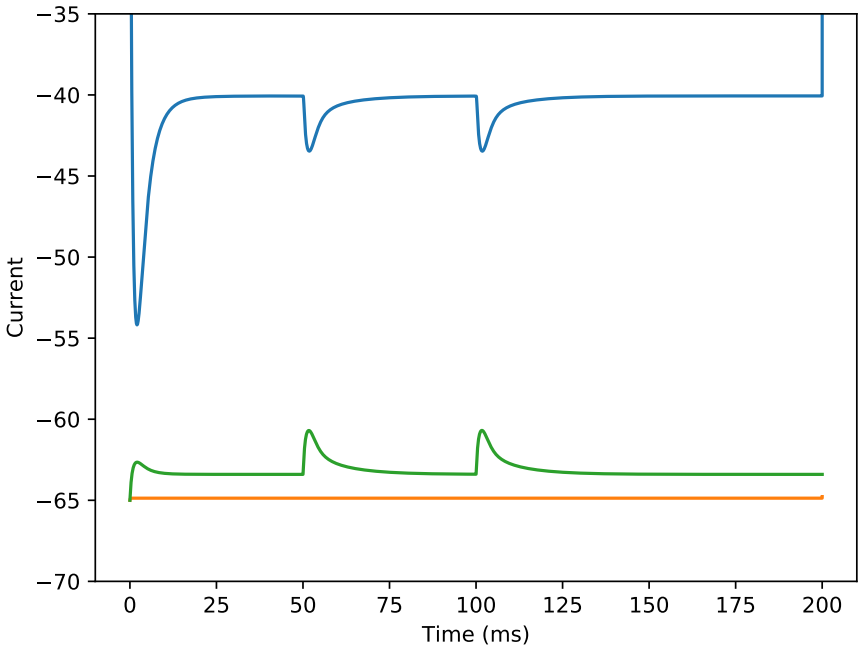


Fig 13 n11.py: Synaptic current measured in the voltage-clamp mode (green). Synaptic input at dendrite[0](0.75). The voltage-clamp electrode is positioned at soma. Potentials at soma (soma) and dendrite[0](0.75) (red). Currents are in nA, and magnified by a factor of 100.

第 III 部
NEURON を複雑なかたちの神経細胞に用いる

11 Morphology

11.1 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。[*4]

```
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)):
```

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

```
            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
```

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。

[b]

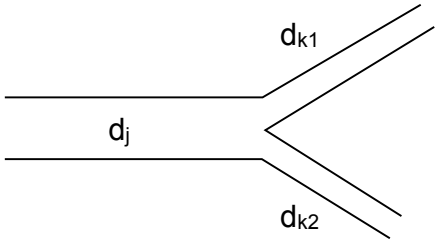


Fig 14　Branching of dendrites

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。

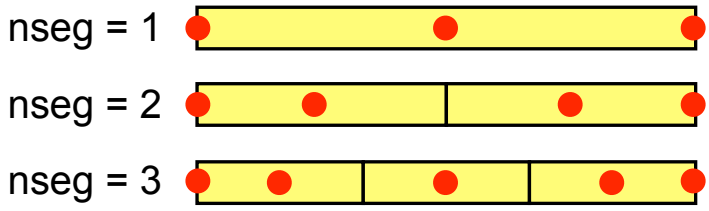


Fig 15　Calculation is performed at thenodes with red markings

11.2　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、

child.connect (*parent*, *parent_x*, *child_x*)

