

Julia で学ぶ計算論的神経科学

山本 拓都

2024 年 3 月 31 日

目次

第1章	はじめに	9
1.1	神経科学と数理モデル	9
1.2	記号の表記	10
1.2.1	変数の命名規則	10
1.3	Julia 言語の基本構文	10
1.3.1	変数	10
1.3.2	条件分岐	11
1.3.3	再帰的処理	11
1.3.4	関数	11
1.3.5	数値計算	12
1.3.6	その他の関数について	12
1.4	線形代数	12
1.4.1	要素ごとの演算	13
1.4.2	配列に新しい軸を追加	14
1.4.3	Roth's column lemma	15
1.4.4	ArrayArrayFloat64, $x, 1$ を ArrayFloat64, $x+1$ に変換	16
1.5	微分方程式	17
1.5.1	微分方程式の基礎	17
1.5.2	連続時間モデルから離散時間モデルへの変換	20
1.6	線形回帰と最小二乗法	21
1.6.1	線形回帰	21
1.6.2	最小二乗法によるパラメータの推定	21
1.7	確率論	24
1.7.1	期待値 (Expectation)	24
1.7.2	情報量 (Information)	24
1.7.3	平均情報量 (エントロピー, entropy)	25

1.7.4	Kullback-Leibler 情報量	25
1.7.5	相互情報量 (Mutual information)	25
1.8	確率過程と確率微分方程式	25
第 2 章 神経細胞のモデル		27
2.1	神経細胞の形態と生理	27
2.2	コンダクタンスベースモデル	27
2.2.1	Hodgkin-Huxley モデル	27
2.2.2	Connor-Stevens モデル	31
2.2.3	F-I 曲線	33
2.2.4	全か無かの法則の反例	35
2.3	FitzHugh-Nagumo モデル	36
2.3.1	FitzHugh-Nagumo モデルの定義	36
2.3.2	FitzHugh-Nagumo モデルのシミュレーションの実行	38
2.3.3	相図の描画	39
2.4	積分発火モデル	40
2.4.1	Leaky integrate-and-fire モデル	40
2.4.2	LIF モデルのシミュレーションの実行	43
2.4.3	LIF モデルの F-I curve	43
2.5	Izhikevich モデル	47
2.5.1	Izhikevich モデルの定義	47
2.5.2	Izhikevich モデルのシミュレーションの実行	49
2.5.3	様々な発火パターンのシミュレーション	50
2.6	Inter-spike interval モデル	52
2.6.1	ポアソン過程モデル	52
2.6.2	死時間付きポアソン過程モデル (Poisson process with dead time, PPD)	58
2.6.3	ガンマ過程モデル	59
2.7	神経突起の成長モデル	61
2.7.1	神経突起の木構造	62
2.7.2	Van Pelt モデル	64
参考文献		70
第 3 章 シナプス伝達のモデル		71
3.1	シナプスの形態と生理	71
3.2	Current / Conductance-based シナプス	71

3.2.1	化学シナプスの2つの記述形式	71
3.2.2	Current-based シナプス	72
3.2.3	Conductance-based シナプス	72
3.3	指数関数型シナプスモデル	73
3.3.1	单一指数関数型モデル (Single exponential model)	74
3.3.2	二重指数関数型モデル (Double exponential model)	75
3.4	動力学モデル	77
3.4.1	チャネル動態の動力学的表現	77
3.4.2	Hodgkin-Huxley モデルにおけるシナプスモデル	79
3.5	シナプス入力の重みづけ	81
3.6	動的シナプス	82
参考文献		85
第4章	神経回路網の演算処理	87
4.1	ゲイン調節と四則演算	87
参考文献		90
第5章	局所学習則	91
5.1	Hebb 則と教師なし学習	91
5.1.1	Hebb 則	91
5.1.2	Hebb 則の安定化と LTP/LTD	92
5.1.3	Hebb 則と主成分分析	95
5.1.4	非線形 Hebb 学習	99
5.2	Slow Feature Analysis (SFA)	104
5.3	自己組織化マップと視覚野の構造	108
5.3.1	単純なデータセット	108
5.3.2	視覚野マップ	114
参考文献		120
第6章	生成モデルとエネルギーベースモデル	121
6.1	エネルギーベースモデル	121
6.2	Hopfield モデル	121
6.2.1	モデルの定義	122
6.2.2	稠密連想記憶 (dense associative memory) モデル	125
6.3	Boltzmann マシン	126
6.3.1	Boltzmann マシン	126

6.3.2	制限 Boltzmann マシン	126
6.4	スパース符号化	131
6.4.1	Sparse coding と生成モデル	131
6.4.2	目的関数の設定と最適化	133
6.4.3	Locally competitive algorithm (LCA)	135
6.4.4	重み行列の更新則	138
6.4.5	Sparse coding network の実装	138
6.5	予測符号化	147
6.5.1	観測世界の階層的予測	147
6.5.2	損失関数と学習則	148
参考文献	156
第 7 章	貢献度分配問題の解決策	157
7.1	勾配法と誤差逆伝播法	157
7.1.1	ニューラルネットワークモデル	157
7.1.2	Zipser-Andersen モデル	161
7.2	BPTT (backpropagation through time)	166
参考文献	172
第 8 章	運動制御	173
8.1	躍度最小モデル	173
8.1.1	等式制約下の二次計画法 (Equality Constrained Quadratic Programming)	173
8.1.2	躍度最小モデルの実装	174
8.1.3	経由点を通る場合	176
8.2	終点誤差分散最小モデル	177
8.2.1	終点誤差分散最小モデルの実装	178
8.3	最適フィードバック制御モデル	180
8.3.1	最適フィードバック制御モデルの構造	180
8.3.2	実装	182
8.4	無限時間最適フィードバック制御モデル	188
8.4.1	モデルの構造	188
8.4.2	実装	188
8.4.3	Target jump	193
8.5	ラット自由行動下の軌跡のシミュレーション	194
参考文献	198

第 9 章	神経回路網によるベイズ推論	199
9.1	ベイズ線形回帰 (Bayesian linear regression)	199
9.2	マルコフ連鎖モンテカルロ法 (MCMC)	201
9.2.1	Metropolis-Hastings 法	202
9.2.2	ランジュバン・モンテカルロ法 (LMC)	204
9.2.3	ハミルトニアン・モンテカルロ法 (HMC 法)	206
9.2.4	線形回帰への適応	207
9.3	神経サンプリング	210
9.3.1	ガウス尺度混合モデル	211
9.3.2	興奮性・抑制性神経回路によるサンプリング	219
9.3.3	Spiking ニューラルネットワークにおけるサンプリング	223
9.3.4	シナプスサンプリング	224
9.4	確率的集団符号化 (probabilistic population coding)	224
参考文献	225

第1章

はじめに

1.1 神経科学と数理モデル

本書では神経科学における数理モデルを主として取り扱う。初めに神経科学におけるモデルの役割について触れておこう。まず、神経科学の目標は端的に言えば「脳神経系を理解する」ことにある。神経科学に限らず、種々の学問分野においては実験と理論の2本柱で、対象とする現象や物質の理解が進められる。ここで実験は調査等も踏まえ実データを取る行為とする。理論の役割は複数あり、実験結果の抽象化、仮説の提供、現象の予測等である (Blohm et al., 2020)。「脳神経系を理解する」ということに関して、その定義は研究者により様々である。ここでは脳の計算処理に関する理論的理解を進めるための1つの方法として Marr の3レベル (Marr's Three Levels) を紹介する (Marr, 1982)。Marr の3レベルは視覚系における計算処理の理解を主としていたが、他でも適用可能である。3レベルとは(1)計算理論 (computational theory), (2)表現・アルゴリズム (representation and algorithm), (3)実装 (implementation) であり、それぞれの段階での議論や理解を行う。(1)では脳の目的関数とそれを用いた最適化問題の設定を行う。(2)では(1)を実現するための表現およびアルゴリズムを解明する。(3)では(1,2)を神経回路・ハードウェア上で実装する方法を解明することを目標とし、平易には「脳」を作つて理解すると言い換えることもできる^{*1}。本書ではこの(3)を重視し、読者が自らの手で理論を検証し、数値計算による結果を再現できることを目標とした。また、本書は数式をプログラミングのコードに変換する具体例集としての役割も持っている。モデルの中でも、本書では機械学習に関連する内容が多数登場する。これは神経科学と機械学習は互いに影響を及ぼし合ってきたためである (Hassabis et al., 2017)。神経科学から機械学習への応用は例えば、ニューラルネットワーク、記憶モデル、注意モデルなどがある。逆に機械学習から神経科学への応用は強化学習、運動制御、ベイズ脳仮説などが挙げられる。

^{*1} ここで「作る」は計算機等でシミュレーションするという意味であり、脳オルガノイド (brain organoid) を作成するなどの意味ではない

1.2 記号の表記

本書では次のような記号表記を用いる。

- 実数全体を \mathbb{R} , 複素数全体は \mathbb{C} と表記する。
- スカラーは小文字・斜体で x のように表記する。
- ベクトルは小文字・立体・太字で \mathbf{x} のように表記し, 列ベクトル(縦ベクトル)として扱う。
- 行列は大文字・立体・太字で \mathbf{X} のように表記する。
- $n \times 1$ の実ベクトルの集合を \mathbb{R}^n , $n \times m$ の実行列の集合を $\mathbb{R}^{n \times m}$ と表記する。
- 行列 \mathbf{X} の置換は \mathbf{X}^\top と表記する。ベクトルの要素を表す場合は $\mathbf{x} = (x_1, x_2, \dots, x_n)^\top$ のように表記する。
- 単位行列を \mathbf{I} と表記する。
- ゼロベクトルは $\mathbf{0}$, 要素が全て 1 のベクトルは $\mathbf{1}$ と表記する。
- e を自然対数の底とし, 指数関数を $e^x = \exp(x)$ と表記する。また, 自然対数を $\ln(x)$ と表記する。
- 定義を $:=$ を用いて行う。例えば, $f(x) := 2x$ は $f(x)$ という関数を $2x$ として定義するという意味である。定義する対象が右側である場合は, $=:$ を用いる。
- 平均 μ , 標準偏差 σ の正規分布を $\mathcal{N}(\mu, \sigma^2)$ と表記する。

1.2.1 変数の命名規則

- `tp1`, `tm1` : time plus one ($t+1$), time minus one ($t-1$)

1.3 Julia 言語の基本構文

1.3.1 変数

変数への代入は `=` で行う。変数は Unicode も使用可能であり, LaTeX コマンドと TAB キーで入力することが可能である。

```
x = 1
α = 2 # \alpha + TAB key
```

`var` を用いることで, 任意の文字列を変数にすることができる。

```
var"log(1+θ)" = 3
```

演算子	説明	使用例	結果
+	和	aaa	aaa
-	差	aaa	aaa
*	積	aaa	aaa
.*	配列の要素積	aaa	aaa
/	除算, 右から逆行列をかける	aaa	aaa
\	左から逆行列をかける	aaa	aaa

1.3.2 条件分岐

if 構文を用いることで条件分岐が可能である.

```
a = 2
if a > 0
    print("positive")
elseif a == 0
    print("zero")
else
    print("negative")
end
```

1.3.3 再帰的処理

再帰的な処理を行う場合には主に **forloop** 構文を用いる.

```
x = 1
for i in 1:10
    x += 1
end
println(x)
```

1.3.4 関数

function により関数を定義する. なお, 慣習として関数への入力を変更する場合に!を付けることがある. 関数内で配列を変更する場合には注意が必要である. 以下に入力された配列を同じサイズの要素1の配列で置き換える, ということを目的として書かれた2つの関数がある. 違いは v の後に [:] しているかどうかである.

```
function wrong!(a::Array)
```

```
a = ones(size(a))
end

function right!(a::Array)
    a[:] = ones(size(a))
end
```

実行すると `wrong!` の場合には入力された配列が変更されていないことがわかる。

```
using Random
v = rand(2, 2)
println("v : ", v)

wrong!(v)
println("wrong : ", v)

right!(v)
println("right : ", v)
```

1.3.5 数値計算

`broadcasting` の回避を行うには以下のような方法がある。

```
foo(a,b) = sum(a) + b
```

```
println(foo.(Ref([1,2]),[3,4,5]))
println(foo.(([1,2],), [3,4,5]))
println(foo.([[1,2]], [3,4,5]))
```

1.3.6 その他の関数について

Julia の余りの関数は `rem(x, y)` と `mod(x, y)` がある。Julia の `x % y` は `rem` と同じだが、Python の場合は `mod` と同じなので注意。

```
println("% : ", -1 % 2, ", rem : ", rem(-1, 2), ", mod : ", mod(-1, 2))
```

1.4 線形代数

aaa

```
using LinearAlgebra
```

ベクトル

Julia は列ベクトルが基本. $\mathbf{a} = [a_1, a_2, \dots, a_n]^\top$

```
a = [1, 2, 3]
```

行列

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix} \in \mathbb{R}^n \quad (1.1)$$

行列 \mathbf{A} の (i, j) 成分を $\mathbf{A}_{ij} = a_{ij}$ とする.

```
A = []
```

1.4.1 要素ごとの演算

行列の行・列ごとの正規化

シミュレーションにおいてニューロン間の重み行列を行あるいは列ごとに正規化 (weight normalization) する場合がある. これは各ニューロンへの入力の大きさと同じにする働きや重みの発散を防ぐ役割がある. 以下では行ごとの和を 1 にする.

```
W = rand(3,3)
```

```
Wnormed = W ./ sum(W, dims=1)
```

```
println(sum(Wnormed, dims=1))
```

行列の結合 (concatenate)

行列の結合は MATLAB に近い形式で行うことができる. まず, 2 つの行列 A, B を用意する.

```
A = [1 2; 3 4]
```

```
B = [4 5 6; 7 8 9]
```

水平結合 (Horizontal concatenation)

`hcat` を使うやり方と、`[]` を使うやり方がある。

```
H1 = hcat(A,B)
```

```
H2 = [A B]
```

なお、MATLAB のように次のようにすると正しく結合はされない。

```
H3 = [A, B]
```

垂直結合 (Vertical concatenation)

aaa

```
V1 = vcat(A, B')
```

```
V2 = [A; B']
```

```
[V2 [A;B']]
```

1.4.2 配列に新しい軸を追加

要は numpy での `A[None, :]` や `A[np.newaxis, :]` のようなことがしたい場合。やや面倒だが、`reshape` を使うか、`[CartesianIndex()]` を用いる。

```
v = rand(3)
```

```
newaxis = [CartesianIndex()]
v1 = v[newaxis, :]
```

単位行列

aaa

```
I
```

```
I(3)
```

対角行列

```
aaa
```

線形行列方程式

$\mathbf{Ax} = \mathbf{b}$ は \mathbf{A} が正則の場合、逆行列が存在し、 $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ が解となる。

```
A = rand(2,2)
b = rand(2)
```

```
x = inv(A) * b
```

Julia ではバックスラッシュ演算子 `\` を用いることで明示的に逆行列を計算せずに解を求めることができる。

```
x = A \ b
```

1.4.3 Roth's column lemma

Roth's column lemma は、例えば、 A, B, C が与えられていて、 X を未知とするときの方程式 $AXB = C$ を考えると、この方程式は

$$(B^\top \otimes A)\text{vec}(X) = \text{vec}(AXB) = \text{vec}(C) \quad (1.2)$$

の形に書き下すことができる、というものである。 $\text{vec}(\cdot)$ は vec 作用素（行列を列ベクトル化する作用素）である。 $\text{vec}(X) = \text{vcat}(X\dots)$ で実現できる。Roth's column lemma を用いれば、 $AXB = C$ の解は

$$X = \text{vec}^{-1}((B^\top \otimes A)^{-1}\text{vec}(C)) \quad (1.3)$$

として得られる。ただし、 $\text{vec}(\cdot)^{-1}$ は列ベクトルを行列に戻す作用素（inverse of the vectorization operator）である。 $\text{reshape}()$ で実現できる。2つの作用素をまとめると、

$$\text{vec} : R^{m \times n} \rightarrow R^{mn} \quad (1.4)$$

$$\text{vec}^{-1} : R^{mn} \rightarrow R^{m \times n} \quad (1.5)$$

であり、 $\text{vec}^{-1}(\text{vec}(X)) = X$ (for all $X \in R^{m \times n}$)、 $\text{vec}(\text{vec}^{-1}(x)) = x$ (for all $x \in R^{mn}$) となる。

```
using LinearAlgebra, Kronecker, Random

m = 4
A = randn(m, m)
B = randn(m, m)
C = convert(Array{Float64}, reshape(1:16, (m, m)))

X = reshape((B' ⊗ A) \ vec(C), (m, m))

A * X * B
```

配列の 1 次元化

配列を一次元化 (flatten) する方法. まずは 3 次元配列を作成する.

```
B = rand(2, 2, 2)
```

用意されている `flatten` を素直に用いると次のようになる.

```
import Base.Iterators: flatten
collect(flatten(B))
```

ただし, 単に `B[:, :]` とするだけでもよい.

```
B[:, :]
```

reshape における残りの次元の指定

numpyにおいては (2, 3, 5) 次元の配列に対し, `reshape(-1, 5)` を行うと (6, 5) 次元の配列となった. これと同様なことは, Juliaでは: を使うことで実装できる.

```
a = rand(2,3,5)
b = reshape(a, (:, 5))
```

1.4.4 Array{Array{Float64, 1}, 1} を Array{Float64, 1} に変換

numpyでは `array([matrix for i in range()])`などを用いると, 1 次元配列のリストを 2 次元配列に変換できた. Juliaでも同様にする場合は `hcat(...)` や `cat(...)` を用いる. ... は `splat` operator と呼ばれる.

```
A1 = [i*rand(3) for i=1:5]
println("Type : ", typeof(A1))
println("Size : ", size(A1))
```

```
A2 = hcat(A1...)
println("Type : ", typeof(A2))
println("Size : ", size(A2))
```

```
stack(A1)
```

以下は多次元配列の場合. `cat(...)` で配列を結合し, `permutedims` で転置する.

```
B1 = [i*rand(3, 4, 5) for i=1:6]
println("Type : ", typeof(B1))
println("Size : ", size(B1))
```

```
stack(B1)
```

```
B2 = permutedims(cat(B1..., dims=4), [4, 1, 2, 3])
println("Type : ", typeof(B2))
println("Size : ", size(B2))
```

1.5 微分方程式

1.5.1 微分方程式の基礎

微分方程式はある関数とそれを微分した導関数の関係式であり, 関数の特定の変数に対する変化を記述することができる. まず, 1階線形微分方程式を例として見てみよう.

$$\frac{dx(t)}{dt} = a_c x(t) + b_c u(t) \quad (1.6)$$

状態変数 $x(t)$ は, 時間 t に対する関数である. 添え字の c は連続 (continuous) を意味するが, これは後で離散化する際に区別するためである. この方程式においては $b_c = 0$ の場合を同次方程式, $b_c \neq 0$ の場合を非同次方程式という.