を用いる。

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
- `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")
```

11.3 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。

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。

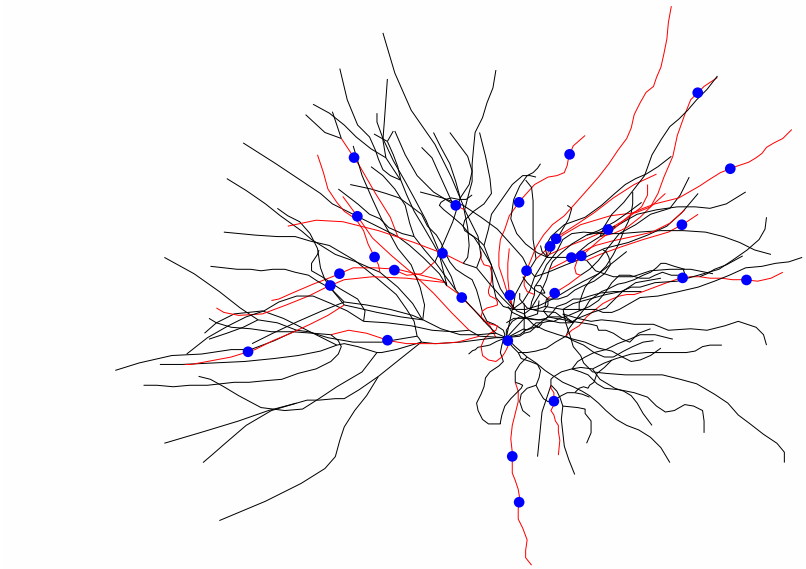


Fig 16　n12.py: tc neuron and random synapses

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。

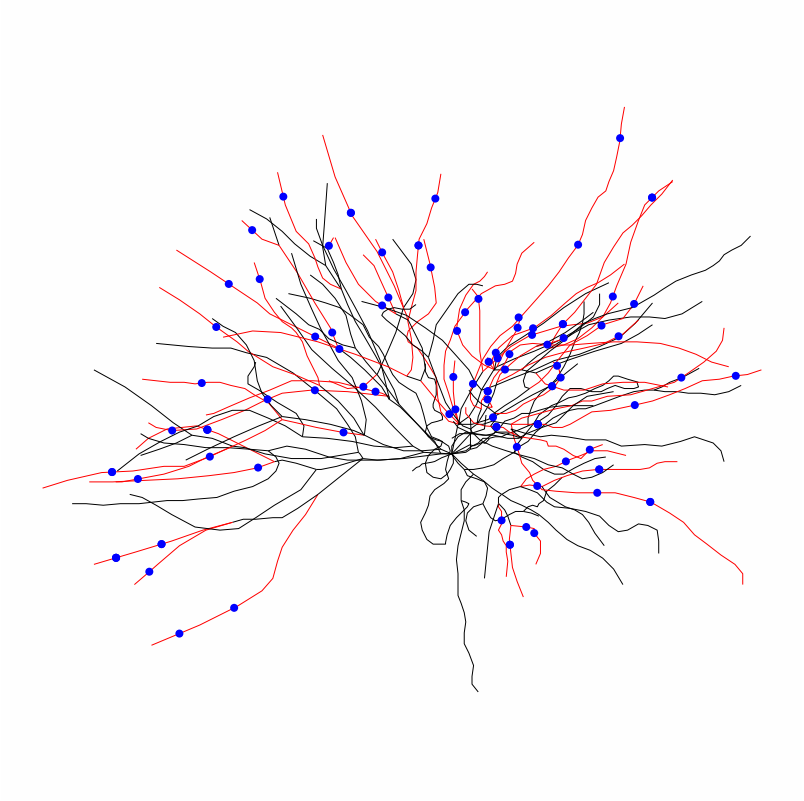


Fig 17　n13.py; 100 random synapses

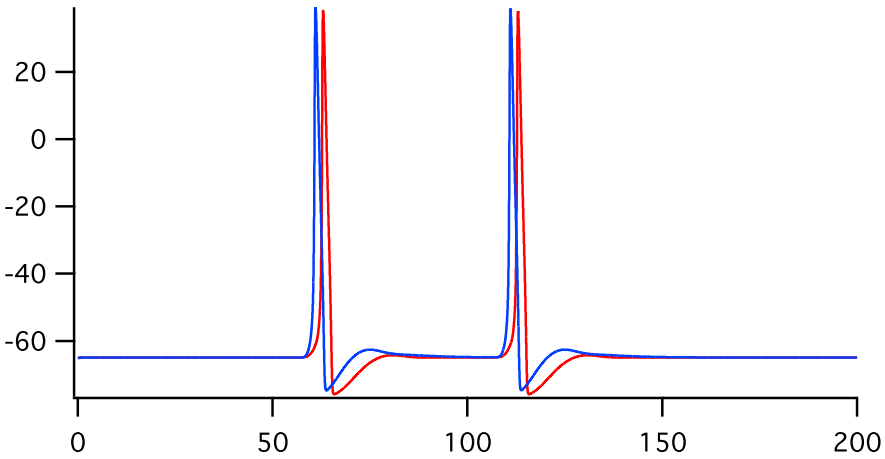


Fig 18　赤：n12.py のトレース、青：n13.py のトレース。比較のために時間軸を合わせている。

第 IV 部

NEURON のカスタマイズ

12　NMODL と mod ファイル

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る。その後 gcc（C コンパイラ）によりコンパイルされ、ライブラリが作成される (MacOS の場合は libnrnmech.0.so)。nrniv もしくは nrngui を起動するときに、このライブラリは自動的に読み込まれる。

12.1　Windows の場合

C:\Users\xyz\nrn\bin にある mknrndll という shell script を使用する。ただしこの script では NEURONHOME を示す変数 N が定義されていないので、

```
N=/c/Users/xyz/nrn
export N
```

の 2 行を、2 行目以降に付け加える。1 行目の#!sh は、1 行目でなくてはならない。

1. コマンドプロンプトを立ち上げる
2. sh で shell を起動
3. 現在の位置は、pwd で確かめることができる
4. mod ファイルがあるディレクトリに移動
5. mknrndll
6. nrnmech.dll が作成される。この dll は NEURON のプログラム起動時に自動的に使われる。

12.2　MacOS の場合

MacOS 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。もしくは、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、

```
nrnivmodl
```

もしくは、mod ファイルがあるディレクトリを *mod_directory* とすると、

`nrnivmodl` *mod_direcotry*

とする。

この操作により、ディレクトリの中の x86_64 ディレクトリに、special という名前のスクリプトファイル

とライブラリファイル libnrnmech.so（.libs の中）が作成される。

nrniv が実行されると、自動的に special を実行し、ライブラリが組み込まれる。

13 mod ファイルの 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。

13.0.1 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。

13.0.2 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。

13.0.3 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。

13.0.4 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。

13.0.5 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。

13.0.6 DERIVATIVE block

ここには常微分方程式がくる。

13.0.7 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。

14 A simple ordinary differential equation (ODE)

m01.mod では、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。

n14.py は、この mod ファイルを試すための py ファイル。

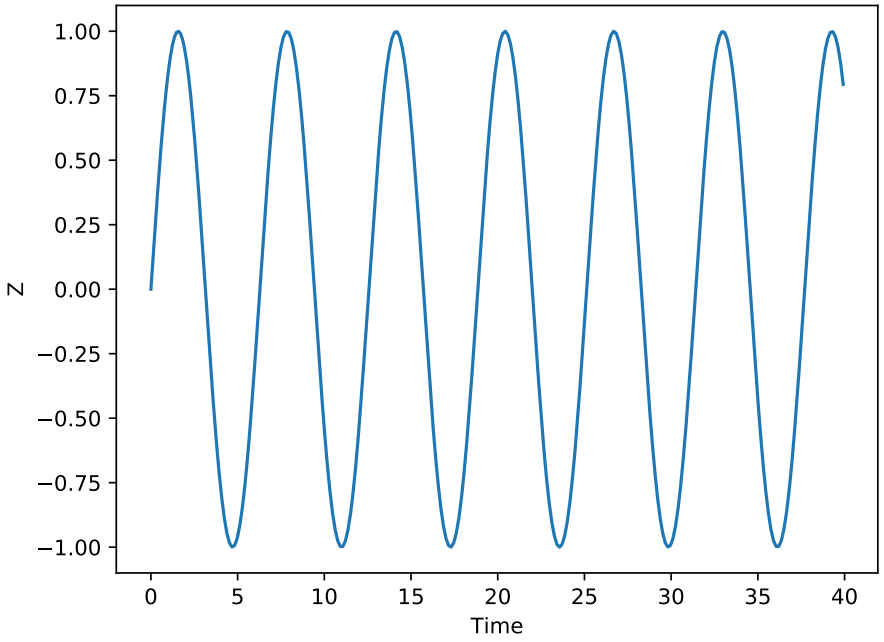


Fig 19　n14.py: Solve a simple ODE.

15　Markov 過程

ODE を計算できるなら、Markov process として表すことのできるイオンチャネルの gating を計算できる。電位依存性 Na^+ チャネルには、閉じた状態、開いた状態、不活性化した状態があることが Hodgkin-Huxley の時代より知られているが、それ以降の研究で、複数の閉じた状態と複数の不活性化した状態があると考えられるようになっている。状態間の rate constant は、あるものは定数、あるものは電位依存的であり、実験的にこれらの値をもとめることにより、チャネルの gating を Markov process として数式化することがなされて来た。

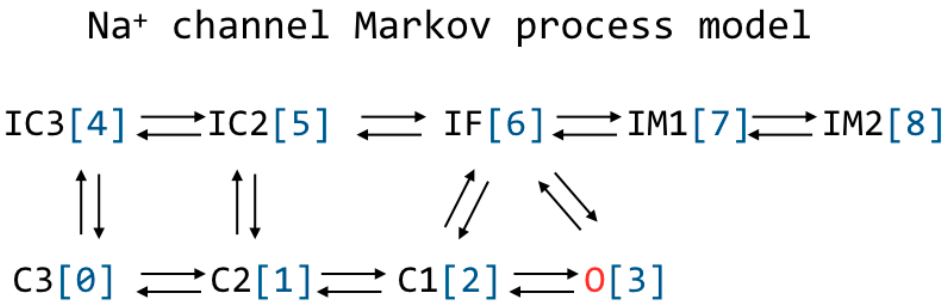


Fig 20　電位依存性 Na^+ チャネルの Luo-Rudy モデル.

mna.mod では、心筋 Na^+ チャネルの Luo-Rudy モデルを、NMODL で表した。9 つの states を想定し、state[3] が open state である。プログラムは、**m01.mod** よりは複雑になっているが、基本的な構成は変わらない。

n15.py は、Voltage-clamp を行った時の、Na^+ チャンネルのそれぞれの state の時間的経過を計算している。30, 31 行目では、python のコマンドの文字列を作成してから execute している。これは、z0〜z8 を含むコマンドを作るための手段である。

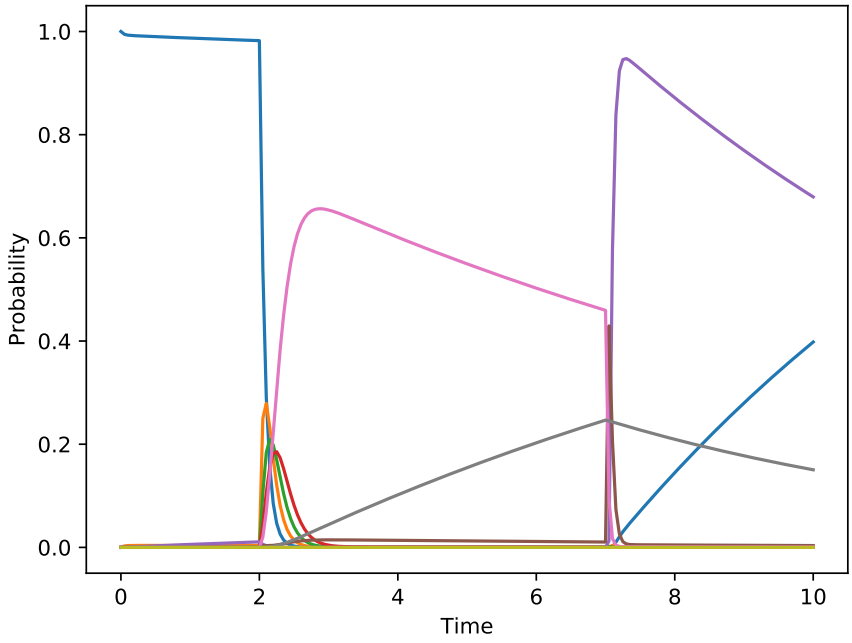


Fig 21　n15.py: Markov process of voltage-gated Na^+ channel

n16.py では、**n15.py** で得られる open probability を用いて、Conductance および電流の電位依存性を計算した。

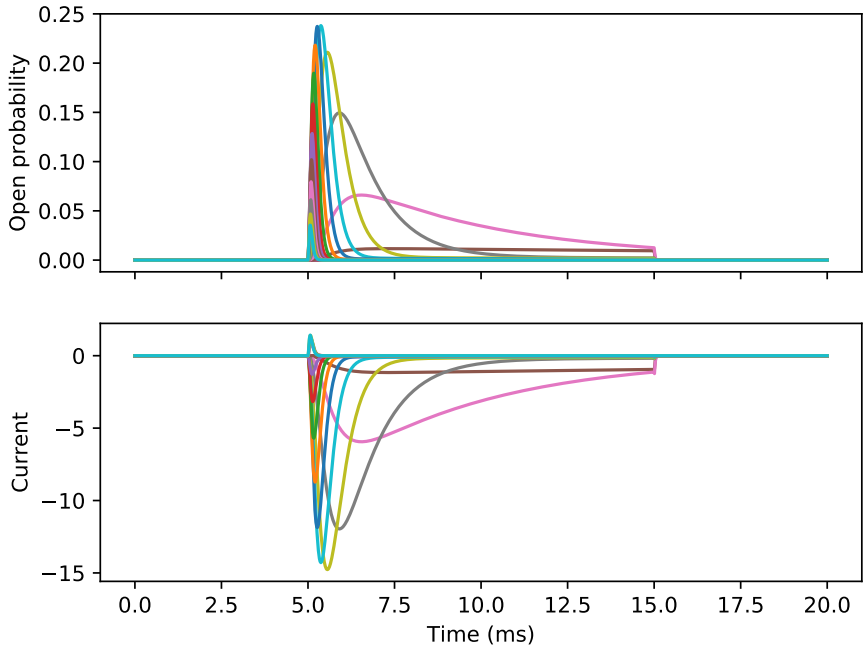


Fig 22　n16.py: Voltage-dependent gating of Na^+ channel

16　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。

$t = 0$ 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、

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

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

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。[*5]