微分方程式の解

微分方程式を解くとは $x(t)$ のような関数の具体的な式を求めることである。上式の解は

$$x(t) = e^{a_c t} x(0) + \int_0^t e^{a_c(t-\tau)} b_c u(\tau) d\tau \quad (1.7)$$

として与えられる。微分方程式を解く手法は様々で、それの方程式について適切な手法を選択する。本書では Laplace 変換を多用するが、細かい説明は付録にて行う。

連立線形微分方程式

n 個の微分方程式連立線形微分方程式という。これをベクトル、行列を用いて時不变 (time-invariant) の定数行列を $\mathbf{A}_c \in \mathbb{R}^{n \times n}$, $\mathbf{B}_c \in \mathbb{R}^{n \times m}$, 状態ベクトルを $\mathbf{x}(t) \in \mathbb{R}^n$, 入力ベクトルを $\mathbf{u}(t) \in \mathbb{R}^m$ とする。

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{A}_c \mathbf{x}(t) + \mathbf{B}_c \mathbf{u}(t) \quad (1.8)$$

解は

$$\mathbf{x}(t) = e^{t\mathbf{A}_c} \mathbf{x}(0) + \int_0^t e^{(t-\tau)\mathbf{A}_c} \mathbf{B}_c \mathbf{u}(\tau) d\tau \quad (1.9)$$

ラプラス変換

Laplace 変換は Fourier 変換に似た手法であり、微分方程式を解く上で便利である。ToDo: Laplace 変換の詳細

$$F(s) := \int_0^\infty f(t) e^{-st} dt = \mathcal{L}(f(t)) \quad (1.10)$$

e^{-st} を引つ付けて積分することで、被積分関数が $t \rightarrow \infty$ で収束し、積分可能となっている。実用上は次の対応表を用いて計算すればよい。ToDo: Laplace 変換の対応表

1 階線形行列微分方程式の解

時不变 (time-invariant) の定数行列を $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times m}$, 状態ベクトルを $\mathbf{x}(t) \in \mathbb{R}^n$, 入力ベクトルを $\mathbf{u}(t) \in \mathbb{R}^m$ とする。

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad (1.11)$$

この線形行列微分方程式を Laplace 変換 \mathcal{L} を用いて解こう。 $\mathbf{X}(s) := \mathcal{L}(\mathbf{x}(t))$, $\mathbf{U}(s) := \mathcal{L}(\mathbf{u}(t))$ とすると、

$$s\mathbf{X}(s) - \mathbf{x}(0) = \mathbf{A}\mathbf{X}(s) + \mathbf{B}\mathbf{U}(s) \quad (1.12)$$

$$(s\mathbf{I} - \mathbf{A})\mathbf{X}(s) = \mathbf{x}(0) + \mathbf{B}\mathbf{U}(s) \quad (1.13)$$

$$\mathbf{X}(s) = (s\mathbf{I} - \mathbf{A})^{-1}(\mathbf{x}(0) + \mathbf{B}\mathbf{U}(s)) \quad (1.14)$$

$$(1.15)$$

行列指数関数 (matrix exponential) は

$$e^{\mathbf{A}} = \exp(\mathbf{A}) := \sum_{k=0}^{\infty} \frac{1}{k!} \mathbf{A}^k = \mathbf{I} + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \cdots \quad (1.16)$$

として定義される。天下り的だが、

$$\mathcal{L}(e^{at}) = \frac{1}{s-a} \quad (1.17)$$

$$\mathcal{L}(e^{t\mathbf{A}}) = (s\mathbf{I} - \mathbf{A})^{-1} \quad (1.18)$$

$$(1.19)$$

となる。よって

$$\mathbf{X}(s) = (s\mathbf{I} - \mathbf{A})^{-1}(\mathbf{x}(0) + \mathbf{B}\mathbf{U}(s)) \quad (1.20)$$

$$= (s\mathbf{I} - \mathbf{A})^{-1}\mathbf{x}(0) + (s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B}\mathbf{U}(s) \quad (1.21)$$

$$\mathbf{x}(t) = e^{t\mathbf{A}}\mathbf{x}(0) + \int_0^t e^{(t-\tau)\mathbf{A}}\mathbf{B}\mathbf{u}(\tau)d\tau \quad (1.22)$$

となる。最後の式は両辺を逆 Laplace 変換した。ここで、 $\mathcal{L}^{-1}(F(s)G(s)) = \int_0^t f(\tau)g(t-\tau)d\tau$ であることを用いた。区間 $[t, t + \Delta t]$ において入力 $\mathbf{u}(t)$ が一定であると仮定すると、

$$\mathbf{x}(t + \Delta t) = e^{(t+\Delta t)\mathbf{A}}\mathbf{x}(0) + \int_0^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}}\mathbf{B}\mathbf{u}(\tau)d\tau \quad (1.23)$$

$$= e^{\Delta t \mathbf{A}} e^{t\mathbf{A}}\mathbf{x}(0) + e^{\Delta t \mathbf{A}} \int_0^t e^{(t-\tau)\mathbf{A}}\mathbf{B}\mathbf{u}(\tau)d\tau + \int_t^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}}\mathbf{B}\mathbf{u}(\tau)d\tau \quad (1.24)$$

$$\approx \underbrace{e^{\Delta t \mathbf{A}}}_{=: \mathbf{A}_d} \mathbf{x}(t) + \underbrace{\left[\int_t^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}}d\tau \right]}_{=: \mathbf{B}_d} \mathbf{B} \mathbf{u}(t) \quad (1.25)$$

$$= \mathbf{A}_d \mathbf{x}(t) + \mathbf{B}_d \mathbf{u}(t) \quad (1.26)$$

$$(1.27)$$

となる。添え字の d は離散化 (discretization) を意味する。 \mathbf{A}_c が正則行列の場合、

$$\mathbf{B}_d = \left[\int_t^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}}d\tau \right] \mathbf{B} \quad (1.28)$$

$$= \mathbf{A}^{-1} [e^{\Delta t \mathbf{A}} - \mathbf{I}] \mathbf{B} \quad (1.29)$$

が成り立つ。

1.5.2 連続時間モデルから離散時間モデルへの変換

離散化方法 1: 解析解を用いた方法

1 次元の場合 区間 $[t, t + \Delta t]$ において入力 $u(t)$ が一定であると仮定すると,

$$x(t + \Delta t) = \underbrace{e^{a_c \Delta t}}_{=: a_d} x(t) + \underbrace{\left[\int_t^{t+\Delta t} e^{a_c(t+\Delta t-\tau)} d\tau \right]}_{=: b_d} b_c u(t) \quad (1.30)$$

$$= a_d x(t) + b_d u(t) \quad (1.31)$$

$$(1.32)$$

n 次元の場合 区間 $[t, t + \Delta t]$ において入力 $\mathbf{u}(t)$ が一定であると仮定すると,

$$\mathbf{x}(t + \Delta t) = e^{(t+\Delta t)\mathbf{A}_c} \mathbf{x}(0) + \int_0^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}_c} \mathbf{B}_c \mathbf{u}(\tau) d\tau \quad (1.33)$$

$$= e^{\Delta t \mathbf{A}_c} e^{t \mathbf{A}_c} \mathbf{x}(0) + e^{\Delta t \mathbf{A}_c} \int_0^t e^{(t-\tau)\mathbf{A}_c} \mathbf{B}_c \mathbf{u}(\tau) d\tau + \int_t^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}_c} \mathbf{B}_c \mathbf{u}(\tau) d\tau \quad (1.34)$$

$$\approx \underbrace{e^{\Delta t \mathbf{A}_c} \mathbf{x}(t)}_{=: \mathbf{A}_d} + \underbrace{\left[\int_t^{t+\Delta t} e^{(t+\Delta t-\tau)\mathbf{A}_c} d\tau \right]}_{=: \mathbf{B}_d} \mathbf{B}_c \mathbf{u}(t) \quad (1.35)$$

$$= \mathbf{A}_d \mathbf{x}(t) + \mathbf{B}_d \mathbf{u}(t) \quad (1.36)$$

$$(1.37)$$

離散化した場合は

$$\mathbf{x}_{t+1} = \mathbf{A}_c \mathbf{x}_t + \mathbf{B}_c \mathbf{u}_t \quad (1.38)$$

となり, これを状態遷移方程式 (dynamics equations) とも呼ぶ.

離散化方法 2: 微分方程式の数値解法

Euler 法 Euler 法は $\frac{dx}{dt} = f(x, t)$ において, $x_{n+1} = x_t + \Delta t f(x_n, t_n)$ とする手法である.

$$x(t + \Delta t) = x(t) + [a_c x(t) + b_c u(t)] \Delta t \quad (1.39)$$

$$= (1 + a_c \Delta t) x(t) + b_c \Delta t u(t) \quad (1.40)$$

ここで, 解析解を用いる方法と Euler 法の離散化係数の比較をしよう. $\Delta t = 0$ で Taylor 展開により 1 次近似すると $e^{a_c \Delta t} \approx 1 + a_c \Delta t$ となる. $a_c \neq 0$ の場合,

$$\int_t^{t+\Delta t} e^{a_c(t+\Delta t-\tau)} d\tau = \frac{1}{a_c} (e^{a_c \Delta t} - 1) \quad (1.41)$$

$$\approx \frac{1}{a_c} \cdot a_c \Delta t = \Delta t \quad (1.42)$$

Runge-Kutta 法

その他の solver adaptive な方法など、Julia であれば DifferentialEquations.jl などで実装されている solver を用いる方が効率的である。本書では主に Euler 法を用いて実装を行う。Euler 法は精度が低い手法であるという欠点があるものの、実装が簡便で可読性が高いことや、本書で扱うモデルに関しては Euler 法でも定性的に同様の結果が再現できることなどが採用する理由である。ToDo: phase space, fixed points

1.6 線形回帰と最小二乗法

1.6.1 線形回帰

n 個のデータ $(y_1, x_{11}, \dots, x_{1p}), \dots, (y_n, x_{n1}, \dots, x_{np})$ を説明変数 p 個の線形モデル

$$y = \theta_0 + \theta_1 x_1 + \dots + \theta_p x_p + \varepsilon = \theta_0 + \sum_{j=1}^p \theta_j x_j + \varepsilon \quad (1.43)$$

で説明することを考える（説明変数が単一の場合を单回帰、複数の場合を重回帰と呼ぶことがある）。ここで、

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix} \quad (1.44)$$

とすると、線形回帰モデルは $\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \varepsilon$ と書ける。ただし、 \mathbf{X} は計画行列 (design matrix), ε は誤差項である。特に、 ε が平均 0, 分散 σ^2 の独立な正規分布に従う場合、 $\mathbf{y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\theta}, \sigma^2 \mathbf{I})$ と表せる。

1.6.2 最小二乗法によるパラメータの推定

最小二乗法 (ordinary least squares) により線形回帰のパラメータを推定する。 y の予測値は $\mathbf{X}\boldsymbol{\theta}$ なので、誤差 $\delta \in \mathbb{R}^n$ は $\delta = \mathbf{y} - \mathbf{X}\boldsymbol{\theta}$ と表せる。ゆえに目的関数 $L(\boldsymbol{\theta})$ は

$$L(\boldsymbol{\theta}) = \sum_{i=1}^n \delta_i^2 = \|\delta\|^2 = \delta^\top \delta \quad (1.45)$$

となり、 $L(\boldsymbol{\theta})$ を最小化する $\boldsymbol{\theta}$ 、つまり $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} L(\boldsymbol{\theta})$ を求める。

勾配法を用いた推定

最小二乗法による回帰直線を勾配法で求めてみよう。 $\boldsymbol{\theta}$ の更新式は $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{n} \delta \mathbf{X}$ と書ける。ただし、 α は学習率である。

```

using PyPlot, LinearAlgebra, Random
rc("axes.spines", top=false, right=false)

# Ordinary least squares regression
function OLSRegGradientDescent(X, y, initθ; lr=1e-4, num_iters=10000)
    θ = initθ
    for i in 1:num_iters
        y^ = X * θ # predictions
        δ = y - y^ # error
        θ += lr * X' * δ # Update
    end
    return θ
end;

# Generate Toy datas
num_train, num_test = 100, 500 # sample size
dims = 4 # dimensions
Random.seed!(0);

x = rand(num_train) #range(0.1, 0.9, length=num_train)
y = sin.(2π*x) + 0.3randn(num_train);
X = hcat([x .^ p for p in 0:dims-1]...); # design matrix

xtest = range(0, 1, length=num_test)
ytest = sin.(2π*xtest)
Xtest = hcat([xtest .^ p for p in 0:dims-1]...);

# Gradient descent
initθ = zeros(dims) # init variables
θgd = OLSRegGradientDescent(X, y, initθ, lr=1e-2, num_iters=1e5)
y^gd = Xtest * θgd; # predictions

figure(figsize=(5,3.5))
title("Linear Regression with Gradient descent")
scatter(x, y, color="gray", s=10) # samples
plot(xtest, ytest, label="actual") # regression line
plot(xtest, y^gd, label="predicted") # regression line
xlabel("x"); ylabel("y"); legend()
tight_layout()

```

正規方程式を用いた推定

条件に基づいて目的関数 $L(\theta)$ を微分すると次のような方程式が得られる.

$$\mathbf{X}^\top \mathbf{X} \hat{\theta} = \mathbf{X}^\top \mathbf{y} \quad (1.46)$$

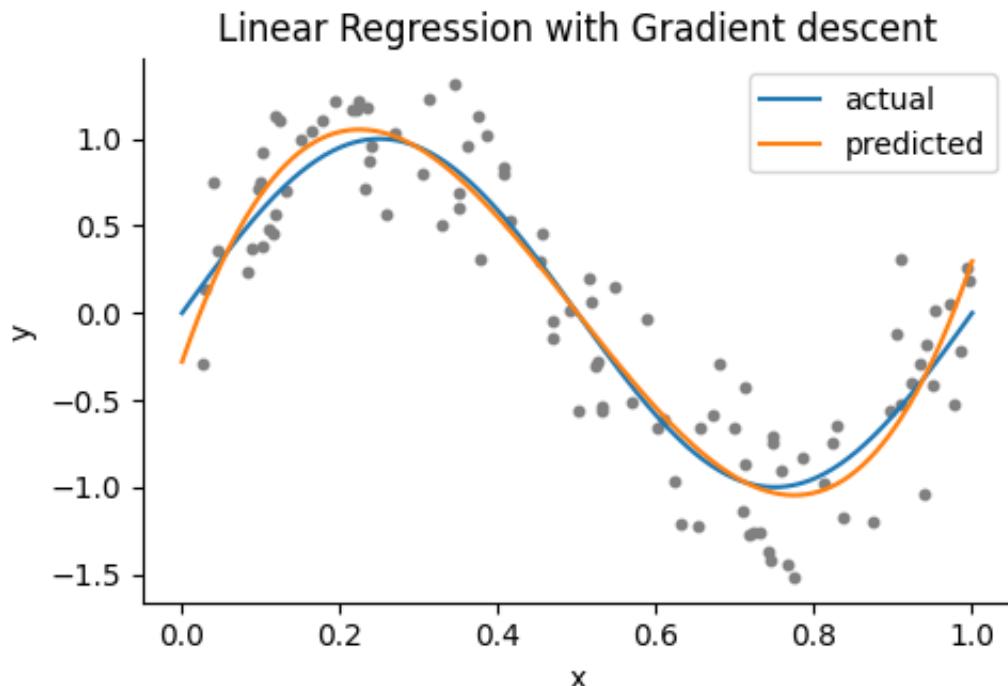


図 1.1 cell007.png

これを正規方程式 (normal equation) と呼ぶ。この正規方程式より、係数の推定値は $\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ という式で得られる。なお、正規方程式自体は $\mathbf{y} = \mathbf{X}\theta$ の左から \mathbf{X}^\top をかける、と覚えると良い。

```
θne = (X' * X) \ X' * y
y^ne = Xtest * θne; # predictions
```

```
figure(figsize=(5,3.5))
title("Linear Regression with Normal equation")
scatter(x, y, color="gray", s=10) # samples
plot(xtest, ytest, label="actual") # regression line
plot(xtest, y^ne, label="predicted") # regression line
xlabel("x"); ylabel("y"); legend()
tight_layout()
```

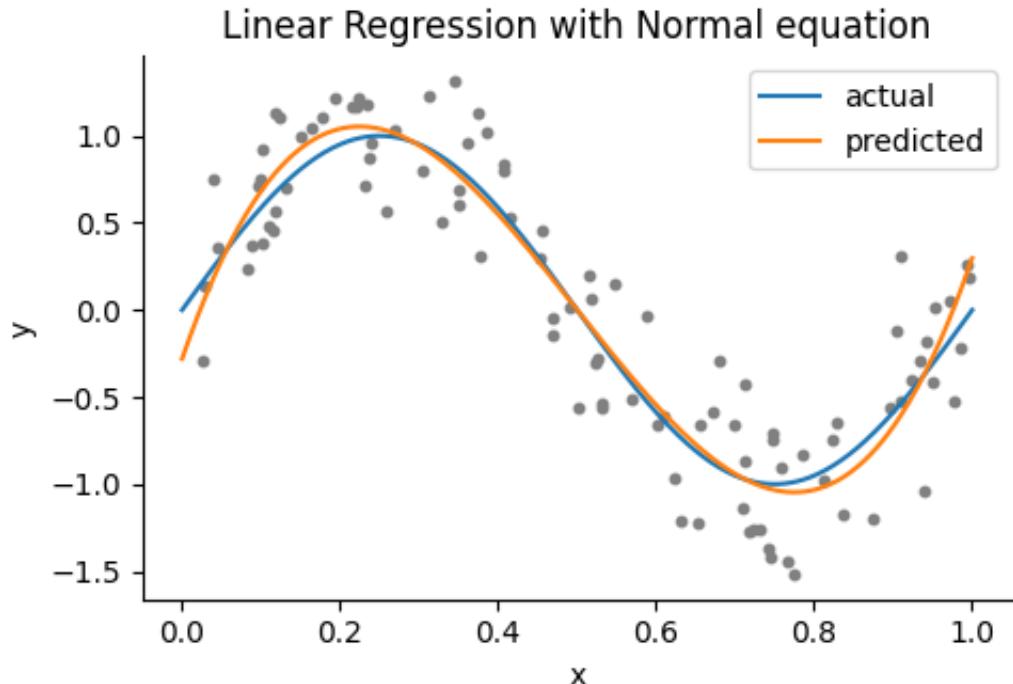


図 1.2 cell010.png

1.7 確率論

1.7.1 期待値 (Expectation)

$$\mathbb{E}_{x \sim p(x)} [f(x)] := \int f(x)p(x)dx \quad (1.47)$$

$x \sim p(x)$ が明示的な場合は $\mathbb{E}_{p(x)} [f(x)]$ や $\mathbb{E} [f(x)]$ と表す.

1.7.2 情報量 (Information)

出現頻度が低い事象は多くの情報量を持つ (Shannon, 1948).

$$\mathbb{I}(x) := \ln \left(\frac{1}{p(x)} \right) = -\ln p(x) \quad (1.48)$$

\mathbf{I} は単位行列なので注意.

1.7.3 平均情報量 (エントロピー, entropy)

$$\mathbb{H}(x) := \mathbb{E}[-\ln p(x)] \quad (1.49)$$

$$\mathbb{H}(x|y) := \mathbb{E}[-\ln p(x|y)] \quad (1.50)$$

1.7.4 Kullback-Leibler 情報量

Kullback-Leibler (KL) divergence (Kullback and Leibler, 1951)

$$D_{\text{KL}}(p(x) \| q(x)) := \int p(x) \ln \frac{p(x)}{q(x)} dx \quad (1.51)$$

$$= \int p(x) \ln p(x) dx - \int p(x) \ln q(x) dx \quad (1.52)$$

$$= \mathbb{E}_{x \sim p(x)}[\ln p(x)] - \mathbb{E}_{x \sim p(x)}[\ln q(x)] \quad (1.53)$$

$$= -\mathbb{H}(x) - \mathbb{E}_{x \sim p(x)}[\ln q(x)] \quad (1.54)$$

1.7.5 相互情報量 (Mutual information)

aaa

1.8 確率過程と確率微分方程式

aaa

第2章

神経細胞のモデル

2.1 神経細胞の形態と生理

2.2 コンダクタンスベースモデル

2.2.1 Hodgkin-Huxley モデル

Hodgkin-Huxley モデル (HH モデル) は、ニューロンの膜興奮を表現する、初めに導出された数理モデルである (Hodgkin and Huxley, 1952)^{*1}. Hodgkin および Huxley はヤリイカの巨大神経軸索に対して電位固定法 (voltage-clamp) を用いた実験を行い、実験から得られた観測結果を元にモデルを構築した (Schwiening, 2012). HH モデルには等価な電気回路モデルがあり、**膜の並列等価回路モデル** (parallel conductance model) と呼ばれている。膜の並列等価回路モデルでは、ニューロンの細胞膜をコンデンサ、細胞膜に埋まっているイオンチャネルを可変抵抗 (動的に変化する抵抗) として置き換える。**イオンチャネル** (ion channel) は特定のイオン (例えばナトリウムイオンやカリウムイオンなど) を選択的に通す膜輸送体の一種である。それぞれのイオンの種類において、異なるイオンチャネルがある (同じイオンでも複数の種類のイオンチャネルがある)。また、イオンチャネルにはイオンの種類に応じて異なる**コンダクタンス** (抵抗の逆数で電流の「流れやすさ」を意味する) と**平衡電位** (equilibrium potential) がある。HH モデルでは、ナトリウム (Na^+) チャネル、カリウム (K^+) チャネル、漏れ電流 (leak current) のイオンチャネルを仮定する。漏れ電流のイオンチャネルは当時特定できなかったチャネルで、膜から電流が漏れ出すチャネルを意味する。なお、現在では漏れ電流の多くは Cl^- イオン (chloride ion) によることが分かっている。それでは、等価回路モデルを用いて電位変化の式を立ててみよう。上図において、 C_m は細胞膜のキャパシタンス (膜容量), $I_m(t)$ は細胞膜を流れる電流 (外部からの入力電流), $I_{\text{Cap}}(t)$ は膜のコンデンサを流れる電流, $I_{\text{Na}}(t)$ 及び $I_K(t)$ はそれぞれナトリウムチャネルとカリウムチャネルを通って膜か

^{*1} Hodgkin および Huxley の論文の図をカラー化して分かりやすくした論文 (Hopper et al., 2022) がある。

ら流出する電流, $I_L(t)$ は漏れ電流である。このとき,

$$I_m(t) = I_{\text{Cap}}(t) + I_{\text{Na}}(t) + I_K(t) + I_L(t) \quad (2.1)$$

という仮定をしている。膜電位を $V(t)$ とすると, Kirchhoff の第二法則 (Kirchhoff's Voltage Law) より,

$$\underbrace{C_m \frac{dV(t)}{dt}}_{I_{\text{Cap}}(t)} = I_m(t) - I_{\text{Na}}(t) - I_K(t) - I_L(t) \quad (2.2)$$

となる。Hodgkin らはチャネル電流 I_{Na}, I_K, I_L が従う式を実験的に求めた。

$$I_{\text{Na}}(t) = \bar{g}_{\text{Na}} \cdot m^3 h(V - E_{\text{Na}}) \quad (2.3)$$

$$I_K(t) = \bar{g}_K \cdot n^4 (V - E_K) \quad (2.4)$$

$$I_L(t) = \bar{g}_L (V - E_L) \quad (2.5)$$

ただし, $\bar{g}_{\text{Na}}, \bar{g}_K$ はそれぞれ Na^+ , K^+ の最大コンダクタンスである (ここで上付き棒は上限値であることを示す)。 \bar{g}_L はオームの法則に従うコンダクタンスで, L コンダクタンスは時間的に変化はしないと仮定する。また, m は Na^+ コンダクタンスの活性化パラメータ, h は Na^+ コンダクタンスの不活性化パラメータ, n は K^+ コンダクタンスの活性化パラメータであり, ゲートの開閉確率を表している。よって, HH モデルの状態は V, m, h, n の 4 変数で表される。これらの変数は以下の x を m, n, h に置き換えた 3 つの微分方程式に従う。

$$\frac{dx}{dt} = \alpha_x(V)(1-x) - \beta_x(V)x \quad (2.6)$$

ただし, V の関数である $\alpha_x(V), \beta_x(V)$ は m, h, n によって異なり, 次の 6 つの式に従う。

$$\begin{aligned} \alpha_m(V) &= \frac{0.1(V+40)}{1-\exp(-0.1(V+40))}, & \beta_m(V) &= 4 \exp(-(V+65)/18) \\ \alpha_h(V) &= 0.07 \exp(-0.05(V+65)), & \beta_h(V) &= 1/(1+\exp(-0.1(V+35))) \\ \alpha_n(V) &= \frac{0.01(V+55)}{1-\exp(-0.1(V+55))}, & \beta_n(V) &= 0.125 \exp(-0.0125(V+65)) \end{aligned} \quad (2.7)$$

これまでに説明した式を用いて HH モデルを実装する。まず必要なパッケージを読み込む。

```
using Parameters: @unpack # or using UnPack
using PyPlot
rc("axes.spines", top=false, right=false)
```

```
abstract type Layer end
abstract type Neuron <: Layer end
abstract type SpikeNeuron <: Neuron end

abstract type Synapse <: Layer end
```

変更しない定数を保持する **struct** の **HHParameter** と、変数を保持する **mutable struct** の **HH**を作成する。定数は次のように設定する。

$$C_m = 1.0 \text{ } \mu\text{F}/\text{cm}^2, \bar{g}_{\text{Na}} = 120 \text{ mS}/\text{cm}^2, \bar{g}_{\text{K}} = 36 \text{ mS}/\text{cm}^2, \bar{g}_{\text{L}} = 0.3 \text{ mS}/\text{cm}^2 \\ E_{\text{Na}} = 50.0 \text{ mV}, E_{\text{K}} = -77 \text{ mV}, E_{\text{L}} = -54.387 \text{ mV}$$
(2.8)

```

@kwdef struct HHParameter{FT}
    Cm::FT = 1 # 膜容量(μF/cm^2)
    gNa::FT = 120; gK::FT = 36; gL::FT = 0.3 # Na+, K+, leakの最大コンダクタンス(mS/cm^2)
    ENa::FT = 50; EK::FT = -77; EL::FT = -54 # Na+, K+, leakの平衡電位(mV)
end

@kwdef mutable struct HH{FT} <: SpikeNeuron
    num_neurons::UInt16
    dt::FT = 1e-3
    param::HHParameter = HHParameter{FT}()
    v::Vector{FT} = fill(-65, num_neurons)
    m::Vector{FT} = fill(0.05, num_neurons)
    h::Vector{FT} = fill(0.6, num_neurons)
    n::Vector{FT} = fill(0.32, num_neurons)
end

function update!(neuron::HH, Iext::Vector)
    @unpack num_neurons, dt, v, m, h, n = neuron
    @unpack Cm, gNa, gK, gL, ENa, EK, EL = neuron.param
    @inbounds for i = 1:num_neurons
        αm = 0.1(v[i]+40)/(1 - exp(-0.1(v[i]+40)))
        βm = 4exp(-(v[i]+65)/18)
        αh = 0.07exp(-0.05*(v[i]+65))
        βh = 1/(1 + exp(-0.1*(v[i]+35)))
        αn = 0.01(v[i]+55)/(1 - exp(-0.1(v[i]+55)))
        βn = 0.125exp(-0.0125(v[i]+65))

        m[i] += dt * (αm *(1 - m[i]) - βm * m[i])
        h[i] += dt * (αh *(1 - h[i]) - βh * h[i])
        n[i] += dt * (αn *(1 - n[i]) - βn * n[i])

        INa = gNa * m[i]^3 * h[i] * (v[i] - ENa)
        IK = gK * n[i]^4 * (v[i] - EK)
        IL = gL * (v[i] - EL)

        v[i] += dt / Cm * (Iext[i] - INa - IK - IL)
    end
    return v
end

(layer)::Layer)(x) = update!(layer, x)

```

次に変数を更新する関数 **update!**を書く。ソルバーとしては陽的 Euler 法または 4 次の Runge-Kutta 法を用いる。以下では Euler 法を用いている。Julia では for ループを用いて 1 つのニューロンごとにパ

ラメータを更新する方がベクトルを用いるよりも高速である。

Hodgkin-Huxley モデルのシミュレーションの実行

いくつかの定数を設定してシミュレーションを実行する。

```
T = 450 # ms
dt = 0.01 # ms
nt = Int(T/dt) # number of timesteps
num_neurons = 1 # number of neurons
time = (1:nt)*dt # time array
Ie = repeat(10 * ((time .> 50) - (time .> 200)) + 35 * ((time .> 250) - (time .> 400)), 1, num_neurons) # injection current
varr, gatearr = zeros(nt, num_neurons), zeros(nt, 3, num_neurons) # 記録用
hh_neurons = HH{Float32}(num_neurons=num_neurons, dt=dt) # model の定義

# simulation
@time for t = 1:nt
    v = hh_neurons(Ie[t, :])
    varr[t, :] = v
    gatearr[t, :, :] .= [hh_neurons.m; hh_neurons.h; hh_neurons.n]
end
```

ニューロンの膜電位 v , ゲート変数 m , h , n , 刺激電流 I_e の描画をする。

```
fig, axes = subplots(3,1, figsize=(5, 3), height_ratios=[1, 1, 0.5], constrained_layout=true)
axes[1].set_title("Hodgkin-Huxley model")
axes[1].plot(time, varr[:, 1], color="black"); axes[1].set_ylabel("V (mV)")

labellist=["m" "h" "n"]
for i in 1:3
    axes[2].plot(time, gatearr[:, i, 1], label=labellist[i])
end;
axes[2].set_ylabel("Gating Value"); axes[2].legend(loc="center left", bbox_to_anchor=(1, 0.5))
axes[3].plot(time, Ie[:, 1], color="black");
axes[3].set_ylabel("Current\n*L\"(\$\mu$A/cm$^2$)"); axes[3].set_xlabel("Times (ms)")
```

次項で用いるために発火回数を求める。ここでは膜電位が 0 を超えた点を数えることで、簡易的に求める。

```
function get_num_spikes(varr)
    spike = (varr[1:end-1, :] .< 0) .& (varr[2:end, :] .> 0)
    return sum(spike, dims=1)
end

num_spikes = get_num_spikes(varr)
println("Num. of spikes : ", num_spikes[1])
```

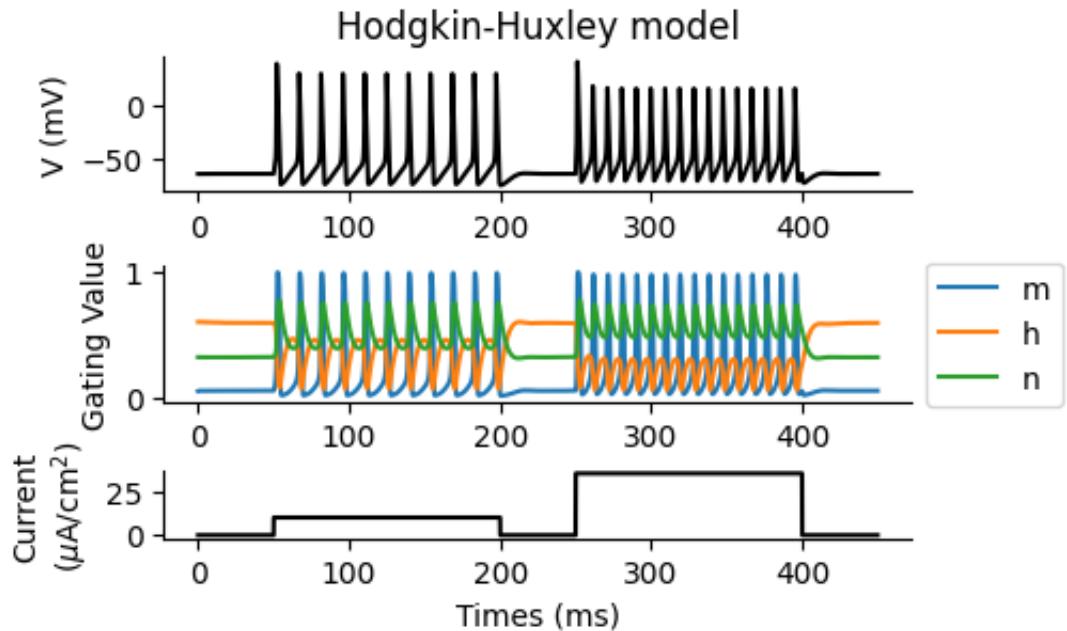


図 2.1 cell011.png

50ms から 200ms までで 11 回, 250ms から 400ms までで 16 回発火しているので発火回数は計 27 回であり, この結果は正しい.

2.2.2 Connor-Stevens モデル

HH モデルはイカの巨大軸索の活動を再現したものであるが, 脊椎動物のニューロンの神経活動を再現するために HH モデルを修正したモデル (modified Hodgkin-Huxley model) が提案されてきた. その一種である, **Connor-Stevens モデル** は HH モデルに 2 つ目のカリウム電流 (A 型カリウム電流) を追加し, 低い発火率でも活動を維持できる (振動を維持できる) ようにしたものである (Connor and Stevens, 1971; Connor et al., 1977). ここでパラメータは (Dayan and Abbott, 2005) に記載のものを使用する.

$$\begin{aligned} C_m &= 1.0 \text{ } \mu\text{F}/\text{cm}^2, \\ \bar{g}_{\text{Na}} &= 120 \text{ } \text{mS}/\text{cm}^2, \bar{g}_{\text{K}} = 20 \text{ } \text{mS}/\text{cm}^2, \bar{g}_{\text{A}} = 47.7 \text{ } \text{mS}/\text{cm}^2, \bar{g}_{\text{L}} = 0.3 \text{ } \text{mS}/\text{cm}^2 \\ E_{\text{Na}} &= 55.0 \text{ mV}, E_{\text{K}} = -72 \text{ mV}, E_{\text{A}} = -75 \text{ mV}, E_{\text{L}} = -17 \text{ mV} \end{aligned} \quad (2.9)$$

$$\begin{aligned} \alpha_m &= \frac{0.38(V + 29.7)}{1 - \exp(-0.1(V + 29.7))} & \beta_m &= 15.2 \exp(-(V + 54.7)/18) \\ \alpha_h &= 0.266 \exp(-0.05(V + 48)) & \beta_h &= 3.8/(1 + \exp(-0.1(V + 18))) \\ \alpha_n &= \frac{0.02(V + 45.7)}{1 - \exp(-0.1(V + 45.7))} & \beta_n &= 0.25 \exp(-0.0125(V + 55.7)) \end{aligned} \quad (2.10)$$

$$\frac{dx}{dt} = \frac{x_\infty - x}{\tau_x} \quad (x = a, b) \quad (2.11)$$

$$\begin{aligned} a_\infty &= \left(\frac{0.0761 \exp[(V + 94.22)/31.84]}{1 + \exp((V + 1.17)/28.93)} \right)^{\frac{1}{3}} \\ \tau_a &= 0.3632 + 1.158/(1 + \exp((V + 55.96)/20.12)) \\ b_\infty &= [1 + \exp((V + 53.3)/14.54)]^{-4} \\ \tau_b &= 1.24 + 2.678/(1 + \exp[(V + 50)/16.027]) \end{aligned} \quad (2.12)$$

```
@kwdef struct CSParameter{FT}
    Cm::FT = 1 # 膜容量(uF/cm^2)
    gNa::FT = 120; gK::FT = 20; gA::FT = 47.7; gL::FT = 0.3 # Na+, K+, KA, +
        leakの最大コンダクタンス(mS/cm^2)
    ENa::FT = 55; EK::FT = -72; EA::FT = -75; EL::FT = -17 # Na+, K+, KA, +
        leakの平衡電位(mV)
end
```

```
@kwdef mutable struct CS{FT} <: SpikeNeuron
    num_neurons::UInt16
    dt::FT = 1e-3
    param::CSParameter = CSParameter{FT}()
    v::Vector{FT} = fill(-65, num_neurons)
    m::Vector{FT} = fill(0.05, num_neurons)
    h::Vector{FT} = fill(0.6, num_neurons)
    n::Vector{FT} = fill(0.32, num_neurons)
    a::Vector{FT} = fill(0.66, num_neurons)
    b::Vector{FT} = fill(0.22, num_neurons)
end
```

```
function update!(neuron::CS, Iext::Vector)
    @unpack num_neurons, dt, v, m, h, n, a, b = neuron
    @unpack Cm, gNa, gK, gA, gL, ENa, EK, EA, EL = neuron.param
    @inbounds for i = 1:num_neurons
        αm = 0.38(v[i]+29.7)/(1 - exp(-0.1*(v[i]+29.7)))
        βm = 15.2exp(-(v[i]+54.7)/18)
        αh = 0.266exp(-0.05*(v[i]+48))
        βh = 3.8/(1 + exp(-0.1*(v[i]+18)))
        αn = 0.02(v[i]+45.7)/(1 - exp(-0.1*(v[i]+45.7)))
        βn = 0.25exp(-0.0125(v[i]+55.7))

        a∞ = ((0.0761exp((v[i]+94.22)/31.84))/(1+exp((v[i]+1.17)/28.93)))^(1/3)
        τa = 0.3632+1.158/(1+exp((v[i]+55.96)/20.12))
        b∞ = (1+exp((v[i]+53.3)/14.54))^-4
        τb = 1.24+2.678/(1+exp((v[i]+50)/16.027))

        m[i] += dt * (αm * (1 - m[i]) - βm * m[i])
        h[i] += dt * (αh * (1 - h[i]) - βh * h[i])
        n[i] += dt * (αn * (1 - n[i]) - βn * n[i])
        a[i] += dt * (a∞ - a[i]) / τa
        b[i] += dt * (b∞ - b[i]) / τb

        INa = gNa * m[i]^3 * h[i] * (v[i] - ENa)
```

```

IK = gK * n[i]^4 * (v[i] - EK)
IA = gA * a[i]^3 * b[i] * (v[i] - EA)
IL = gL * (v[i] - EL)

v[i] += dt / Cm * (Iext[i] - INa - IK - IA - IL)
end
return v
end

```

```

T = 450 # ms
dt = 0.01 # ms
nt = Int(T/dt) # number of timesteps
num_neurons = 1 # number of neurons
time = (1:nt)*dt # time array
Iext_cs = repeat(10 * ((time .> 50) - (time .> 200)) + 20 * ((time .> 250) - (time .> 400)), 1, num_neurons) # injection current
varr_cs = zeros(nt, num_neurons) # 記録用

cs_neurons = CS{Float32}(num_neurons=num_neurons, dt=dt) # model の定義

# simulation
@time for t = 1:nt
    v = cs_neurons(Iext_cs[t, :])
    varr_cs[t, :] = v
end

```

```

fig, axes = subplots(2,1, figsize=(4, 2), height_ratios=[1, 0.5], -
                     constrained_layout=true)
axes[1].set_title("Connor-Stevens model")
axes[1].plot(time, varr_cs[:, 1], color="black"); axes[1].set_ylabel("V (mV)")
axes[2].plot(time, Iext_cs[:, 1], color="black");
axes[2].set_ylabel("Current\n" * L"($\mu$A/cm$^2$)"); axes[2].set_xlabel("Times (ms)")

```

2.2.3 F-I 曲線

HH モデルにおいて、入力電流に対する発火率がどのように変化するかを調べる。次のコードのように入力電流を徐々に増加させたときの発火率を見てみよう。

```

function fi_curve(NeuronType; num_neurons=200, T=1000, dt=0.04,
                   current_range = [1, 30])
    nt = Int(T/dt) # number of timesteps
    Iext_range = Array{Float32}(range(current_range..., length=num_neurons)) # -
                           injection current
    neurons = NeuronType{Float32}(num_neurons=num_neurons, dt=dt) # model の定義
    varr_fi = zeros(Float32, nt, num_neurons) # 記録用

    # simulation
    for t = 1:nt

```

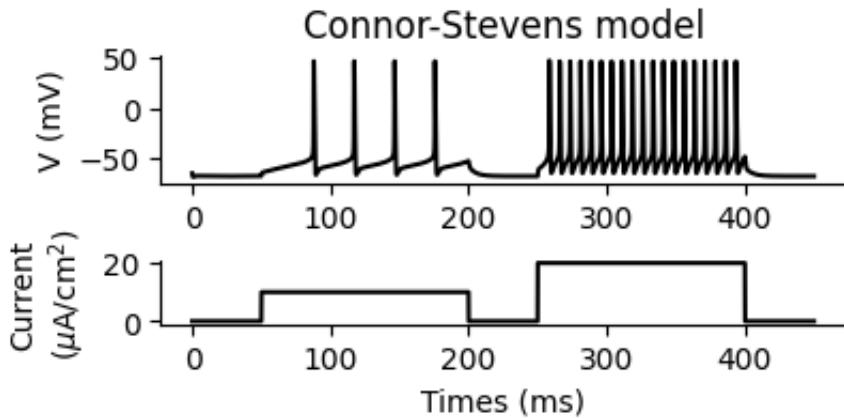


図 2.2 cell019.png

```

v = neurons(Iext_range)
varr_fi[t, :] = v
end
num_spikes = get_num_spikes(varr_fi)
rate = num_spikes/T*1e3;
threshold = Iext_range[findfirst(rate .> 1)[2]]
return Iext_range, rate, threshold
end

```

```

Iext_range_hh, rate_hh, threshold_hh = fi_curve(HH, current_range = [1, 25])
Iext_range_cs, rate_cs, threshold_cs = fi_curve(CS, current_range = [1, 25]);

```

発火率を計算して結果を描画する。

```

fig, axes = subplots(1,2, figsize=(4, 2), constrained_layout=true)
axes[1].set_title("Type I (CS model)")
axes[1].text(threshold_cs+1, 0, L"$I_{\theta}=$"+string(round(threshold_cs, digits=2)))
axes[1].plot(Iext_range_cs[:, rate_cs[1, :]]);
axes[1].set_ylabel("Firing rate (Hz)")

axes[2].set_title("Type II (HH model)")
axes[2].text(threshold_hh+1, 0, L"$I_{\theta}=$"+string(round(threshold_hh, digits=2)))
axes[2].plot(Iext_range_hh[:, rate_hh[1, :]]);
fig.supxlabel(L"Input current ($\mu$A/cm$^2$)", size=10)

```

このような曲線を **frequency-current (F-I) 曲線** (または neuronal input/output (I/O) 曲線) と呼ぶ。 I_θ は閾値電流を意味する (ここでは発火率が 1Hz 以上になる点を閾値と設定している)。F-I 曲線の

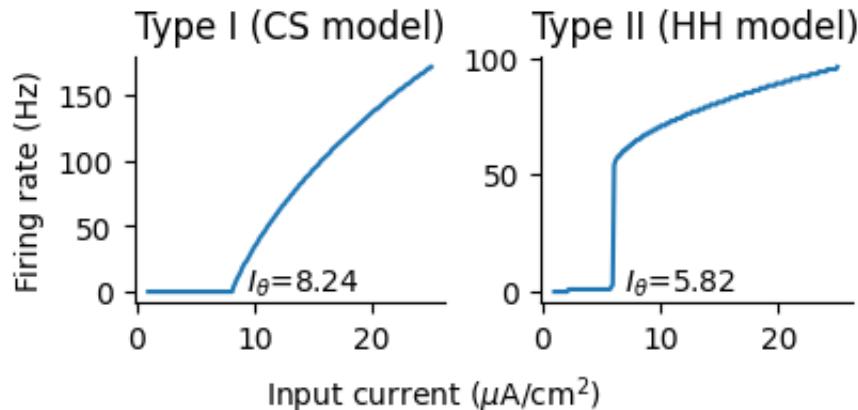


図 2.3 cell024.png

種類に応じて Type I および II に分けられる^{*2}.