[*5] $factor$ 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、

$$\frac{1}{tau2}\exp(-\frac{t_p}{tau2}) = \frac{1}{tau1}\exp(-\frac{t_p}{tau1})$$

$$tau1\exp(-\frac{t_p}{tau2}) = tau2\exp(-\frac{t_p}{tau1})$$

$$\log(tau1) - \frac{t_p}{tau2} = \log(tau2) - \frac{t_p}{tau1}$$


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。

17 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。

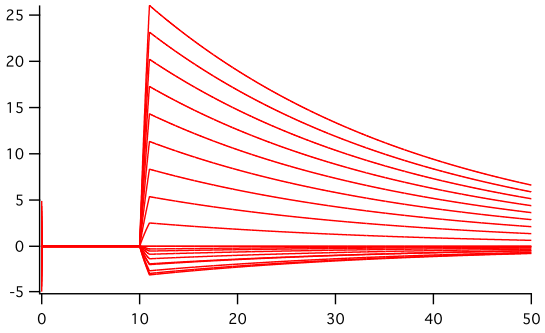

NET_RECEIVE block の引数。引数の数が 1 の場合、`NetCon` の weight が渡される。引数が $n+1$ 個の場合、最初の引数は、`NetCon` の weight であり、残りの引数はこの mod の変数を `NetCon` で記憶しておくために用いられる。これらの引数は、通常の”call by value”ではなく、”call by reference”で渡されるので、NET_RECEIVE block で変更された値は `NetCon` で保存される。

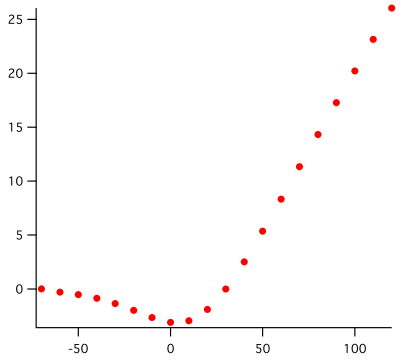
17.1 NMDA receptor channel

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

n15.py 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。





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

$$t_p=\frac{tau1*tau2}{tau2-tau1}\log(\frac{tau2}{tau1})$$

従って、

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

第 V 部

Network のシミュレーション

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。神経細胞のネットワークとなると、その目的に応じた神経細胞の数が必要となる。数千個の神経細胞のシミュレーションとなると、パソコンで行う場合は、神経細胞の単純化あるいは抽象モデル化が必要となる。ここでは、HH モデルの単純化した神経細胞モデルと抽象的なモデルの例を示す。

シミュレーション中のスパイクの記録は NetCon クラスの record() 関数を用いて行なうことができる。

18　A simplified HH neuron model: Ball-and-stick model

まずは、これまでの神経細胞に近い Ball-and-Stick モデルをつかって、簡単な細胞を扱う。このモデル細胞は、soma と 1 本の dendrite を持つ。

参考までに Python に書き換えた場合の例をあげておく。書き換えはほぼ機械的な作業であるが、hoc で動くのであれば、そのまま hoc ファイルを利用するのが効率的である。

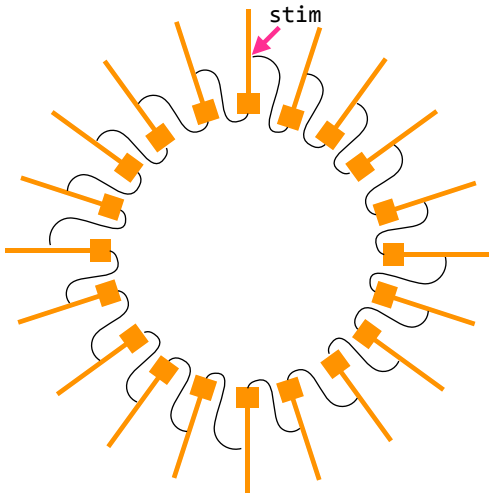


Fig 23　n15.py: Ring network の模式図

さて、ここからがネットワークシミュレーションで、20 個の神経細胞を環状にシナプス結合し、その一つに単発の外部刺激を与えている。

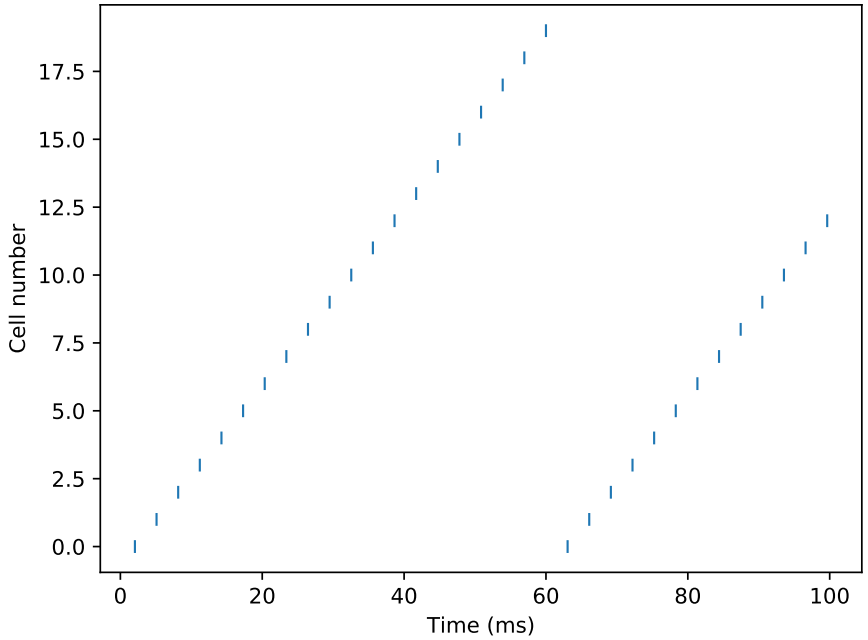


Fig 24　n17.py: spike の raster plot

18.1　Random network

次に、シナプス連絡をランダムに行う。神経細胞には、n15.py の場合と同じ Ball-and-Stick model を使用。このモデルでは、それぞれの神経細胞は、シナプスのリスト `synlist` を持っており、初めのシナプスは興奮性、次のシナプスは抑制性となっている。ランダム結合を作るために、興奮性シナプスのプレの細胞を乱数によって決めている。

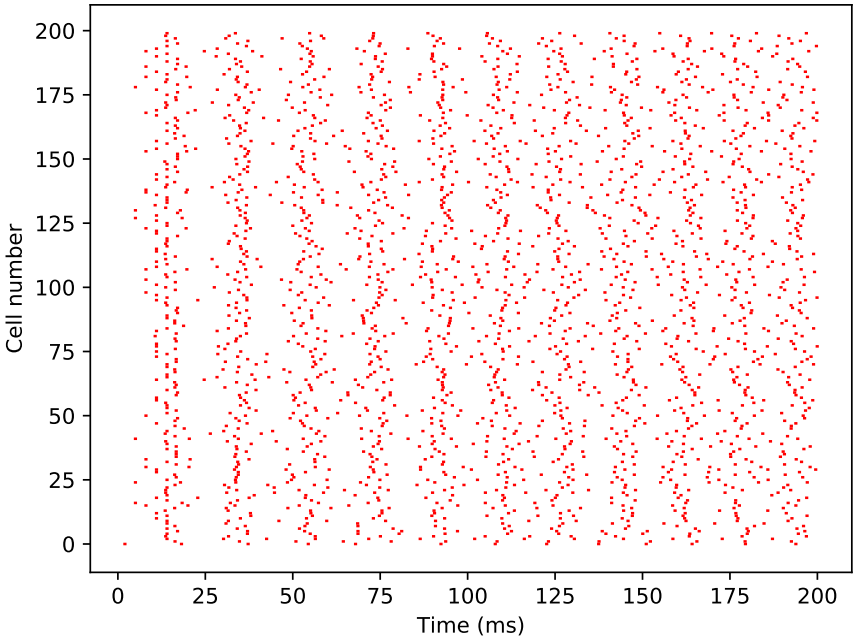


Fig 25　n18.py: spike の raster plot

19　Artificial neuron

19.1　Interval fire neuron

次により概念化したモデル細胞 Interval Fire を扱う。このモデルは、NEURON Book に掲載されている人工細胞であり、変数 m は、微分方程式