2.2.4 全か無かの法則の反例

ニューロンは電流が流入することで膜電位が変化し、膜電位がある一定の閾値を超えると活動電位が発生する、というのはニューロンの活動電位発生についての典型的な説明である。膜電位が閾値を超えるか超えないかで活動電位の発生が決まるという法則を、全か無かの法則 (all-or-none principle) と呼ぶ。後に説明する LIF モデルなどは、全か無かの法則に従って神経活動のモデル化を行っている。しかし、この全か無かの法則の法則は必ずしも成立するわけではない。反例として抑制後リバウンド (Postinhibitory rebound; PIR) という現象がある。抑制後リバウンドは過分極性の電流の印加を止めた際に膜電位が静止膜電位に回復するのみならず、さらに脱分極をして発火をするという現象である。この時生じる発火をリバウンド発火 (rebound spikes) と呼ぶ。この現象が生じる要因としてアノーダルブレイク (anodal break, または anode break excitation; ABE) や、遅い T 型カルシウム電流 (slow T-type calcium current) が考えられている (Chik et al., 2004)。HH モデルはこのうちアノーダルブレイクを再現できるため、シミュレーションによりどのような現象か確認してみよう。これは入力電流を変更するだけで行える。

```
T = 450 # ms
dt = 0.01 # ms
nt = Int(T/dt) # number of timesteps
num_neurons = 1 # number of neurons
time = (1:nt)*dt # time array
```

^{*2} Type III ニューロンも存在する

```
Ie = repeat(10 * (-(time .> 50) + (time .> 200)) + 20* (-(time .> 250) + (time .> 400)), 1, num_neurons) # injection current
varr, gatearr = zeros(nt, num_neurons), zeros(nt, 3, num_neurons) # 記録用

hh_neurons = HH{Float32}(num_neurons=num_neurons, dt=dt) # modelの定義

# simulation
@time for t = 1:nt
    v = hh_neurons(Ie[t, :])
    varr[t, :] = v
    gatearr[t, :, :] .= [hh_neurons.m; hh_neurons.h; hh_neurons.n]
end
```

結果は次のようになる。

```
fig, axes = subplots(3,1, figsize=(5, 3), height_ratios=[1, 1, 0.5], constrained_layout=true)
axes[1].set_title("Postinhibitory rebound")
axes[1].plot(time, varr[:, 1], color="black"); axes[1].set_ylabel("V (mV)")

labellist=["m" "h" "n"]
for i in 1:3
    axes[2].plot(time, gatearr[:, i, 1], label=labellist[i])
end;
axes[2].set_ylabel("Gating Value"); axes[2].legend(loc="center left", bbox_to_anchor=(1, 0.5))
axes[3].plot(time, Ie[:, 1], color="black");
axes[3].set_ylabel("Current\n*L\"$(\mu A/cm^2)$"); axes[3].set_xlabel("Times (ms)")
```

なぜこのようなことが起こるか、というと過分極の状態から静止膜電位へと戻る際に Na^+ チャネルが活性化 (Na^+ チャネルの活性化パラメータ m が増加し、不活性化パラメータ h が減少) し、膜電位が脱分極することで再度 Na^+ チャネルが活性化する、というポジティブフィードバック過程（自己再生的過程）に突入するためである（もちろん、この過程は通常の活動電位発生のメカニズムである）。この際、発火に必要な閾値が膜電位の低下に応じて下がった、ということもできる。なお、PIR に関する現象として抑制後促通 (Postinhibitory facilitation; PIF) がある。これは抑制入力の後に興奮入力がある一定の時間内で入ると発火が起こるという現象である (Dodla and Rinzel, 2006)。

2.3 FitzHugh-Nagumo モデル

2.3.1 FitzHugh-Nagumo モデルの定義

前節では神経活動のダイナミクスを微分方程式で表した Hodgkin-Huxley(HH) モデルを扱った。HH モデルの特徴は、4 変数で構成され、各変数が膜電位および Na チャネルや K チャネルなどの活性/不活性状態を意味することである。この HH モデルをより簡易化し、2 変数で神経活動の興奮とその伝播を表そうと提案されたのが **FitzHugh-Nagumo (FHN) モデル** である。FHN モデルは van der Pol 振動

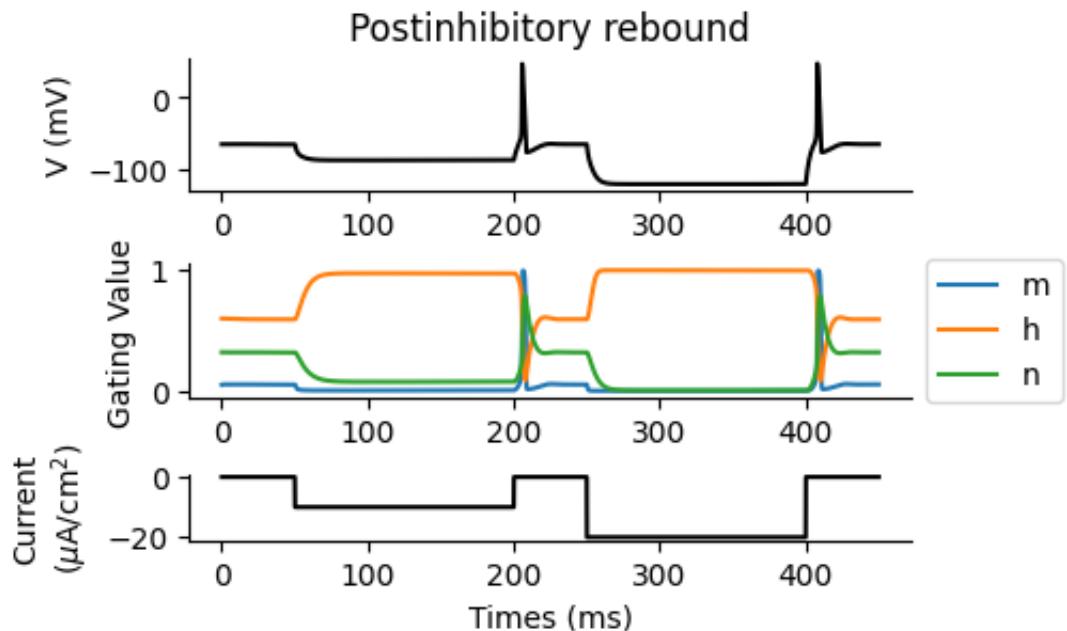


図 2.4 cell029.png

子を FitzHugh が修正し (FitzHugh, 1955) (Fitzhugh, 1961), 南雲らによりトンネル (江崎) ダイオードを用いて電子回路上に実装^{*3}された (Nagumo et al., 1962) という経緯がある。FHN モデルは以下で表される。

$$\frac{dv}{dt} = c \left(v - \frac{v^3}{3} - u + I_e \right) \quad (2.13)$$

$$\frac{du}{dt} = v - bu + a \quad (2.14)$$

v は膜電位で、 u は回復変数 (recovery variable) と呼ばれる。FitzHugh により、HH モデルにおける (V, m) および (n, h) がそれぞれ FHN モデルの v および u に対応すると説明されている (Fitzhugh, 1961)^{*4}。 a, b, c は定数であり、 $a = 0.7, b = 0.8, c = 10$ がよく使われる。 I_e は外部刺激電流に対応する。

```
using Parameters: @unpack # or using UnPack
using PyPlot
rc("axes.spines", top=false, right=false)
```

*3 神経活動を再現する電子回路をニューリスター (neuristor) という。

*4 HH モデルにおける V と m は強い正の相関があり、 n と h は強い負の相関があるため、それぞれの変数の組は 1 つの変数に縮約されうる。

変更しない定数を保持する `struct` の `FHNParameter` と、変数を保持する `mutable struct` の `FHN` を作成する。

```
abstract type Layer end
abstract type Neuron <: Layer end

@kwdef struct FHNParameter{FT}
    a::FT = 0.7; b::FT = 0.8; c::FT = 10.0
end

@kwdef mutable struct FHN{FT} <:Neuron
    num_neurons::UInt16
    dt::FT = 1e-2
    param::FHNParameter = FHNParameter{FT}()
    v::Vector{FT} = fill(-1.0, num_neurons)
    u::Vector{FT} = zeros(num_neurons)
end
```

次に変数を更新する関数 `update!`を書く。ソルバーとしては陽的 Euler 法または 4 次の Runge-Kutta 法を用いる。以下では Euler 法を用いている。Julia では for ループを用いて 1 つのニューロンごとにパラメータを更新する方がベクトルを用いるよりも高速である。

```
function update!(neuron::FHN, x::Vector)
    @unpack num_neurons, dt, v, u = neuron
    @unpack a, b, c = neuron.param
    @inbounds for i = 1:num_neurons
        v[i] += dt * c * (-u[i] + v[i] - v[i]^3 / 3 + x[i])
        u[i] += dt * (v[i] - b*u[i] + a)
    end
    return v
end

(layer)::Layer)(x) = update!(layer, x)
```

2.3.2 FitzHugh-Nagumo モデルのシミュレーションの実行

いくつかの定数を設定してシミュレーションを実行する。

```
T = 100 # ms
dt = 0.01 # ms
nt = UInt32(T/dt) # number of timesteps
num_neurons = 1 # ニューロンの数

# 入力刺激
time = (1:nt)*dt
Ie = repeat(0.5 * ((time .> 10) - (time .> 45)) + 0.34 * ((time .> 55) - (time .> 90)), 1, num_neurons) # injection current

# 記録用
```

```

varr, uarr = zeros(Float32, nt, num_neurons), zeros(Float32, nt, num_neurons)

# modelの定義
fhn_neurons = FHN{Float32}(num_neurons=num_neurons, dt=dt)

# simulation
@time for t = 1:nt
    v = fhn_neurons(Ie[t, :])
    varr[t, :], uarr[t, :] = v, fhn_neurons.u
end

```

結果を描画する。

```

figure(figsize=(5,4))
subplot(3, 1, 1); plot(time, varr[:, 1], label=false, color="black"); ylabel("v")
subplot(3, 1, 2); plot(time, uarr[:, 1], label=false); ylabel("u");
subplot(3, 1, 3); plot(time, Ie, label=false); ylabel("Current"); xlabel("Times (ms)")
tight_layout()

```

2.3.3 相図の描画

phase plot

```

margin = 1.0
vmax, vmin = maximum(varr) + margin, minimum(varr) - margin
umax, umin = maximum(uarr) + margin, minimum(uarr) - margin
vrange, urange = vmin:0.1:vmax, umin:0.1:umax
U = [i for i in urange, j in 1:length(vrange)]
V = [j for i in 1:length(urange), j in vrange]

a, b, c, Ie = 0.7, 0.8, 10.0, 0.34
dV = c * (-U + V - V.^3 / 3 .+ Ie)
dU = V - b*U .+ a;

```

```

figure(figsize=(4,3))
streamplot(V, U, dV, dU, density=[0.8, 0.8], linewidth=2)
contour(V, U, dU, levels=[0])
contour(V, U, dV, levels=[0])
    plot(varr, uarr); xlim(vmin, vmax); ylim(umin, umax); xlabel(L"\$v\$");
    ylabel(L"\$u\$")
tight_layout()

```

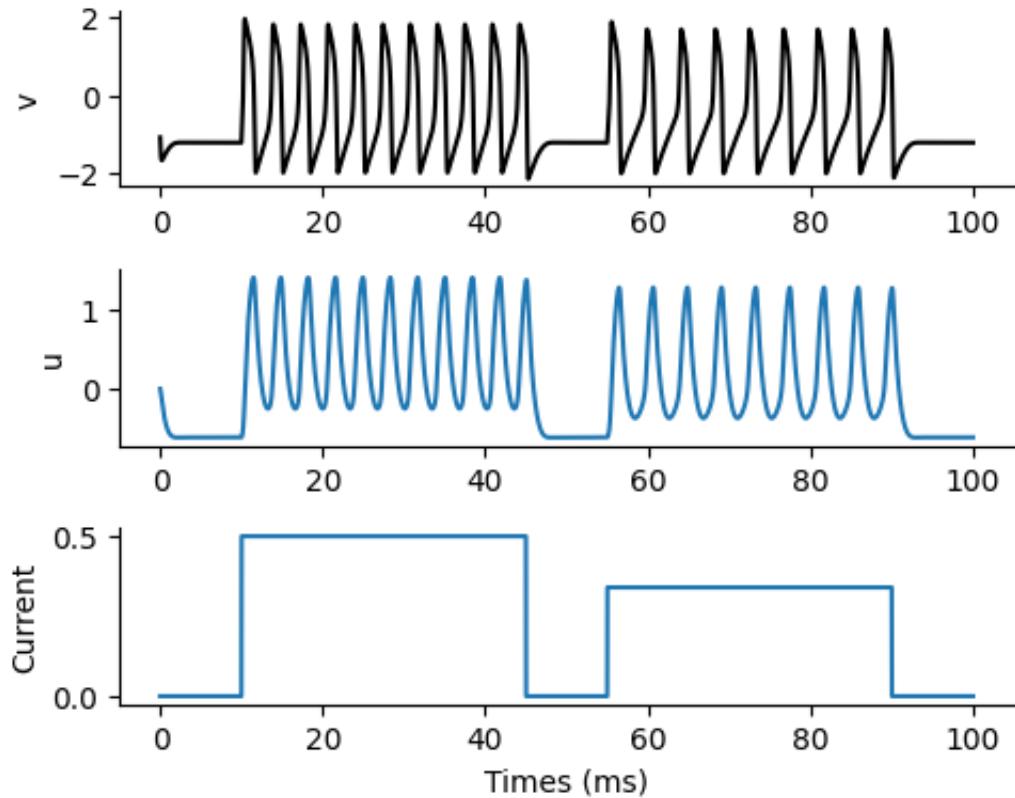


図 2.5 cell009.png

2.4 積分発火モデル

2.4.1 Leaky integrate-and-fire モデル

生理学的なイオンチャネルの挙動は考慮せず、入力電流を膜電位が閾値に達するまで時間的に積分するというモデルを **Integrate-and-fire (IF, 積分発火) モデル** という。さらに、IF モデルにおいて膜電位の漏れ (leak)^{*5} も考慮したモデルを **Leaky integrate-and-fire (LIF, 漏れ積分発火) モデル** と呼ぶ。ここでは LIF モデルのみを取り扱う。ニューロンの膜電位を $V_m(t)$ 、静止膜電位を V_{rest} 、入力電流^{*6}を

^{*5} この漏れはイオンの拡散などによるもの。

^{*6} シナプス入力による電流がどうなるかは、第三章「シナプス伝達のモデル」で扱う。

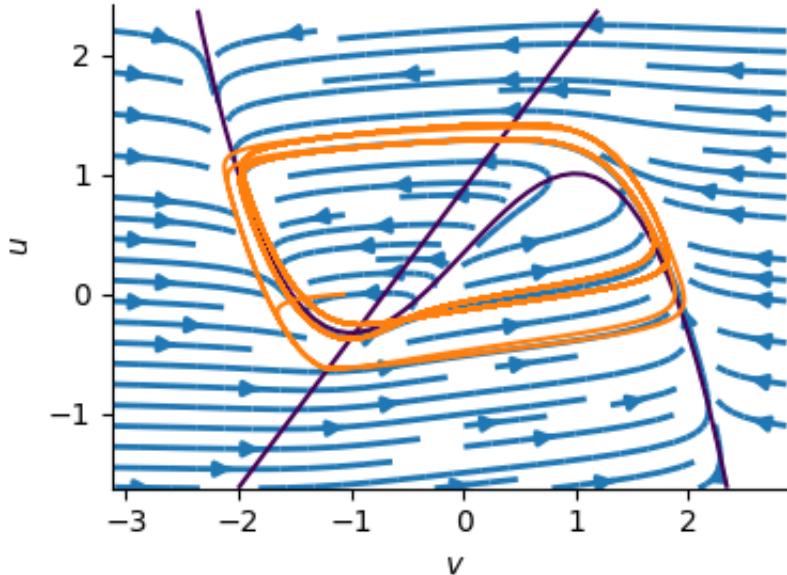


図 2.6 cell012.png

$I(t)$, 膜抵抗を R_m , 膜電位の時定数を τ_m ($= R_m \cdot C_m$) とすると, 式は次のようにになる^{*7}.

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\text{rest}}) + R_m I(t) \quad (2.15)$$

ここで, V_m が閾値 (threshold)^{*8} V_{th} を超えると, 脱分極が起こり, 膜電位はピーク電位 V_{peak} まで上昇する. 発火後は再分極が起こり, 膜電位はリセット電位 V_{reset} まで低下すると仮定する^{*9}. 発火後, 一定の期間 τ_{ref} の間は膜電位が変化しない^{*10} とする. これを 不応期 (refractory time period) と呼ぶ. 以上を踏まえて LIF モデルを実装してみよう. まず必要なパッケージを読み込む.

```
using Base: @kwdef
using Parameters: @unpack # or using UnPack
using PyPlot
rc("axes.spines", top=false, right=false)
```

HH モデルと同様に変更しない定数を保持する **struct** の **LIFParameter** と, 変数を保持する **mutable struct** の **LIF** を作成する.

*7 $(V_m(t) - V_{\text{rest}})$ の部分は膜電位の基準を静止膜電位としたことにして, 単に $V_m(t)$ だけの場合もある. また, 右辺の $R I(t)$ の部分は単に $I(t)$ とされることもある. 同じ表記だが, この場合の $I(t)$ はシナプス電流に比例する量, となっている (単位は mV).

*8 th から始まるので文字 θ が使われることもある.

*9 リセット電位は静止膜電位と同じ場合もあれば, 過分極を考慮して静止膜電位より低めに設定することもある.

*10 実装によっては不応期の間は膜電位の変化は許容するが発火は生じないようにすることもある.

```

@kwdef struct LIFParameter{FT}
    tref::FT = 2; tc_m::FT = 10 # 不応期, 膜時定数 (ms)
    vrest::FT = -60; vreset::FT = -65; vthr::FT = -40; vpeak::FT = 30 # 静止膜電位,
    リセット電位, 閾値電位, ピーク電位 (mV)
end

@kwdef mutable struct LIF{FT}
    param::LIFParameter = LIFParameter{FT}()
    N::UInt32 # ニューロンの数
    v::Vector{FT} = fill(-65.0, N); v_-::Vector{FT} = fill(-65.0, N) # 膜電位,
    発火電位も記録する膜電位 (mV)
    fire::Vector{Bool} = zeros(Bool, N) # 発火
    tlast::Vector{FT} = zeros(N) # 最後の発火時刻 (ms)
    tcount::FT = 0 # 時間カウント
end

```

次に変数を更新する関数 `update!`を書く.

```

function update!(variable::LIF, param::LIFParameter, Ie::Vector, dt)
    @unpack N, v, v_-, fire, tlast, tcount = variable
    @unpack tref, tc_m, vrest, vreset, vthr, vpeak = param

    @inbounds for i = 1:N
        #v[i] += dt * ((vrest - v[i] + Ie[i]) / tc_m) # 不応期を考慮しない場合の更新式
        v[i] += dt * ((dt*tcount) > (tlast[i] + tref)) * ((vrest - v[i] + Ie[i]) / tc_m)
        #v[i] += dt * ifelse(dt*tcount[1] > tlast[i] + tref, (vrest - v[i] + Ie[i]) /
        #/ tc_m, 0)
    end
    @inbounds for i = 1:N
        fire[i] = v[i] >= vthr
        v_-[i] = ifelse(fire[i], vpeak, v[i]) # 発火時の電位も含めて記録するための変数
        # (除いてもよい)
        v[i] = ifelse(fire[i], vreset, v[i])
        tlast[i] = ifelse(fire[i], dt*tcount, tlast[i]) # 発火時刻の更新
    end
end

```

いくつかの処理について解説しておく. まず、一番目の for ループ内の `v[i]` の `((dt*tcount) > (tlast[i] + tref))` は最後にニューロンが発火した時刻 `tlast[i]` に不応期 `tref` を足した時刻よりも現在の時刻 `dt*tcount[1]` が大きければ膜電位の更新を許可し、小さければ更新しない. 二番目の for ループにおける `fire[i]` はニューロンの膜電位が閾値電位 `vthr` を超えたなら `True` となる. `v[i]` などの更新式にある `ifelse(a, b, c)` は `a` が `True` の時は `b` を返し、`False` の時は `c` を返す関数であり、`v[i] = ifelse(fire[i], vreset, v[i])` は `fire[i]` が `True` なら `v[i]` をリセット電位 `vreset` とし、そうでなければそのままの値を返すという処理である. 同様にして `tlast[i]` は発火したときにその時刻を記録する変数となっている. なお、`v_-[i] = ifelse(fire[i], vpeak, v[i])` は実際のシミュレーションにおいて意味をなさない. 単に発火時の電位 `vpeak` を含めて記録すると描画時の見栄えが良いというだけである. これらの `struct` と関

数を用いてシミュレーションを実行する。I は HH モデルのときと同じように矩形波を入力する。実は I は入力電流ではなく入力電流に比例する量となっているが、これは膜抵抗を乗じた後の値であると考えるとよい。

2.4.2 LIF モデルのシミュレーションの実行

いくつかの定数を設定してシミュレーションを実行する。

```
T = 450 # ms
dt = 0.01f0 # ms
nt = UInt32(T/dt) # number of timesteps
N = 1 # ニューロンの数

# 入力刺激
t = Array{Float32}(1:nt)*dt
Ie = repeat(25f0 * ((t .> 50) - (t .> 200)) + 50f0 * ((t .> 250) - (t .> 400)), 1, ~
    N) # injection current

# 記録用
varr = zeros(Float32, nt, N)

# modelの定義
neurons = LIF{Float32}(N=N)

# simulation
@time for i = 1:nt
    update!(neurons, neurons.param, Ie[i, :], dt)
    neurons.tcount += 1
    varr[i, :] = neurons.v_
end
```

発火時電位を含む膜電位 v_- と入力電流 I を描画する。

```
figure(figsize=(4, 4))
subplot(2,1,1); plot(t, varr[:, 1]); ylabel("V (mV)")
subplot(2,1,2); plot(t, Ie[:, 1]); xlabel("Times (ms)")
tight_layout()
```

2.4.3 LIF モデルの F-I curve

数値的計算による F-I curve の描画

この項目では LIF モデルにおける入力電流に対する発火率の変化 (F-I curve) を描画する。方法は HH モデルの場合と同様だが、今回は発火したかどうかがモデル内の変数として明示的に記録されているので処理が少ない。

```
T = 1000 # ms
```

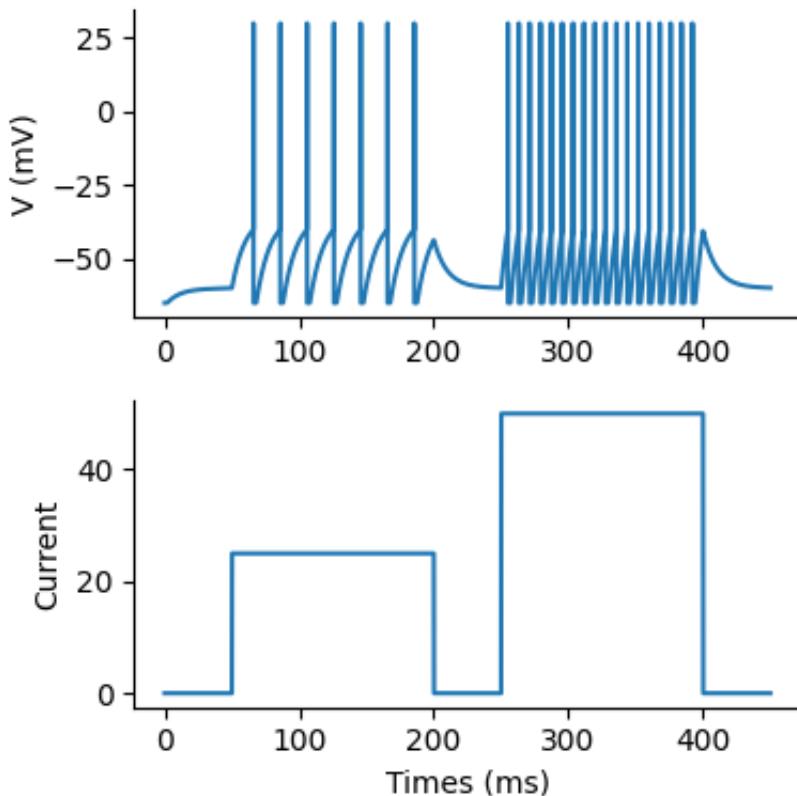


図 2.7 cell010.png

```

dt = 0.01f0 # ms
nt = UInt32(T/dt) # number of timesteps
N = 100 # ニューロンの数

# 入力刺激
mincurrent, maxcurrent = 15, 40
t = Array{Float32}(1:nt)*dt
Ie = Array{Float32}((range(mincurrent,maxcurrent,length=N))) # injection current

# modelの定義
neurons = LIF{Float32}(N=N)

# 記録用
firearr = zeros(Bool, nt, N)

# simulation
@time for i = 1:nt
    update!(neurons, neurons.param, Ie, dt)
    neurons.tcount += 1
    firearr[i, :] = neurons.fire

```

```
end
```

発火率を計算し、描画する。

```
num_spikes = sum(firearr, dims=1)
rate_numeric = num_spikes/T*1e3;

figure(figsize=(4, 3))
plot(Ie, rate_numeric[1, :]); xlabel("Input current"); ylabel("Firing rate (Hz)")
tight_layout()
```

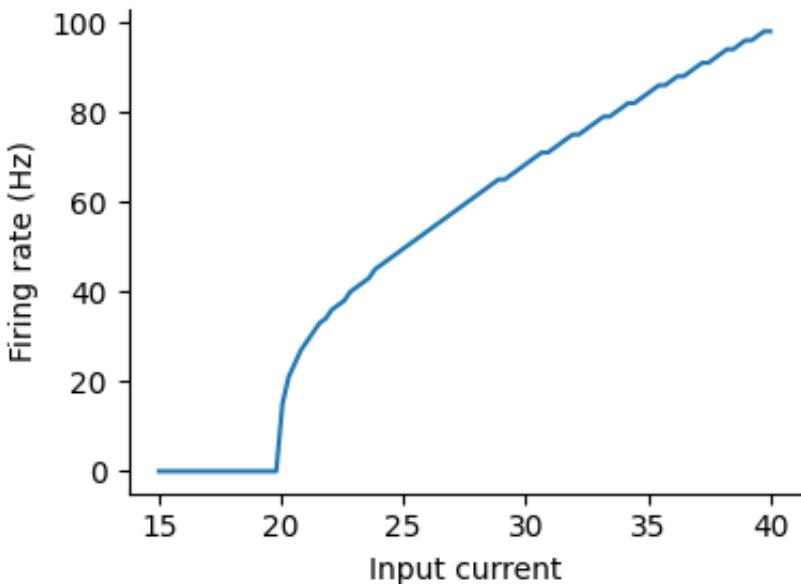


図 2.8 cell015.png

さらに電流を強めると発火率は飽和 (saturation) する。なお、不応期がない、すなわち 0 の場合は閾値付近以外は ReLU 関数のような挙動をする。

解析的計算による F-I curve の描画

ここまででは数値的なシミュレーションにより F-I curve を求めた。以下では解析的に F-I curve の式を求めよう。具体的には、一定かつ持続的な入力電流を I としたときの LIF ニューロンの発火率 (firing rate) が

$$\text{rate} \approx \left(\tau_m \ln \frac{R_m I}{R_m I + V_{\text{rest}} - V_{\text{th}}} \right)^{-1} \quad (2.16)$$

と近似できることを示す。まず、 $t = t_1$ にスパイクが生じたとする。このとき、膜電位はリセットされるので $V_m(t_1) = V_{\text{rest}}$ である（リセット電位と静止膜電位が同じと仮定する）。 $[t_1, t]$ における膜電位は LIF の式を積分することで得られる。

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\text{rest}}) + R_m I \quad (2.17)$$

の式を積分すると、

$$\int_{t_1}^t \frac{\tau_m dV_m}{R_m I + V_{\text{rest}} - V_m} = \int_{t_1}^t dt \quad (2.18)$$

$$\ln \left(1 - \frac{V_m(t) - V_{\text{rest}}}{R_m I} \right) = -\frac{t - t_1}{\tau_m} \quad (\because V_m(t_1) = V_{\text{rest}}) \quad (2.19)$$

$$V_m(t) = V_{\text{rest}} + R_m I \left[1 - \exp \left(-\frac{t - t_1}{\tau_m} \right) \right] \quad (2.20)$$

となる。 $t > t_1$ における初めのスパイクが $t = t_2$ に生じたとすると、そのときの膜電位は $V_m(t_2) = V_{\text{th}}$ である（実際には閾値以上となっている場合もあるますが近似する）。 $t = t_2$ を上の式に代入して

$$V_{\text{th}} = V_{\text{rest}} + R_m I \left[1 - \exp \left(-\frac{t_2 - t_1}{\tau_m} \right) \right] \quad (2.21)$$

$$T = t_2 - t_1 = \tau_m \ln \frac{R_m I}{R_m I + V_{\text{rest}} - V_{\text{th}}} \quad (2.22)$$

となる。ここで T は 2 つのスパイクの時間間隔 (spike interval) である。 $t_1 \leq t < t_2$ におけるスパイクは $t = t_1$ 時の 1 つなので、発火率は $1/T$ となる。よって

$$\text{rate} \approx \frac{1}{T} = \left(\tau_m \ln \frac{R_m I}{R_m I + V_{\text{rest}} - V_{\text{th}}} \right)^{-1} \quad (2.23)$$

となる。不応期 τ_{ref} を考慮すると、持続的に入力がある場合は単純に τ_{ref} だけ発火が遅れるので発火率は $1/(\tau_{\text{ref}} + T)$ となる。それではこの式に基づいて F-I curve を描画してみよう。

```
R = 1.0 # 膜抵抗
tc_m, tref = 10, 2# 膜時定数, 不応期 (ms)
vrest, vthr = -60.0, -40.0 # 静止膜電位, 閾値電位 (mV)
rate_exact = zeros(N)

for i = 1:N
    z = R*Ie[i] / (R*Ie[i] + vrest - vthr)
    rate_exact[i] = (z > 0) ? 1 / (tref + tc_m * log(z)) * 1e3 : 0
end
```

`log` の中身が 0 になると Error が生じるので 3 項演算子で場合分けをしている。なお、`1e3` を乗じているのは 1/ms から Hz に変換するためである。結果は次のようになる。数値的な計算結果とほぼ一致していることがわかる。

```
figure(figsize=(4, 3))
plot(Ie, rate_exact); xlabel("Input current"); ylabel("Firing rate (Hz)")
tight_layout()
```

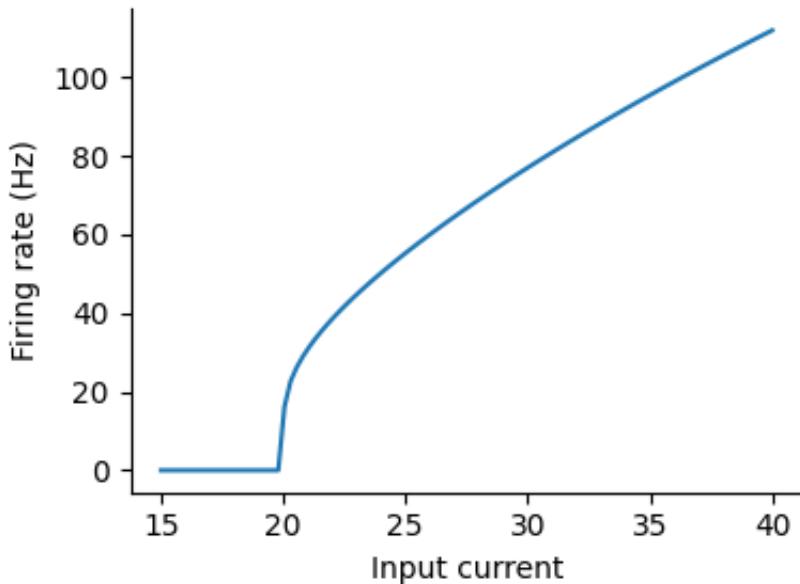


図 2.9 cell020.png

2.5 Izhikevich モデル

2.5.1 Izhikevich モデルの定義

Izhikevich モデル（または Simple model）は HH モデルと LIF モデルの中間のようなモデルである cite:pIzhikevich2003-by. HH モデルのような生理学的な知見に基づいたモデルは実際のニューロンの発火特性をよく再現できるが、式が複雑化するため、数学的な解析が難しく、計算量が増えるために大規模なシミュレーションも困難となる^{*11}. そこで、生理学的な正しさには目をつぶり、生体内でのニューロンの発火特性を再現するモデルが求められた. その特徴を持つのが Izhikevich モデルである（以下では Iz モデルと表記する）. Iz モデルは 2 変数しかない^{*12} 簡素な微分方程式だが、様々

^{*11} これに関しては必ずしも正しくない. 計算機の発達により HH モデルで大きなモデルをシミュレーションすることも可能である.

^{*12} 数値計算をする上では簡易的だが、if 文が入るために解析をするのは難しくなる. ([Bernardo, et al., 2008](<https://www.springer.com/gp/book/9781846280399>)<http://www.scholarpedia.org/article/>

なニューロンの活動を模倣することができる (Izhikevich, 2004). 定式化には主に 2 種類ある. まず, cite:pIzhikevich2003-by で提案されたのが次式である.

$$\frac{dv(t)}{dt} = 0.04v(t)^2 + 5v(t) + 140 - u(t) + I(t) \quad (2.24)$$

$$\frac{du(t)}{dt} = a(bv(t) - u(t)) \quad (2.25)$$

ここで, v と u が変数であり, v は膜電位 (membrane potential; 単位は mV), u は回復電流 (recovery current; 単位は pA)^{*13} である. また, a は回復時定数 (recovery time constant; 単位は ms⁻¹) の逆数 (これが大きいと u が元に戻る時間が短くなる), b は u の v に対する感受性 (共鳴度合い, resonance; 単位は pA/mV) である. この式は簡便だが, 生理学的な意味づけが分かりにくい. 改善された式として (Izhikevich, 2007) の Chapter 8 で紹介されているのが次式である.

$$C \frac{dv(t)}{dt} = k(v(t) - v_r)(v(t) - v_t) - u(t) + I(t) \quad (2.26)$$

$$\frac{du(t)}{dt} = a\{b(v(t) - v_r) - u(t)\} \quad (2.27)$$

ここで, C は膜容量 (membrane capacitance; 単位は pF), v_r は静止膜電位 (resting membrane potential; 単位は mV), v_t は閾値電位 (instantaneous threshold potential; 単位は mV), k はニューロンのゲインに関わる定数で, 小さいと発火しやすくなる (単位は pA/mV). 以後はこちらの式を用いる. Iz モデルの閾値の取り扱いは LIF モデルと異なり, HH モデルに近い. LIF モデルでは閾値を超えた時に膜電位をピーク電位まで上昇させ (この過程は無くてもよい), 続いて膜電位をリセットする. Iz モデルの閾値は v_t だが, 膜電位のリセットは閾値を超えたかで判断せず, 膜電位 v がピーク電位 v_{peak} になったとき (または超えた時) に行う. そのため Iz モデルの実際の閾値は膜電位の挙動が変化する (発火状態に移行する), つまり分岐 (bifurcation) が生じる点であり, パラメータの閾値 v_t との間には差異がある. さて, 膜電位がピーク電位 v_{peak} に達したとき (すなわち **if** $v \geq v_{peak}$), u, v を次のようにリセットする^{*14}.

$$u \leftarrow u + d \quad (2.28)$$

$$v \leftarrow v_{reset} \quad (2.29)$$

とする. ただし, v_{reset} は過分極を考慮して静止膜電位 v_r よりも小さい値とする. また, d はスパイク発火中に活性化される正味の外向き電流の合計を表し, 発火後の膜電位の挙動に影響する (単位は pA). 以上を踏まえて, シミュレーションを行う. まず, 必要なパッケージを読み込む.

Piecewise_smooth_dynamical_systems を読むといいらしい.

*13 ここでの「回復」というのは脱分極した後の膜電位が静止膜電位へと戻る, という意味である (対義語は activation で膜電位の上昇を意味する). u は v の導関数において v の上昇を抑制するように $-u$ で入っているため, u としては K⁺ チャネル電流や Na⁺ チャネルの不活性化動態などが考えられる.