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

に従う。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。

m の値が増加していく、あるいは外からに入力 w があって m の値が w だけ増加し、m の値が 1 を越えると fire する。fire すると、event を発生し、$m = 0$ に戻る。

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。何を実行するかは、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。

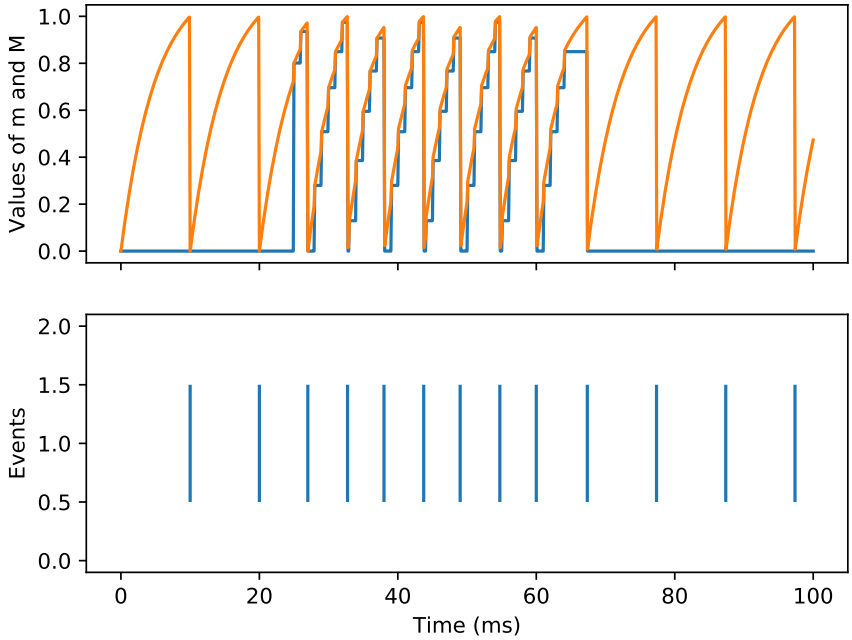


Fig 26　n19.py: m および M の時間的変化（上）と spike の raster plot。m の値は、event が起きた時だけ更新される。

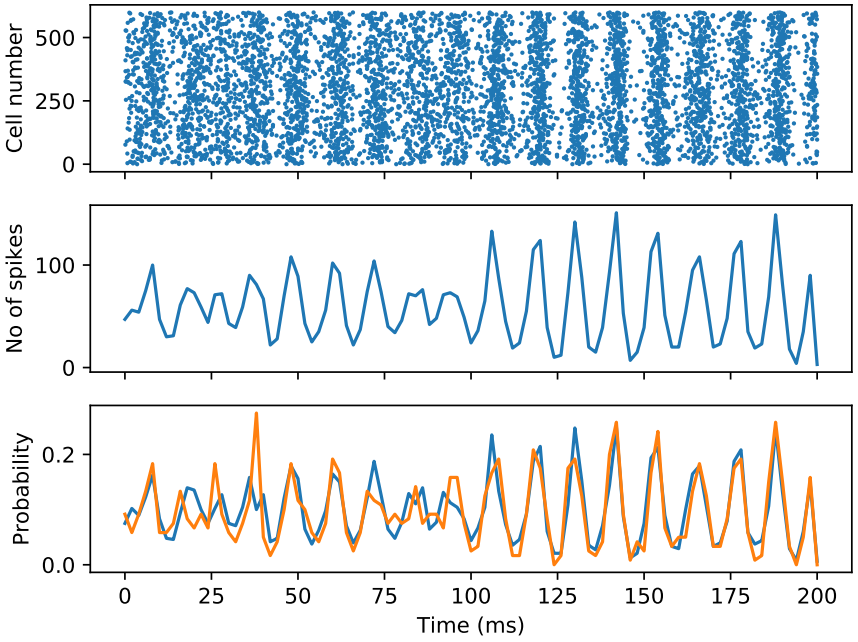


Fig 27 n20.py: Random network of artificial neurons. Raster plot of spikes.

第 VI 部
Appendix

A　NEURON のインストール

2017 年 12 月現在のバージョンは 7.5。インストールの方法は、当然のことながら OS によって異なる。

A.1　Windows でのインストール

もともと NEURON は、Mac で開発されたようで、MacOS や類似性の高い Linux ではあまり手間をかけずにフル機能を活用できるが、環境の異なる Windows の場合は、同じように動かすことが困難であった。しかし Windows 上で、Linux/MacOS に似た環境を提供するシステム (Cygwin、mingw、Msys など) とそれらにあったコンパイラ (GCC C/C++ コンパイラなど) が安定して使えるようになったことから、Windows でも比較的簡単に使えるようになっている。現在の Windows 版 NEURON は、mingw のサブセットを含んでいるが (C/C++ コンパイラも)、Python は含まれていない。

A.1.1　何が必要か？

どのように NEURON を利用するかによって、準備の程度は違ってくる。それぞれの場合、上から順番にインストールしていく。各項目の詳細は下に説明してある。この本の例題を実行するには、Python と NEURON が必要である。

1. NEURON を oc で利用する場合
   - NEURON
2. NEURON を python モードで利用する場合
   - Anaconda Python
   - NEURON

A.1.2　Python: Anaconda もしくは Miniconda のインストール

Python にはいろいろなバージョンがあるが、現時点では Anaconda 社のものを使うのが最も安定的である。その Anaconda 社の Python をインストールするには、パッケージ全体を含む Anaconda をインストールする方法と、最小限のインストールを行う Miniconda をインストールする方法の 2 つがある。Anaconda の場合、2 GB 以上というかなりのスペースを必要とするため、ここでは Miniconda のインストールを説明する。

インストールには、そのコンピュータの使用者すべてが使えるようにインストールする方法と、自分のフォルダー内にインストールする方法がある。前者は管理者権限を必要とする。下記は、後者の方法を説明している。

Miniconda のダウンロードサイト[*6]から、Python3.6 64-bit (exe installer) をダウンロード

Next → I Agree → Just me, Next → Destination Folder: C:\Users\xyz\Miniconda3 (default), Next

[*6] https://conda.io/miniconda.html

→ check: Add Anaconda to the system PATH environment variable, check: Register Anaconda as the system Python 3.6, Install → Completed, Next → Finish.

次の PATH が設定されていることが必要

- C:\Users\xyz\Minconda3
- C:\Users\xyz\Minconda3\Scripts
- C:\Users\xyz\Minconda3\Library\bin

Miniconda は最小限のシステムであるため、Python の必要なライブラリは個別にインストールすることが必要となる。NEURON を動かすために、とりあえず numpy、scipy、matplotlib をインストールする。
ターミナルから、