*14 バースト発火 (bursting) の挙動を表現するためには, 速い回復変数 (fast recovery variable) と遅い回復変数 (slow recovery variable) の 2 つが必要となる (従って膜電位も合わせて全部で 3 変数必要). 一方で, Iz モデルでは LIF モデルのような if 文によるリセットを用いているため, 速い回復変数が必要なく, 遅い回復変数 u のみでバースト発火を表現できる.

```
using Parameters: @unpack # or using UnPack
using PyPlot
rc("axes.spines", top=false, right=false)
```

変更しない定数を保持する **struct** の IZParameter と、変数を保持する **mutable struct** の IZ を作成する。2つの定式化でパラメータの値が異なるので注意すること。

```
@kwdef struct IZParameter{FT}
    C::FT = 100 # 膜容量 (pF)
    a::FT = 0.03; b::FT = -2 # 回復時定数の逆数 (1/ms), uのvに対する共鳴度合い (pA/mV)
    d::FT = 100; k::FT = 0.7 # 発火で活性化される正味の外向き電流 (pA), ゲイン (pA/mV)
    vthr::FT = -40; vrest::FT = -60; vreset::FT = -50; vpeak::FT = 35 # 閾値電位, ←
        静止膜電位, リセット電位, ピーク電位 (mV)
end

@kwdef mutable struct IZ{FT}
    param::IZParameter = IZParameter{FT}()
    N::UInt32
    v)::Vector{FT} = fill(param.vrest, N); u)::Vector{FT} = zeros(N)
    fire)::Vector{Bool} = zeros(Bool, N)
end
```

次に変数を更新する関数 update! を書く。LIF の場合と異なり、 $v[i] \geq v_{\text{peak}}$ であることに注意する ($v[i] \geq v_{\text{thr}}$ ではない)。

```
function update!(variable::IZ, param::IZParameter, Ie::Vector, dt)
    @unpack N, v, u, fire = variable
    @unpack C, a, b, d, k, vthr, vrest, vreset, vpeak = param
    @inbounds for i = 1:N
        v[i] += dt/C * (k*(v[i]-vrest)*(v[i]-vthr) - u[i] + Ie[i])
        u[i] += dt * (a * (b * (v[i]-vrest) - u[i]))
    end
    @inbounds for i = 1:N
        fire[i] = v[i] >= vpeak
        v[i] = ifelse(fire[i], vreset, v[i])
        u[i] += ifelse(fire[i], d, 0)
    end
end;
```

2.5.2 Izhikevich モデルのシミュレーションの実行

いくつかの定数を設定してシミュレーションを実行する。

```
T = 450 # ms
dt = 0.01f0 # ms
nt = UInt32(T/dt) # number of timesteps
N = 1 # ニューロンの数
```

```

# 入力刺激
t = Array{Float32}(1:nt)*dt
Ie = repeat(150f0 * ((t .> 50) - (t .> 200)) + 300f0 * ((t .> 250) - (t .> 400)), 1, -
N) # injection current

# 記録用
varr, uarr = zeros(Float32, nt, N), zeros(Float32, nt, N)

# modelの定義
neurons = IZ{Float32}(N=N)

# simulation
@time for i = 1:nt
    update!(neurons, neurons.param, Ie[i, :], dt)
    varr[i, :], uarr[i, :] = neurons.v, neurons.u
end

```

Plots を読み込み、膜電位 v 、回復変数 u 、入力電流 I を描画する。

```

figure(figsize=(4, 4))
suptitle("Regular Spiking (RS) Neurons")
subplot(3,1,1); plot(t, varr[:, 1]); ylabel("Membrane\n potential (mV)")
subplot(3,1,2); plot(t, uarr[:, 1]); ylabel("Recovery\n current (pA)")
subplot(3,1,3); plot(t, Ie[:, 1]); xlabel("Times ←\n(ms)")
tight_layout(rect=[0,0,1,0.96])

```

2.5.3 様々な発火パターンのシミュレーション

次に様々な発火パターンを模倣するように Iz モデルの定数を変化させてみよう。Intrinsically Bursting (IB) ニューロンと Chattering (CH) ニューロン（または fast rhythmic bursting (FRB) ニューロン）のシミュレーションを行う。基本的には定数を変えるだけである。本書で用いている式における発火パターンに対するパラメータは cite:pIzhikevich2003-by では得られないが、(Izhikevich, 2007) には記載がある。他の発火パターンに関してはこの本を参照のこと。

```

# 記録用
varr_ib, varr_ch = zeros(Float32, nt, N), zeros(Float32, nt, N)
Ie = repeat(500f0 * ((t .> 50) - (t .> 200)) + 700f0 * ((t .> 250) - (t .> 400)), 1, -
N) # injection current

# IB neurons
neurons_ib = IZ{Float32}(N=N,
    param=IZParameter{Float32}(C = 150, a = 0.01, b = 5, k = 1.2, d = 130, vrest = -
-75, vreset = -56, vthr = -45, vpeak = 50))

# CH neurons
neurons_ch = IZ{Float32}(N=N,

```

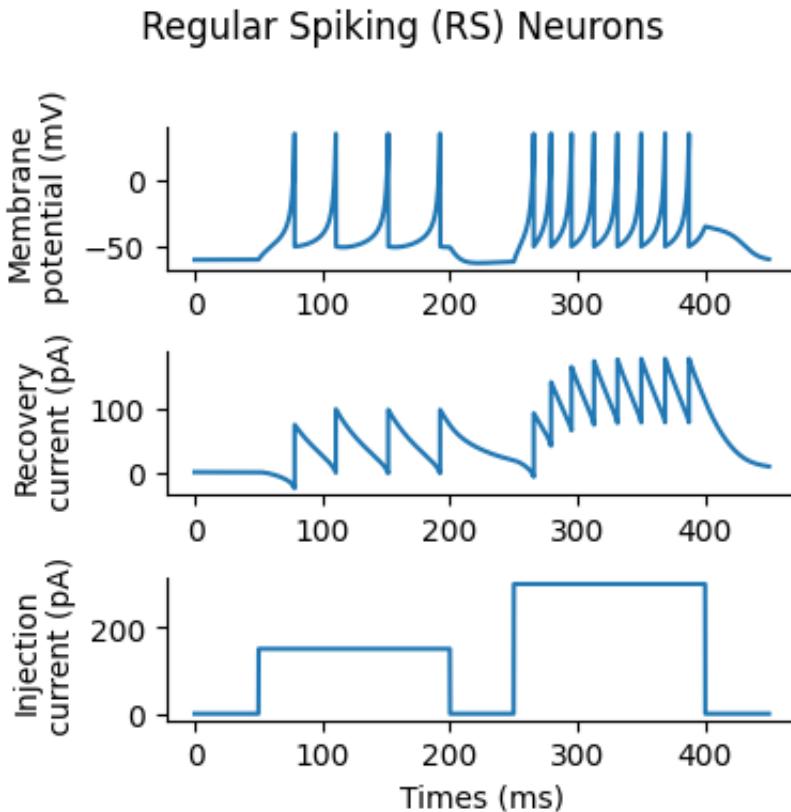


図 2.10 cell009.png

```

param=IZParameter{Float32}(C = 50, a = 0.03, b = 1, k = 1.5, d = 150, vrest = -60,
                         vreset = -40, vthr = -40, vpeak = 35))

# simulation
@time for i = 1:n
    update!(neurons_ib, neurons_ib.param, Ie[i, :], dt)
    update!(neurons_ch, neurons_ch.param, Ie[i, :], dt)
    varr_ib[i, :], varr_ch[i, :] = neurons_ib.v, neurons_ch.v
end

```

これまでと異なり、モデルの定義時に `param` を設定していることに注意しよう。最後に膜電位変化を描画する。

```

figure(figsize=(6, 2))
subplot(1,2,1); plot(t, varr_ib[:, 1]); title("IB Neurons"); ylabel("Membrane\npotential (mV)");
               xlabel("Times (ms)")
subplot(1,2,2); plot(t, varr_ch[:, 1]); title("CH neurons"); xlabel("Times (ms)")

```

```
tight_layout()
```

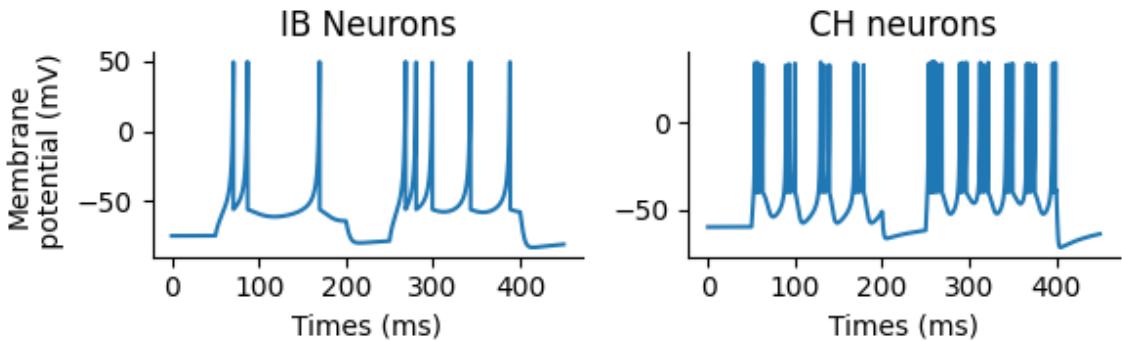


図 2.11 cell013.png

2.6 Inter-spike interval モデル

これまで紹介したモデルでは、入力に対する膜電位などの時間変化に基づき発火が起こるかどうか、ということを考えてきた。この節では、発火が生じるまでの過程を考慮せず、発火の時間間隔 (inter-spike interval, ISI) の統計による現象論的モデルを考える。これを **Inter-spike interval (ISI)** モデルと呼ぶ。ISI モデルは点過程 (point process) という統計的モデルに基づいており、各モデルには ISI が従う分布の名称がついている。この節では、使用頻度の高い **ポアソン過程 (Poisson process)** モデル、ポアソン過程モデルにおいて不応期を考慮した **死時間付きポアソン過程 (Poisson process with dead time, PPD)** モデル、皮質の定常発火においてポアソン過程モデルよりも当てはまりがよいとされる **ガンマ過程 (Gamma process)** モデルについて説明する。なお、SNNにおいて、ISI モデルは主に画像入力の際に連続値からスパイク列へのエンコードに用いられる。これに限らず入力として用いられることが多い。この節は (Shimazaki, n.d.), (Pachitariu et al., 2010) を参考に執筆した。

2.6.1 ポアソン過程モデル

点過程とポアソン過程

時間に応じて変化する確率変数のことを**確率過程 (stochastic process)** という。さらに確率過程の中で、連続時間軸上において離散的に生起する点事象の系列を**点過程 (point process)** という。スパイクは離散的に起こるので、点過程を用いてモデル化ができるという話である。ポアソン過程 (Poisson process) は点過程の 1 つである。ポアソン過程モデルはスパイクの発生をポアソン過程でモデル化したもので、このモデルによって生じるスパイクを**ポアソンスパイク (Poisson spike)** と呼ぶ。ポアソン過程

では、時刻 t までに起こった点の数 $N(t)$ はポアソン分布に従う。すなわち、点が起こる確率が強度 λ のポアソン分布に従う場合、時刻 t までに事象が n 回起こる確率は $P[N(t) = n] = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$ となる。ポアソン過程において点が起こる回数がポアソン分布に従うこととは、ポアソン過程という名称の由来となっている。これを定義とする場合もあれば、次の 4 条件を満たす点過程をポアソン過程とするという定義もある。

1. 時刻 0 における初期の点の数は 0 : $P[N(0) = 0] = 1$
2. $[t, t + \Delta t]$ に点が 1 つ生じる確率 : $P[N(t + \Delta t) - N(t) = 1] = \lambda(t)\Delta t + o(\Delta t)$
3. 微小時間 Δt の間に点は 2 つ以上生じない : $P[N(t + \Delta t) - N(t) = 2] = o(\Delta t)$
4. 任意の時点 $t_1 < t_2 < \dots < t_n$ に対して、増分 $N(t_2) - N(t_1), N(t_3) - N(t_2), \dots, N(t_n) - N(t_{n-1})$ は互いに独立である。

ただし、 $o(\cdot)$ は Landau の記号 (Landau の small o) であり、 $o(x)$ は $x \rightarrow 0$ のとき、 $o(x)/x \rightarrow 0$ となる微小な量を表す。ポアソン過程に従ってスパイクが生じるとする場合、条件 2 の強度関数 $\lambda(t)$ は発火率を意味する (また実装において有用)。条件 3 は不応期より小さいタイムステップにおいては、1 つのタイムステップにおいて 1 つしかスパイクは生じないということを表す。条件 4 はスパイクは独立に発生する、ということを意味する。また、これらの条件から $N(t)$ の分布は強度母数 $\lambda(t)$ のポアソン分布に従うことが示せる。強度関数 (点がスパイクの場合、発火率) が $\lambda(t) = \lambda$ (定数) となる場合は点の時間間隔 (点がスパイクの場合、ISI) の確率変数 T が強度母数 λ の指指数分布に従う。なお、指指数分布の確率密度関数は確率変数を T とするとき、

$$f(t; \lambda) = \begin{cases} \lambda e^{-\lambda t} & (t \geq 0) \\ 0 & (t < 0) \end{cases} \quad (2.30)$$

となる。このことは 4 条件と Chapman-Kolmogorov の式により求められるが、ややこしいので、 $P[N(t) = n] = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$ から導出できることを簡単に示す。指指数分布の累積分布関数を $F(t; \lambda)$ とすると、

$$F(t; \lambda) = P(T < t) = 1 - P(T \geq t) = 1 - P(N(t) = 0) = 1 - e^{-\lambda t} \quad (2.31)$$

となる。よって

$$f(t; \lambda) = \frac{dF(t; \lambda)}{dt} = \lambda e^{-\lambda t} \quad (2.32)$$

が成り立つ。

定常ポアソン過程

ここからポアソン過程によるスパイクのシミュレーションを実装する。実装方法には ISI が指指数分布に従うことを利用したものと、ポアソン過程の条件 2 を利用したものの 2 通りがある。実装は後者が楽で計算量も少ないが、後のガンマ過程のために前者の実装を行なう。

ISI の累積により発火時刻を求める手法 ISI が指数分布に従うことを利用してポアソン過程モデルの実装を行う。まず ISI を指数分布に従う乱数とする。次に ISI を累積することで発火時刻を得る。最後に発火時間を整数値に丸めて index とすることで {0,1} のスパイク列が得られる。ISI の取得には `Random.randexp()` を用いる。この関数は scale 1 の指数分布に従う乱数を返す。この scale は指数分布の確率密度関数を $f(t; \frac{1}{\beta}) = \frac{1}{\beta} e^{-t/\beta}$ とした際の $\beta = 1/\lambda$ である（この時、平均は β となる）。よって発火率を `fr(1/s)`、単位時間 `dt(s)` としたときの ISI は `isi = 1/(fr*dt) * randexp()` として得ることができる。まず必要なパッケージを読み込む。

```
using Random, PyPlot, Distributions
rc("axes.spines", top=false, right=false)
```

乱数の seed 値を設定し、必要な定数を定義した後に `isi` を計算する。`isi` を累積することでスパイクの生じた時刻を記録する配列 `spike_time` を作成する。作成後、`spike_time` を用いてラスター図を描画する。

```
Random.seed!(0) # set random seed

T = 1000 # ms
dt = 1f0 # ms
nt = Int32(T/dt) # number of timesteps

n_neurons = 10 # ニューロンの数
fr = 30 # ポアソンスパイクの発火率(Hz)

isi = 1/(fr*dt*1e-3) * randexp(Int(nt*1.5/fr), n_neurons)
spike_time = cumsum(isi, dims=1); # ISIを累積
```

```
figure(figsize=(6, 2))
for i=1:n_neurons
    scatter(spike_time[:, i], i*ones(Int(nt*1.5/fr)), c="k", s=1)
end
xlabel("Time (ms)"); ylabel("Neuron index"); xlim(0, T+10); tight_layout()
```

`spike_time` のように発火時刻で記録しておく方がメモリを節約できるが、シミュレーションにおいてはスパイク列 S はタイムステップごとに発火しているかを表す {0,1} 配列で保持しておくと楽に扱うことができる。そのため冗長ではあるが、発火時刻の配列を {0,1} 配列 `spikes` に変換しスパイクの数と発火率を計算する。

```
spike_time[spike_time .> nt - 1] .= 1 # ntを超える場合を1に
spike_time = round.(Int, spike_time) # float to int
spikes = zeros(nt, n_neurons) # スパイク記録変数

for i=1:n_neurons
    spikes[spike_time[:, i], i] .= 1
```

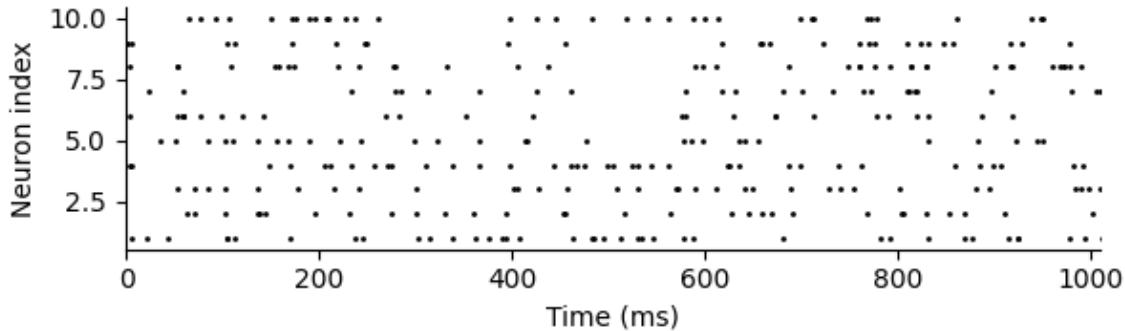


図 2.12 cell006.png

end

```

spikes[1] = 0 # (spike_time=1)の発火を削除
println("Num. of spikes : ", sum(spikes))
println("Firing rate : ", sum(spikes)/(n_neurons*T)*1e3, "Hz")

```

Δt 間の発火確率が $\lambda\Delta t$ であることを利用する方法 次に 2 番目のポアソン過程モデルの実装を行う。こちらは λ を発火率とした場合、区間 $[t, t + \Delta t)$ の間にポアソングラムが発生する確率は $\lambda\Delta t$ となることを利用する。これはポアソン過程の条件だが、ポアソン分布から導けることを簡単に示しておく。事象が起こる確率が強度 λ のポアソン分布に従う場合、時刻 t までに事象が n 回起こる確率は $P[N(t) = n] = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$ となる。よって、微小時間 Δt において事象が 1 回起こる確率は

$$P[N(\Delta t) = 1] = \frac{\lambda \Delta t}{1!} e^{-\lambda \Delta t} \simeq \lambda \Delta t + o(\Delta t) \quad (2.33)$$

となる。ただし、 $e^{-\lambda \Delta t}$ についてはマクローリン展開による近似を行っている。このことから、一様分布 $U(0, 1)$ に従う乱数 ξ を取得し、 $\xi < \lambda dt$ なら発火 ($y = 1$)、それ以外では ($y = 0$) となるようにすればポアソングラムを実装できる。

```

Random.seed!(0) # set random seed

T = 1000 # ms
dt = 1f0 # ms
nt = Int(T/dt) # number of timesteps

n_neurons = 10 # ニューロンの数
fr = 30 # ポアソングラムの発火率(Hz)

spikes = rand(nt, n_neurons) .< fr*dt*1e-3

println("Num. of spikes : ", sum(spikes))

```

```
println("Firing rate : ", sum(spikes)/(n_neurons*T)*1e3, "Hz")
```

```
function rasterplot(spikes)
    # input spike -> time, #neurons
    spike_inds = Tuple.(findall(x -> x > 0, spikes))
    spike_time = first.(spike_inds)
    neuron_inds = last.(spike_inds)

    # raster plot
    scatter(spike_time, neuron_inds, s=1, c="black")
    xlabel("Time (ms)"); ylabel("Neuron index")
    tight_layout()
end
```

```
figure(figsize=(5,2))
rasterplot(spikes)
```

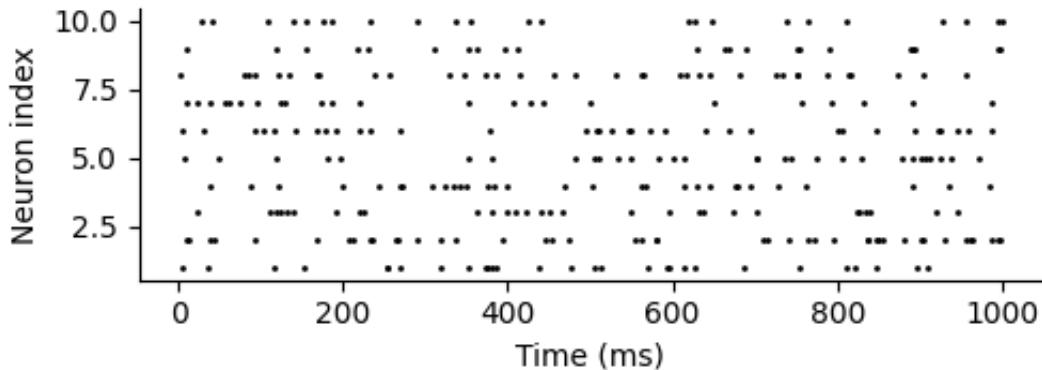


図 2.13 cell012.png

なお、ここでは全時間における発火をまとめて計算しているが、タイムステップごとに発火の有無を計算することもできる。前者は発火情報を保持するためのメモリが必要だが、計算時間は短くなる。後者はメモリの節約になるが、計算時間は長くなる。そのため、これら 2 つの方法はメモリと計算時間のトレードオフとなる。また、他には発火情報を疎行列 (sparse matrix) の形式で保持しておくとメモリの節約になる。

非定常ポアソン過程

これまでの実装は発火率 λ が一定であるとする、定常ポアソン過程 (homogeneous poisson process) であったが、ここからは発火率 $\lambda(t)$ が時間変化するとする**非定常ポアソン過程** (inhomogeneous poisson

process)について考える。とはいって、定常ポアソン過程における発火率を、時間についての関数で置き換えるだけで実装できる。以下は $\lambda(t) = \sin^2(\alpha t)$ (ただし α は定数)とした場合の実装である。

```
Random.seed!(0) # set random seed

T = 1000 # ms
dt = 1f0 # ms
nt = Int32(T/dt) # number of timesteps

n_neurons = Int32(10) # ニューロンの数

t = Array{Float32}(1:nt)*dt
fr = 30(sin.(1e-2t)).^2 # ポアソンスパイクの発火率(Hz)

spikes = rand(nt, n_neurons) .< fr*dt*1e-3

figure(figsize=(5,3))
subplot(2,1,1); plot(t, fr); ylabel("Firing rate (Hz)")
subplot(2,1,2); rasterplot(spikes)
```

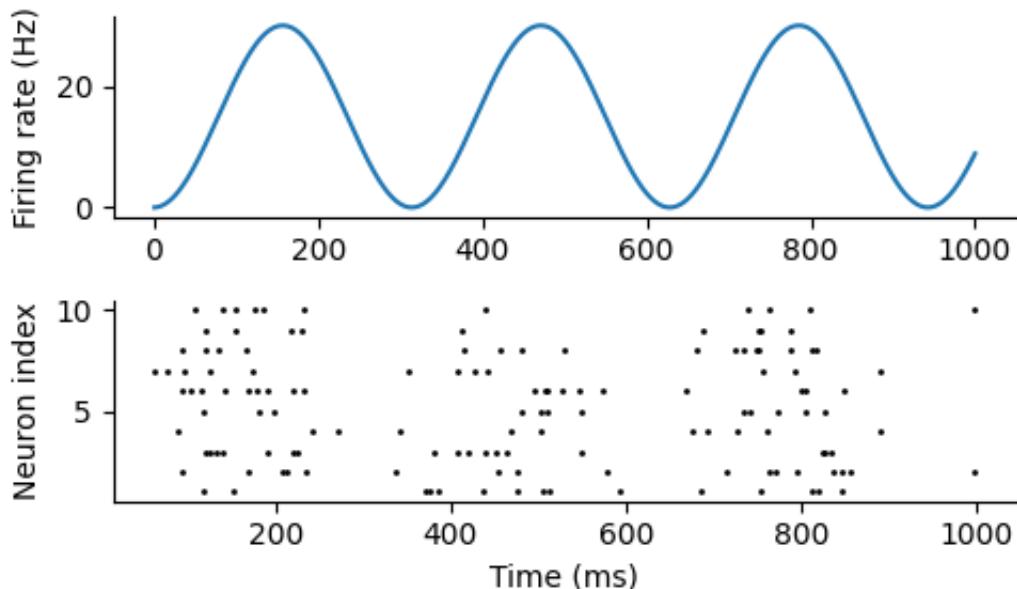


図 2.14 cell015.png

上が発火率 $\lambda(t)$ の時間変化、下がラスター プロットである。

2.6.2 死時間付きポアソン過程モデル (Poisson process with dead time, PPD)

ポアソン過程は簡易的で有用だが、不応期を考慮していない。そのため、時には生理的範疇を超えたバースト発火が起こる場合もある（複数のニューロンからの発火の重ね合わせ (superposition) であると考えることもできる。）。そこで、ポアソン過程において不応期のようなイベントの生起が起こらない **死時間 (dead time)**^{*15} を考慮した**死時間付きポアソン過程 (PPD: Poisson process with dead time または dead time modified Poisson process)** というモデルを導入する。実装においては LIF ニューロンの時と同じような不応期の処理をする。つまり、現在が不応期かどうかを判断し、不応期なら発火を許可しないようにする。

```
Random.seed!(0) # set random seed

T = 1000 # ms
dt = 1f0 # ms
nt = Int32(T/dt) # number of timesteps
tref = 5f0 # 不応期 (ms)

n_neurons = Int32(10) # ニューロンの数
fr = 30 # ポアソンスパイクの発火率(Hz)

tlast, spikes = zeros(n_neurons), zeros(nt, n_neurons) # 発火時刻の記録変数

# simulation
@time for i=1:nt
    fire = rand(n_neurons) .< fr*dt*1e-3
    spikes[i, :] = ((dt*i) .> (tlast .+ tref)) .* fire
    tlast[:] = tlast .* (1 .- fire) + dt*i * fire # 発火時刻の更新
end

println("Num. of spikes : ", sum(spikes))
println("Firing rate : ", sum(spikes)/(n_neurons*T)*1e3, "Hz")
```

不応期があるために発火率は設定値の 30Hz よりも低くなっていることが分かる。次にラスターplot を描画する。

```
figure(figsize=(5,2))
rasterplot(spikes)
```

通常の Poisson spike と差はあまり感じられないが、高頻度発火の場合に通常のモデルとの違いが明瞭となる。

*15 例えば、ガイガー・カウンター (Geiger counter) などの放射線の検出器には放射線の到達を機器の物理的特性として検出できない時間 (つまり死時間) がある。そのため放射線の到達数がポアソン分布に従うとした場合、放射線測定装置のモデルとして PPD が用いられる。

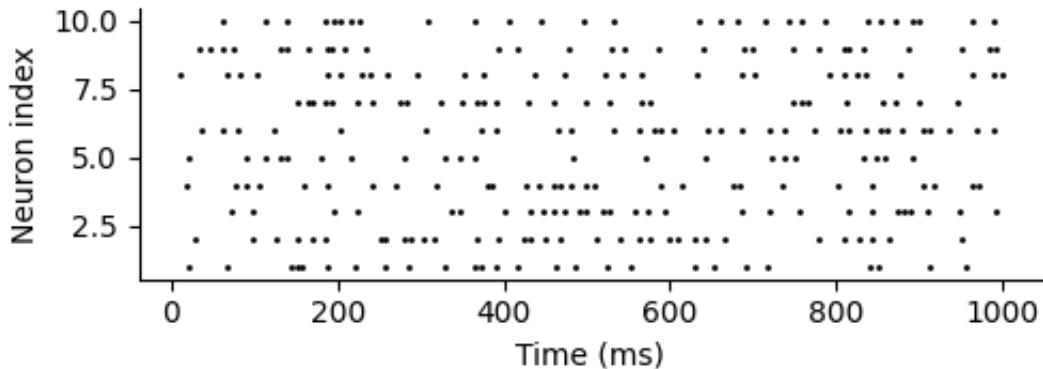


図 2.15 cell020.png

2.6.3 ガンマ過程モデル

ガンマ過程 (gamma process) は点の時間間隔がガンマ分布に従うとするモデルである。ガンマ過程はポアソン過程よりも皮質における定常発火への当てはまりが良いとされている (Maimon and Assad, 2009; Shinomoto et al., 2003)。時間間隔の確率変数を T とした場合、ガンマ分布の確率密度関数は

$$f(t; k, \theta) = t^{k-1} \frac{e^{-t/\theta}}{\theta^k \Gamma(k)} \quad (2.34)$$

と表される。ただし、 $t > 0$ であり、2つの母数は $k, \theta > 0$ である。また、 $\Gamma(\cdot)$ はガンマ関数であり、

$$\Gamma(k) = \int_0^\infty x^{k-1} e^{-x} dx \quad (2.35)$$

と定義される。ガンマ分布の平均は $k\theta$ だが、発火率は ISI の平均の逆数なので、 $\lambda = 1/k\theta$ となる。また、 $k = 1$ のとき、ガンマ分布は指数分布となる。さらに k が正整数のとき、ガンマ分布はアーラン分布となる。ガンマ過程モデルの実装はポアソン過程モデルの ISI を累積する手法と同様に書くことができる。ただしこの時、`Distributions.jl` を用いる。基本的には `randexp(shape)` を `rand(Gamma(a,b), shape)` に置き換えればよい（もちろん多少の修正は必要とする）。スパイク列を生成する関数を書く。

```
function gamma_spike(T, dt, n_neurons, fr, k)
    nt = Int32(T/dt) # number of timesteps
    theta = 1/(k*(fr*dt*1e-3)) # fr = 1/(k*theta)

    isi = rand(Gamma(k, theta), Int32(round(nt*1.5/fr)), n_neurons)
    spike_time = cumsum(isi, dims=1) # ISIを累積
```

```

spike_time[spike_time .> nt - 1] .= 1 # ntを超える場合を1に
spike_time = round.(Int32, spike_time) # float to int
spikes = zeros(Bool, nt, n_neurons) # スパイク記録変数

for i=1:n_neurons
    spikes[spike_time[:, i], i] .= 1
end

spikes[1] = 0 # (spike_time=1)の発火を削除
return spikes
end

```

`gamma_spike` 関数を用いて $k = 1, 12$ の場合のシミュレーションを実行する。なお、 $k = 1$ のときはポアソン過程に一致することに注意しよう。

```

Random.seed!(0) # set random seed

T = 1000 # ms
dt = 1f0 # ms
nt = Int32(T/dt) # number of timesteps

n_neurons = 10 # ニューロンの数
fr = 30 # ガンマスパイクの発火率(Hz)

# k=1のときはポアソン過程に一致
spikes1 = gamma_spike(T, dt, n_neurons, fr, 1)
spikes2 = gamma_spike(T, dt, n_neurons, fr, 12)

println("Num. of spikes : ", sum(spikes1), ", ", sum(spikes2))
println("Firing rate : ", sum(spikes1)/(n_neurons*T)*1e3, "Hz, ", -
       sum(spikes2)/(n_neurons*T)*1e3, "Hz")

```

ISI の分布を描画するための関数を定義する。

```

function gamma_isi_plot(dt, fr, k, n=1000)
    theta = 1/(k*(fr*dt*1e-3)) # fr = 1/(k*theta)
    isi = rand(Gamma(k, theta), n)
    gamma_pdf = pdf.(Gamma(k, theta), minimum(isi):maximum(isi))

    hist(isi, bins=20, density=true, alpha=0.5, ec="black");
    plot(minimum(isi):maximum(isi), gamma_pdf, color="black");
    xlabel("ISI (ms)"); ylabel("Density");
end

```

結果を描画する。上段は ISI の分布、下段はラスター プロットである。左の $k = 1$ の場合をポアソン過程モデルのスパイク列と比較しよう（同じ外観になっていることが分かる）。右は $k = 12$ とした場合である。

```

figure(figsize=(6, 4))
subplot(2,2,1); gamma_isi_plot(dt, fr, 1)

```

```
subplot(2,2,2); gamma_isi_plot(dt, fr, 12)
subplot(2,2,3); rasterplot(spikes1)
subplot(2,2,4); rasterplot(spikes2)
tight_layout()
```

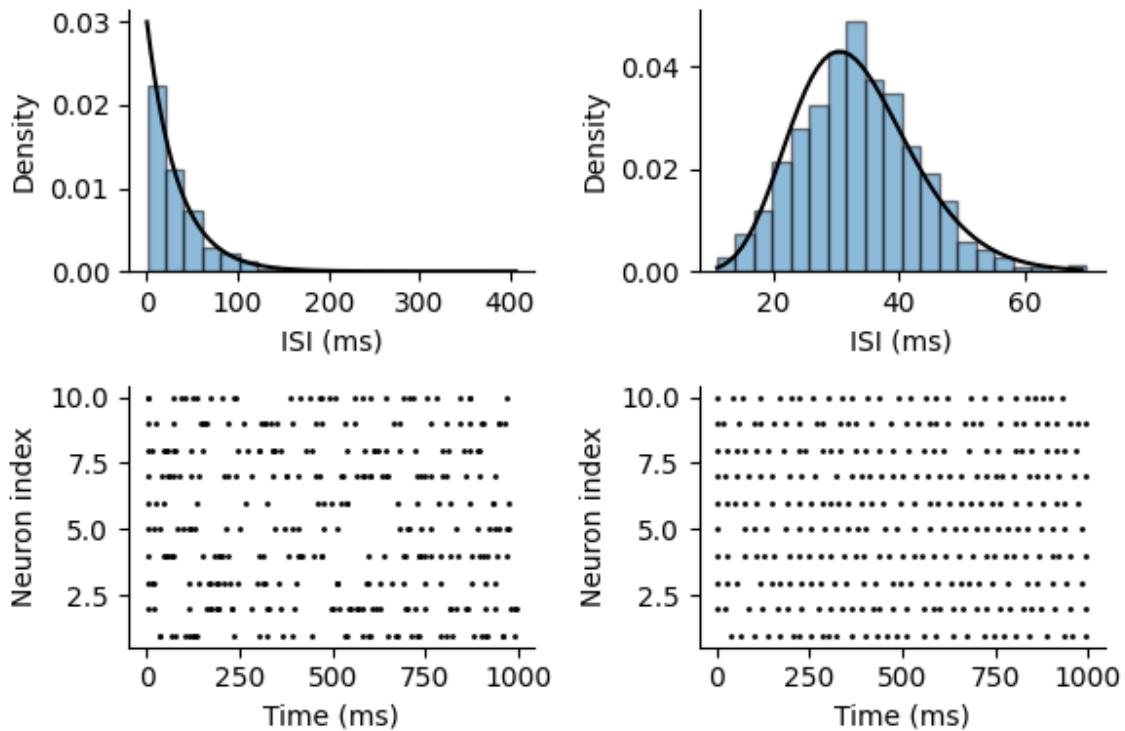


図 2.16 cell030.png

なお、前述したようにガンマ過程モデルの方がポアソン過程モデルよりも皮質ニューロンのモデルとしては優れているが、入力画像のエンコーディングをガンマ過程モデルにすることでSNNの認識精度が向上するかどうかはまだ十分に研究されていない。また、(Deger et al., 2012)ではPPDやガンマ過程の重ね合わせによるスパイク列を生成するアルゴリズムを考案している。

2.7 神経突起の成長モデル

神経細胞は他の細胞に比して特異な形態を持つ。またニューロンの種類およびその役割により基本的な細胞体、樹状突起、軸索等の構造は共通するものの、各部分の形態は異なる。このような形態はどのようにして発達するのだろうか。本節では神経突起(neurite)の成長モデル(growth model)を取り扱う。神経突起とは神経細胞において細胞体から伸びる突起の総称である。

2.7.1 神経突起の木構造

神経突起の形態は樹状突起 (dendrites; ギリシャ語で木を意味する*déndron*に由来) に代表されるように (生物としての) 木に類似している。さらに分節 (segment) に離散化することでグラフ理論における木 (tree; 連結で閉路を持たないグラフ) として捉えることができる。シミュレーション用にデータ構造を作成しよう。なお、Julia で木構造を扱うためのライブラリ `AbstractTrees.jl` は使用しない。`tree_info` は Int 型 vector (要素数 3) の list であり、接続している分節の番号、遠心性位数、分節の種類 (1: 末端, 0: 中間) を表す。`seg_vec` は Float 型 vector (要素数 2) の list であり、分節の 2 次元極座標ベクトル (半径, 角度) を表す。3 次元に拡張することも可能であるが、本書では簡単のために 2 次元とする。多次元配列ではなく vector の list にしているのは、成長に伴って要素を追加していく際に配列に結合 `cat` するより list 化して追加 `push!` する方が高速なためである。

```
using PyPlot, ProgressMeter, Distributions, Random
using PyPlot: matplotlib

rc("axes.spines", top=false, right=false)

tree_info_eg = Vector{Int64}[[1, 0, 0], [1, 0, 0], [2, 1, 0], [2, 1, 0], [3, 2, 1],
                           [3, 2, 1], [4, 2, 1], [4, 2, 1], [8, 2, 1], [9, 2, 1], ~
                           [9, 2, 1]];
seg_vec_eg = Vector{Float64}[[0.0, 0.0], [1, π/2], [√2, 3π/4], [√2, π/4], [√2/2, ~
                           3π/4], ~
                           [√2/2, π/4], [√2/2, 3π/4], [√2/2, π/4], [√2/2, π/4], ~
                           [√2/2, 3π/4], [√2/2, π/4]];
```

木構造を描画するための関数を作成する。以下の `segments_lines` は `tree_info` と `seg_vec` から節点位置 `pos` と各分節の両端点 `lines` を返す関数である。`lines` は主に `matplotlib.collections.LineCollection` で用いる (`plot` を用いるより高速である)。

```
function segments_lines(tree_info, seg_vec, init_pos=nothing)
    num_segments = size(tree_info)[1]

    pos = zeros(num_segments, 2);
    if init_pos != nothing
        pos[1, :] = init_pos
    end

    for j in 1:num_segments
        pos[j, :] = pos[tree_info[j][1], :] + seg_vec[j][1] * [cos(seg_vec[j][2]), ~
                                                               sin(seg_vec[j][2])]
    end

    lines = []
    for j in 1:num_segments
```

```
    x1, y1 = pos[tree_info[j][1], :]
    x2, y2 = pos[j, :]
    push!(lines, [(x1, y1), (x2, y2)])
end
return lines, pos
end;
```

```
lines_eg, _ = segments_lines(tree_info_eg, seg_vec_eg);
```

木構造を描画してみよう。以下では各部位の説明を加えており、(Koene et al., 2009), (Cuntz et al., 2010) を参考に作成した。

```
rc("font", family="Meiryo") # 日本語用フォント
```

```
figure(figsize=(6, 6), dpi=100)
ax = PyPlot.axes()
for i in 1:length(tree_info_eg)
    x, y = lines_eg[i][1]
    dx, dy = lines_eg[i][2] .- lines_eg[i][1]
    arrow(x, y, dx, dy, width=0.01, head_width=0.1, head_length=0.1, length_includes_head=true, color="k")
end
scatter(0, 0, s=100, color="k") # root

text(3, 4, L"遠心性位数 $\gamma$*\n (centrifugal order)", size=10, color="tab:red", ha="center", va="center")
hlines_y = [0, 0, -1, 2];
for i in 1:4
    hlines(i-1, hlines_y[i], 3, color="gray", linestyle="dashed", linewidth=2, alpha=0.5)
    text(3, (i-1)+0.2, string(i-1), size=10, color="tab:red", ha="center", va="center")
end

arrowprops=Dict("arrowstyle" => "->", "color" => "tab:blue");
annotate("根 (root)", xy=(-0.1, 0), xytext=(-1.0, 0), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("根分節\n (root segments)", xy=(0, 0.5), xytext=(-1.6, 0.5), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("中間分節\n (internal segments)", xy=(-0.5, 1.5), xytext=(-2.2, 1.5), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("末端分節\n (terminal segments)", xy=(-1.25, 2.25), xytext=(-2.8, 2.25), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("分岐点\n (branch point)", xy=(1.8, 3), xytext=(0.6, 3), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("成長円錐\n (growth cone)", xy=(1.5, 3.5), xytext=(0.3, 3.5), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
annotate("継続点\n (continuation point)", xy=(1.5, 2.4), xytext=(1.5, 1.5), size=10, color="tab:blue", ha="center", va="center", arrowprops=arrowprops)
```

```

annotate("一組の娘枝\n(daughter branches)", xy=(-0.75, 2.25), xytext=(-1.25, 3), -
        size=10, color="tab:purple", ha="center", va="center", -
        arrowprops=Dict("arrowstyle" => "->", "color" => "tab:purple"))
annotate("", xy=(-1.25, 2.25), xytext=(-1.1, 2.74), ha="center", va="center", -
        arrowprops=Dict("arrowstyle" => "->", "color" => "tab:purple"))

xlim(-4, 3); ylim(-0.3, 4); axis("off")
ax.set_aspect("equal")
tight_layout()

```

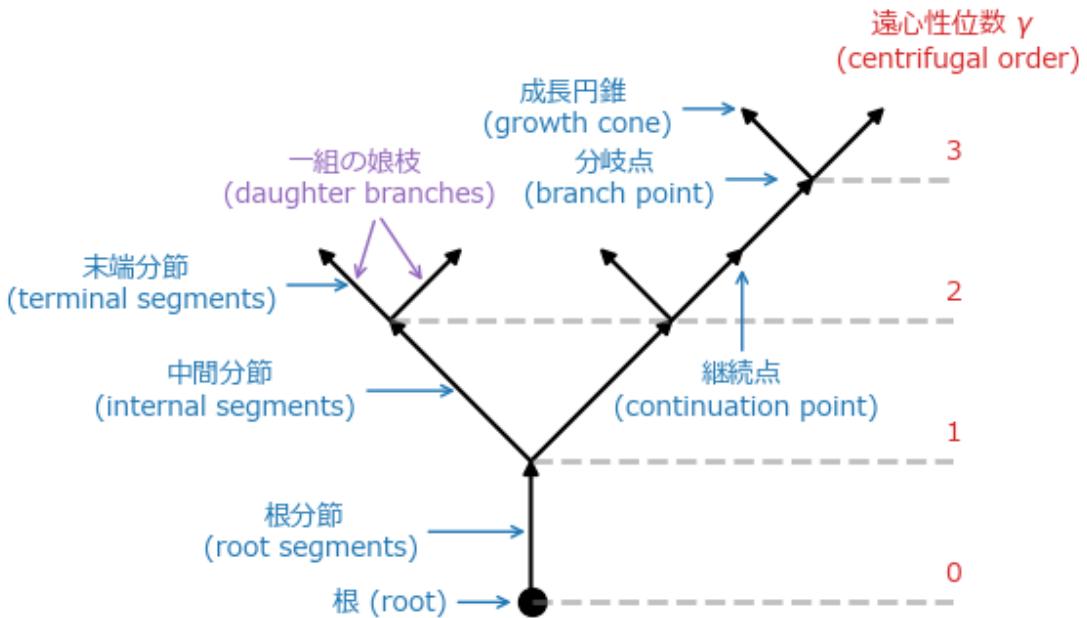


図 2.17 cell009.png

2.7.2 Van Pelt モデル

Van Pelt モデルは Van Pelt らによって構築された、神経突起の成長についての現象論的モデルである (Pelt and Uylings, 2002). 以下では (Koene et al., 2009)に基づいて記述する. なお、このモデルでは軸索誘導分子 (axon guidance molecules) 等の存在は無視している. 神経突起の成長の過程には分岐 (branching), 伸長 (elongation), 転向 (turn) が含まれる. 簡略化のため、空間を 2 次元にし、分節の太さおよび成長円錐が向きを変える時の segment history tension model (後述) を省略する. また Van Pelt モデルを元にした神経回路構築ソフトウェア NETMORPH (Koene et al., 2009) ではシナプス結合の形成も含めたシミュレーションを行っている.