```
> conda install numpy
> conda install scipy
> conda install matplotlib
```

これらが正しくインストールされているかを確かめるには、python を起動して、sine カーブを描いてみる。

```
> python
Python 3.6.3 |Anaconda, Inc.  |(default, Nov 8 2017).  .  .
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> a = np.linspace(0.0, 10.0, 101)
>>> b = np.sin(a)
>>> plt.plot(a,b)
>>> plt.show()
```

A.1.3 NEURON

NEURON のダウンロードサイト[*7]から、Windows installer (64bit) をダウンロードする。

Destination folder: C:\Users\xyz\nrn\, Next → Set DOS environment にもチェック、Install → Completed, Close

PATH には C:\Users\xyz\nrn\bin が加えられている。

mod ファイルを使うには、シェルスクリプト mknrndll を使えるようにする必要がある。NEURON パッケージには、最低限の開発ツールが含まれており、それらがあるディレクトリ (C:\Users\xyz\nrn\mingw\bin) に PATH を通しておくことが必要。

NEURON のインストールができているかを確かめるためには、ターミナルから nrniv を入力する。

```
> nrniv
NEURON -- VERSION 7.5 master (6b4c19f) 2017-09-25
Duke, Yale, and the BlueBrain Project -- Copyright 1984-2016
```

[*7] https://www.neuron.yale.edu/neuron/download

```
See http://neuron.yale.edu/neuron/credits
oc>
```

Python モードで用いるには、先にインストールした python に関係する PATH を設定する必要がある。下記の環境変数を追加する。

- PYTHONHOME → C:\Users\xyz\Miniconda3
- NRN_PYLIB → C:\Users\xyz\Miniconda3\python36.dll

Python モードで NEURON を起動するには、ターミナルから nrniv -python と入力する。

```
> nrniv -python
NEURON -- VERSION 7.5 master (6b4c19f) 2017-09-25
Duke, Yale, and the BlueBrain Project -- Copyright 1984-2016
See http://neuron.yale.edu/neuron/credits
>>>
```

プロンプトが異なるので、python モードであることがわかる。また使用されている python を確認するには、

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

とすると python のバージョンなどが表示される。

Python 使用した matplotlib でグラフが表示されても、NEURON から matplotlib でプロットしようとすると、This application failed to start because it could not find or load the Qt platform plugin "windows" in " ". というエラーメッセージを発して止まることがある。この場合には、C:\Users\xyz\Minconda3 にある qt.conf というファイルを、c:\Users\xyz\nrn\bin にコピーする。

A.2 Mac OS でのインストール

A.2.1 何が必要か？

Windows の場合どう同様に、どのように NEURON を利用するかによって、準備の程度は違ってくる。それぞれの場合、上から順番にインストールしていく。各項目の詳細は下に説明してある。この本の例題を実行するには、NEURON の他に開発ツールが必要である。

1. NEURON を oc もしくは Python を、Mac OS の Python 2.7 で利用
   MacOS に含まれている Python を利用するが、matplotlib を使わない場合
   - NEURON
2. NEURON を oc もしくは Python を、Anaconda Python 3.6 で利用
   Anaconda Python を利用。スクリプトから matplotlib を用いてグラフを描かせる場合、あるいは Jupyter 環境などを利用する場合は、この Python を用いるのがよい。
   - Anaconda Python

- NEURON

3. mod ファイルを使用する場合
   C/C++ コンパイラが必要。
   - Xcode もしくは Xcode-select
   - Anaconda Python
   - NEURON

A.2.2 Xcode もしくは Xcode-select

Xcode は、MacOS の統合開発環境であり、C/C++ コンパイラ、Objective C, Swift などのプログラム言語を含む。高機能の統合開発環境であるが、快適に使用するには高機能の Mac でなくてはならない。ここでは、開発ツール類だけを含む Xcode-select をインストールする。

Terminal から、

```
xcode-select install
```

と入力し、あとは指示に従う。問題なくインストールできていれば、ターミナルから gcc と入力すると、no file inputs というようなメッセージが戻ってくるはずである。

A.2.3 Anaconda Python

システム全体を含む Anaconda と最小限のシステムである Miniconda がある。ここでは、Miniconda をインストールする。

1. ダウンロードサイト[*8]より、インストール用のスクリプトをダウンロードする（Mac OS X 64-bit (bash installer)）
2. ターミナルから、スクリプトを起動。
   `bash Miniconda3-latest-MacOSX-x86_64.sh`
   インストール部位によっては管理者権限が必要なので、
   `sudo bash Miniconda3-latest-MacOSX-x86_64.sh`
   として、あとは指示に従う。
   default のインストール部位は、ユーザーの HOME ディレクトリであるが、複数人数で使用する場合には、/usr/local にインストールするのが無難だと思われる。
3. 環境変数 PATH は、インストール時に自動的に更新される。
4. 環境変数 PYTHONHOME を設定
   Miniconda3 を/usr/local にインストールした場合は、`export PYTHONHOME=/usr/local/miniconda3`
5. Anaconda をインストールした場合は、全てのライブラリがインストールされているが、Miniconda をインストールした場合は、必要なライブラリをインストールする。NEURON を動かすために、とりあえず numpy、scipy、matplotlib をインストールする。
   ターミナルからとして、
   ```
   > conda install numpy
   > conda install scipy
   ```

[*8] https://conda.io/miniconda.html

```
> conda install matplotlib
```

A.2.4 NEURON

NEURON のダウンロードサイトより nrn-7.5.x86_64-osx.pkg をダウンロードして、インストールすれば事足りる。プログラムは/Applications にインストールされる。

コマンドで使用するのは、nrniv と nrnivmodl なので、これらに PATH を通しておくと便利である。

python モードで使用するには、環境変数 PYTHONHOME と NRN_PYLIB を設定する必要がある。いずれの場合も、.bash_profile に書き加える。

- Mac OS の Python を使用
  ```
  export PYTHONHOME=/System/library/Frameworks/Python.framework/versions/2.7
  export NRN_PYLIB=$PYTHONHOME/lib/libpython2.7.dylib
  ```
- Anaconda Python を使用 Miniconda3 を/usr/local にインストールした場合
  ```
  export PYTHONHOME=/usr/local/miniconda3
  export NRN_PYLIB=/usr/local/miniconda3/lib/libpython3.6m.dylib
  ```


Matplotlab を用いたプロットと、hoc の Graph を用いたプロットの両方を使用すると、hoc の Graph が描出されない。解決策は現在のところ不明。ただ、ファイルに出力することは可能。

B PATH を通す

コマンドプロンプト (Windows) もしくはターミナル (MacOS) から、コマンドを入力してプログラムを実行する場合、プログラムをフルのパスを入力して実行することができる。

C:\Users\xyz\nrn\bin\nrniv

しかしこれは面倒なので、プログラムを実行する場合に探す場所を指定する環境変数 PATH にプログラムのフォルダ（ディレクトリ）を追加しておくと、プログラム名だけの入力で実行することができる。環境変数 PATH にフォルダをついかすることを、「PATH を通す」という。

PATH の内容を知るには、

echo %PATH% （Windows の場合）

echo $PATH （MacOS の場合）

とすれば、表示される。

B.1 Windows の場合

システムの PATH を変更する場合は、管理者権限が必要

(コントロールパネル → システム → システムの詳細設定 → Environmental Variables → System variables → Path → New ...)

個人用の PATH を変更する場合は、管理者権限は不必要。

(コントロールパネル → ユーザーアカウント → 環境変数の変更 → xyz のユーザー環境変数 → Path → New ...)

B.2 MacOS の場合

MacOS では、下記の方法で PATH の設定を行う事ができる。

1. ~/.bash_profile を使用。最も古典的な方法。コマンドラインからプログラムを起動する時は問題ないが、アイコンをクリックして起動する場合に PATH を読み込まない、という問題が生じる。
   Linux ではふつう ~/.bash_rc ファイルに PATH を書くようであるが、MacOS では使用されない。
2. /etc/paths
   管理者権限が必要である。先に現れる方が優先度が高くなる。/etc/paths.d を使う事も可能。
3. すでに PATH が通っているところに、symbolic link を作るという方法もある。これは手軽です。例えば、すでに ~/$HOME/bin に PATH が通っている場合、
   `$ cd ~/$HOME/bin`
   `$ ln -s /Applications/NEURON-7.5/nrn/x86_64/bin/nrniv nrniv`
   で symbolic link を張ることができる。

C Vector データのファイルへの保存

Vector を、hoc 関数の as_numpy() で Python の array に変換し、適当なグループにまとめて、テキストファイルに出力するのが簡単。出力したい Vector を v1、v2、v3 (いずれも 1 次元の配列で、サイズが同じであることが条件) とし、出力ファイルを ax.txt としたとき、

```
numpy.savetxt('ax.txt', numpy.stack((v1.as_numpy(), v2.as_numpy(), v3.as_numpy(),axis=-1))
```

とすればよい。

D グラフの描き方

NEURON の計算結果は、通常、Vector に保存される。このデータを可視化するには、いろいろな方法がある。

1. NEURON のグラフ機能を利用する
   hoc の Vector は、基本的には実数の配列であるが、Vector クラスには様々な機能が作り込まれている。その一つは、グラフの機能であり、Graph クラスのインスタンスを用意しておくことにより、簡単にグラフを表示することができる。hoc version のスクリプトは、この方法を使用している。
2. Python の matplotlib ライブラリを使用する
   Python version のスクリプトは、この方法を使用している。
3. 外部プログラムを使用する
   ファイルに出力してそれを利用する。ここでは、時間の配列 t と、m 個の変数配列 $x_0, x_1, \ldots, x_{m-1}$ が、テキストエディタで開いた時に、

$$
\begin{array}{ccccc}
t[0] & x_0[0] & x_1[0] & \dots & x_{m-1}[0] \\
t[1] & x_0[1] & x_1[1] & \dots & x_{m-1}[1] \\
t[2] & x_0[2] & x_1[2] & \dots & x_{m-1}[2] \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
t[n-1] & x_0[n-1] & x_1[n-1] & \dots & x_{m-1}[n-1]
\end{array}
$$

という数値の並び方となることを想定している。列と行が逆の並び方もありうるが、テキストエディタでデータファイルを見る場合に、こちらの方が見やすい場合が多い（例えばデータ数が数千の場合など)。ただ、使用するプログラムに合わせて、出力ファイルの形式を変更する必要があるかもしれない。

- `gnuplot`[*9]
  フリーソフト。`NEURON` スクリプトとの一体性を考えると、Matplotlib の方が便利だと思われる。ただ、3-D プロットは gnuplot の方が優れている。

  ```
  plot 'ax.txt' using 1:2 with lines, \
  'ax.txt' using 1:3 with lines, \
  'ax.txt' using 1:4 with lines
  ```
  gnuplot のスクリプトを作成しておくと使い勝手がよい。

- R[*10]
  フリーソフト。経験がないのでわからない。
- Igor Pro (Wavemetrics)[*11]
  商用ソフト。1 次元のトレースデータの操作に適している。グラフ出力も十分。
- Microsoft Excel
  商用ソフト。急ぎの時には便利かもしれない。Excel に読み込ませるには、csv 形式 (comma-separated values) で保存するようにするとよい。
- 自分で好みのプログラムを書く？
  C/C++ などのコンピュータ言語で計算し、その結果をすぐに表示したい場合などに必要。ファイルに出力し、Python のスクリプトで matplotlib を用いて表示するのが簡便な方法。

  ```
  import sys
  import numpy as np
  import matplotlib.pyplot as plt

  if len(sys.argv) != 2:
      print('Usage: python plot.py filename')
      exit()
  filename=sys.argv[1]
  ```

[*9] http://www.gnuplot.info
[*10] https://www.r-project.org
[*11] https://www.wavemetrics.com/index.html

```
print("file name = ", filename)

aa = np.loadtxt(filename, unpack=True)
nitems, npts = aa.shape
plt.figure()
for i in range(1,nitems):
    plt.plot(aa[0],aa[i])
plt.show()
```

E　動画の作り方

例を **n21.py** に示した。長い axon の両端を刺激した場合に、活動電位は両端から真ん中に向かって伝搬していくが、活動電位の波が衝突するとそこで消えてしまう。

F References

- Carnevale NT, Hines ML (2006)
  "The NEURON Book" Cambridge University Press.
  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。
- 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。
- NEURON + Python Basics
  基本的なチュートリアル
  https://neuron.yale.edu/neuron/static/docs/neuronpython/firststeps.html
- NEURON Python documentation
  https://www.neuron.yale.edu/neuron/static/py_doc/index.html
- NEURON 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.
- Brette R, Rudolph M, Carnevale T, Hines M, Beeman D, Bower JM, Diesmann M, Morrison A, Goodman PH, Harris FC Jr, Zirpe M, Natschläger T, Pecevski D, Ermentrout B, Djurfeldt M, Lansner A, Rochel O, Vieville T, Muller E, Davison AP, El Boustani S, Destexhe A (2007).
  Simulation of networks of spiking neurons: a review of tools and strategies.
  J Comput Neurosci. 23:349-398.
  ネットワークシミュレーションを、NEURON を含めたいろいろなプラットフォームで試したもの。プログラムのソースコードが提供されているので（上記の ModelDB に登録されている）、プロフェッショナルのプログラム技法を学ぶとともに実際試してみることができる。