分岐 (branching)

時刻 $[t_i, t_i + \Delta t]$ において, j 番目の末端分節 (terminal segment) が分岐する確率は

$$p_{i,j} = n_i^{-E} \cdot B_\infty e^{\frac{-t_i}{\tau}} \left(e^{\frac{\Delta t}{\tau}} - 1 \right) \cdot \frac{2^{-S\gamma_j}}{C_{n_i}} \quad (2.36)$$

で表される. ここで, B_∞, E, S, τ は定数である. γ_j は j 番目の末端分節の遠心性位数 (centrifugal order) であり, n_i は時刻 t_i における末端分節の総計である. さらに

$$C_{n_i} = \frac{1}{n_i} \sum_{k=1}^{n_i} 2^{-S\gamma_k} \quad (2.37)$$

とする. n_i^{-E} は末端分節の総計に応じて分岐確率を変化させる項であり, E は競合変数 (competition parameter) と呼ばれる. $B_\infty e^{\frac{-t_i}{\tau}} \left(e^{\frac{\Delta t}{\tau}} - 1 \right)$ は経過時間に応じて分岐確率を変化させる項であり, B_∞ は $E = 0$ の場合の末端分節での分岐数の漸近的な期待値である. $\frac{2^{-S\gamma_j}}{C_{n_i}}$ の項は末端分節の遠心性位数に応じて分岐確率を変化させる項であり, C_{n_i} は正規化定数である. $S = 0$ のときは末端分節は全て同じ確率で分岐するが, $S > 0$ のときは近位の末端分節, $S < 0$ のときは遠位の末端分節における分岐確率が大きくなる.

```
function branching_prob(t, dt, γ, n, C, B∞, E, S, τ)
    return (n^(-E))*B∞*exp(-t/τ)*(exp(dt/τ) - 1)*(2^(-S*γ))/C
end;
```

伸長 (elongation)

末端分節が伸長する速さ $\nu_e(t_i)$ [$\mu m/s$] は正規分布 $\mathcal{N}(\mu_e, \sigma_e^2)$ に従うとする (Ooyen et al., 2014). 伸長する長さは $\Delta L_j(t_i) = \nu_e(t_i) \cdot \Delta t$ となる.

転向 (turn)

神経突起は真っ直ぐに伸び続けるわけではなく, 向きを時折変えながら伸長する. 伸長時に転向するかどうかの確率 $p_d(t_i)$ を次のようにする.

$$p_d(t_i) = r_L \Delta L_j(t_i) \quad (2.38)$$

ただし, r_L [μm^{-1}] は回転率を表す. 確率 $p_d(t_i)$ により転向する部分は新しい分節として定義する. 転向する角度は (Koene et al., 2009) では転向角度の履歴を考慮した segment history tension model が導入されているが, 本書では前述のように省略する. 代わりに転向角度は一様分布 $U(-\alpha, \alpha)$ ($\alpha \in [0, \frac{\pi}{2}]$) に従うとする. 分岐した際にも娘枝の長さと角度の設定が必要となる. ここでは長さは末端分節の伸長と同じ正規分布に従うとする. また, 分岐角度は 2 つの娘枝について一様分布 $U(0, \beta_1)$, $U(-\beta_2, 0)$ ($\beta_1, \beta_2 \in [0, \frac{\pi}{2}]$) にそれぞれ従うとする. 以上をまとめてシミュレーションを実装する.

```

function neurite_growth_model(
    tree_info_init, seg_vec_init, nt, dt, B∞, E, S, τ, μe, σe,
    turn_rate=5, max_branch_angle=0.1π, max_turn_angle=5e-3π,
    history_num=3)
    tree_info = copy(tree_info_init)
    seg_vec = copy(seg_vec_init)
    num_branching = 0

    if history_num > 1
        tree_info_history = []
        seg_vec_history = []
        history_timing = floor.(Int, collect(range(1, nt, ←
            length=history_num+1)))[2:end]
    end

@showprogress for tt in 1:nt
    t = tt*dt # Current time

    n, C = 0.0, 0.0
    for j in 1:size(tree_info)[1]
        if tree_info[j][3] == 1
            n += 1
            C += 2 ^(-S*tree_info[j][2])
        end
    end
    C /= n

    for j in 1:size(tree_info)[1]
        if tree_info[j][3] == 1
            γ = tree_info[j][2]
            p_branch = branching_prob(t, dt, γ, n, C, B∞, E, S, τ)
            if p_branch > rand()
                # Neurite branching
                num_branching += 1
                branch_lens = (μe .+ randn(2) * σe)*dt # branch length
                branch_angles = [1, -1] .* rand(Uniform(0, max_branch_angle), 2)
                for k in 1:2 # add two daughter branches
                    push!(tree_info, [j, γ+1, 1])
                    push!(seg_vec, [branch_lens[k], ←
                        seg_vec[j][2]+branch_angles[k]])
                end
                tree_info[j][3] = 0 # reset centrifugal order of original branch
            else
                # Neurite elongation
                Δlen = (μe .+ randn() * σe)*dt
                p_turn = turn_rate*Δlen
                if p_turn > rand()
                    push!(tree_info, [j, γ, 1]) # add segment
                    seg_angle = seg_vec[j][2] + rand(Uniform(-max_turn_angle, ←
                        max_turn_angle))
                    push!(seg_vec, [Δlen, seg_angle])
                    tree_info[j][3] = 0 # reset centrifugal order of original ←
                        branch
                else

```

```

                seg_vec[j][1] += Δlen
            end
        end
    end
end
if history_num > 1
    if tt in history_timing
        push!(tree_info_history, copy(tree_info))
        push!(seg_vec_history, copy(seg_vec))
    end
end
end
end
println("Num. branching: ", num_branching)
if history_num > 1
    return tree_info_history, seg_vec_history
else
    return tree_info, seg_vec
end
end;

```

パラメータを設定する。このパラメータは (Koene et al., 2009), (Ooyen et al., 2014) に基づいている。Van Pelt モデルにおいて錐体細胞のような複雑な形態を作成するには、各部位に分割してシミュレーションし結合することが必要となる。今回は軸索のパラメータを用いる。

```

B∞ = 13.2; E = 0.319; S = -0.205; τ = 1681541;
μe = 2.14e-4; σe = 3.98e-4;

turn_rate = 3
max_branch_angle = 0.25π
max_turn_angle = 1e-2π

T = 18*24*60*60 # duration of growth; convert 18 days to sec
dt = 200 # sec
nt = round(Int, T/dt);

```

シミュレーションを実行する。

```

history_num = 3 # 記録する状態の数；等間隔で記録。
init_branch_num = 2 # 初めの神経突起枝の数

# 細胞体のパラメータ
tree_info_init = [[1, 0, 0]]
seg_vec_init = [[0.0, 0.0]]

# 初期神経突起のパラメータ (神経突起が放射状に出るように設定)
Random.seed!(0)
for i in 1:init_branch_num
    push!(tree_info_init, [1, 0, 1])
    push!(seg_vec_init, [(μe + randn() * σe) * dt, (i-1)/init_branch_num * 2π + 1e-2 * randn()])
end

@time tree_info_history, seg_vec_history = neurite_growth_model(

```

```
tree_info_init, seg_vec_init, nt, dt, B∞, E, S, τ, μe, σe,
turn_rate, max_branch_angle, max_turn_angle, history_num);
```

結果を表示する。初めの神経突起の数を 2 にしてもそれ以上出ているように見えるのは最初期に分歧しているためである（細胞体を描画しなければ確認できる）。

```
maxwidth, minwidth = 1.5, 0.5 # 描画する神経突起の最大/最小の太さ
days_range = floor.(Int, collect(range(1, 18, history_num+1)))[2:end]

fig, ax = subplots(1, history_num, sharex=true, sharey=true)
for i in 1:history_num
    lines, _ = segments_lines(tree_info_history[i], seg_vec_history[i]);
    linewidths = range(maxwidth, minwidth, length=length(lines));

    line_segments = matplotlib.collections.LineCollection(lines, -
        linewidths=linewidths, color="k")
    ax[i].add_collection(line_segments) # dendrite
    ax[i].scatter(0, 0, s=50, color="k") # soma
    ax[i].set_xlabel(L"$x\text{ }(\mu\text{m})$"); ax[i].set_ylabel(L"$y\text{ }(\mu\text{m})$")
    ax[i].set_aspect("equal")
    ax[i].set_title(string(days_range[i])*" days")
    ax[i].label_outer()
end
```

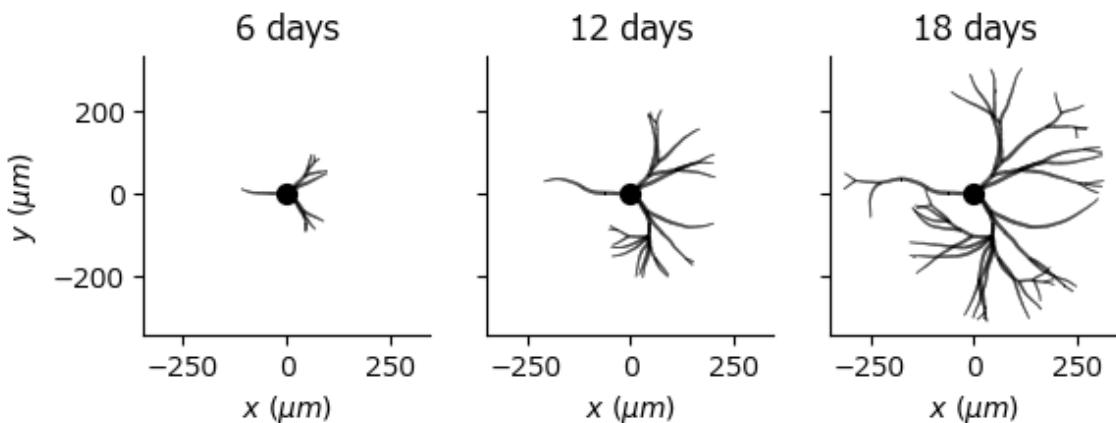


図 2.18 cell019.png

対称性の破れを考慮していないので、円系に成長している。ToDo: 神経細胞極性についての記述。

参考文献

- Chik, D. T. W., Coombes, S., and Wang, Z. D. (2004). “Clustering through postinhibitory rebound in synaptically coupled neurons”. *Phys. Rev. E* 70.1, p. 011908.
- Connor, J. A. and Stevens, C. F. (1971). “Prediction of repetitive firing behaviour from voltage clamp data on an isolated neurone soma”. *J. Physiol.* 213.1, pp. 31–53.
- Connor, J. A., Walter, D., and McKown, R. (1977). “Neural repetitive firing: modifications of the Hodgkin-Huxley axon suggested by experimental results from crustacean axons”. *Biophys. J.* 18.1, pp. 81–102.
- Cuntz, H. et al. (2010). “One rule to grow them all: a general theory of neuronal branching and its practical application”. *PLoS Comput. Biol.* 6.8.
- Dayan, P. and Abbott, L. F. (2005). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press.
- Deger, M. et al. (2012). “Statistical properties of superimposed stationary spike trains”. *J. Comput. Neurosci.* 32.3, pp. 443–463.
- Dodla, R. and Rinzel, J. (2006). “Enhanced neuronal response induced by fast inhibition”. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.* 73.1 Pt 1, p. 010903.
- Fitzhugh, R. (1961). “Impulses and Physiological States in Theoretical Models of Nerve Membrane”. *Biophys. J.* 1.6, pp. 445–466.
- FitzHugh, R. (1955). “Mathematical models of threshold phenomena in the nerve membrane”. *Bull. Math. Biophys.* 17.4, pp. 257–278.
- Hodgkin, A. L. and Huxley, A. F. (1952). “A quantitative description of membrane current and its application to conduction and excitation in nerve”. *J. Physiol.* 117.4, pp. 500–544.
- Hopper, A. J., Beswick-Jones, H., and Brown, A. M. (2022). “A color-coded graphical guide to the Hodgkin and Huxley papers”. *Adv. Physiol. Educ.* 46.4, pp. 580–592.
- Izhikevich, E. M. (2007). *Dynamical Systems in Neuroscience*. MIT press.
- (2004). “Which model to use for cortical spiking neurons?” *IEEE Trans. Neural Netw.* 15.5, pp. 1063–1070.
- Koene, R. A. et al. (2009). “NETMORPH: a framework for the stochastic generation of large scale neuronal networks with realistic neuron morphologies”. *Neuroinformatics* 7.3, pp. 195–210.
- Maimon, G. and Assad, J. A. (2009). “Beyond Poisson: increased spike-time regularity across primate parietal cortex”. *Neuron* 62.3, pp. 426–440.
- Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). “An Active Pulse Transmission Line Simulating Nerve Axon”. *Proceedings of the IRE* 50.10, pp. 2061–2070.

- Ooyen, A. van et al. (2014). “Independently outgrowing neurons and geometry-based synapse formation produce networks with realistic synaptic connectivity”. *PLoS One* 9.1, e85858.
- Pachitariu, M. et al. (2010). “Probabilistic models for spike trains of single neurons”.
- Pelt, J. van and Uylings, H. B. M. (2002). “Branching rates and growth functions in the outgrowth of dendritic branching patterns”. *Network* 13.3, pp. 261–281.
- Schwiening, C. J. (2012). “A brief historical perspective: Hodgkin and Huxley”. *J. Physiol.* 590.11, pp. 2571–2575.
- Shimazaki, H. (n.d.). **スパイク統計モデル入門**. <https://www.neuralengine.org/res/book/index.html>.
- Shinomoto, S., Shima, K., and Tanji, J. (2003). “Differences in spiking patterns among cortical neurons”. *Neural Comput.* 15.12, pp. 2823–2842.

第3章

シナプス伝達のモデル

3.1 シナプスの形態と生理

スパイクが生じたことによる膜電位変化は軸索を伝播し、シナプスという構造により、次のニューロンへと興奮が伝わる。このときの伝達の仕組みとして、シナプスには化学シナプス (chemical synapse) と Gap junction による電気シナプス (electrical synapse) がある。化学シナプスの場合、シナプス前膜からの神経伝達物質の放出、シナプス後膜の受容体への神経伝達物質の結合、イオンチャネル開口によるシナプス後電流 (postsynaptic current; PSC) の発生、という過程が起こる。しかし、これらの過程を全てモデル化するのは計算量がかなり大きくなるので、基本的には簡易的な現象論的なモデルを用いる。このように、シナプス前細胞のスパイク列 (spike train) は次のニューロンにそのまま伝わるのではなく、ある種の時間的フィルターをかけられて伝わる。このフィルターをシナプスフィルター (synaptic filter) と呼ぶ。本章では、このようにシナプス前細胞で生じた発火が、シナプス後細胞の膜電位に与える過程のモデルについて説明する。

3.2 Current / Conductance-based シナプス

3.2.1 化学シナプスの2つの記述形式

具体的なシナプスのモデルの前に、この節では化学シナプスにおけるシナプス入力 (synaptic drive) の2つの形式、Current-based シナプスと Conductance-based シナプスについて説明する。簡単に言うと、Current-based シナプスは入力電流が変化するというモデルで、Conductance-based シナプスはイオンチャネルのコンダクタンス (電気抵抗の逆数、電流の流れやすさ) が変化するというモデルである (Cavallari et al., 2014)。以下では例として、次の LIF ニューロンの方程式におけるシナプス入力を考える。

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\text{rest}}) + R_m I_{\text{syn}}(t) \quad (3.1)$$

ただし、 τ_m は膜電位の時定数、 $V_m(t)$ は膜電位、 V_{rest} は静止膜電位、 R_m は膜抵抗である。ここで、シナプス入力の電流 $I_{\text{syn}}(t)$ ^{*1} が 2 つのモデルにおいて異なる部分となる。

3.2.2 Current-based シナプス

Current-based シナプスは単純に入力電流が変化するというモデルで、モデルを簡素化したい場合によく用いられる。シナプス入力 $I_{\text{syn}}(t)$ はシナプス効率 (synaptic efficacy)^{*2} を J_{syn} (単位は pA) とし、シナプスの動態 (synaptic kinetics) を $s_{\text{syn}}(t)$ とすると、次式のようになる。ただし、シナプスの動態とは前細胞に注目すれば神経伝達物質の放出量、後細胞に注目すれば神経伝達物質の結合量やイオンチャネルの開口率を表す。

$$I_{\text{syn}}(t) = \underbrace{J_{\text{syn}} s_{\text{syn}}(t)}_{\text{電流の変化}} \quad (3.2)$$

ただし、 $s_{\text{syn}}(t)$ は、例えば次節で紹介する α 関数を用いる場合、

$$s_{\text{syn}}(t) = \frac{t}{\tau_s} \exp \left(1 - \frac{t}{\tau_s} \right) \quad (3.3)$$

のようになる。

3.2.3 Conductance-based シナプス

Conductance-based シナプスはイオンチャネルのコンダクタンスが変化するというモデルである。例えば Hodgkin-Huxley モデルは Conductance-based モデルの 1 つである。Current-based よりも Conductance-based の方が生理学的に妥当である。例えば抑制性シナプスは膜電位が平衡電位と比べて脱分極側にあるか、過分極側にあるかで抑制的に働くか興奮的に働くかが逆転するが、これは Current-based シナプスでは再現できない。Conductance-based モデルにおけるシナプス入力は $I_{\text{syn}}(t)$ は次のようになる。

$$I_{\text{syn}}(t) = \underbrace{g_{\text{syn}} s_{\text{syn}}(t)}_{\text{コンダクタンスの変化}} \cdot (V_{\text{syn}} - V_m(t)) \quad (3.4)$$

ただし、 g_{syn} (単位は nS) はシナプスの最大コンダクタンス^{*3}、 V_{syn} (単位は mV) はシナプスの平衡電位を表す。これらも J_{syn} と同じく、シナプスにおける受容体の種類によって決まる定数である。注意しなければならないことは、 $s_{\text{syn}}(t) \leq 0$ としたとき Current-based モデルにおける J_{syn} は正の値 (興奮

*1 シナプス (synapse) 入力であることを明らかにするために syn と添え字をつけている。

*2 シナプス強度 (Synaptic strength) とは違い、受容体の種類 (GABA 受容体や AMPA 受容体、およびそのサブタイプなど) によって決まる。

*3 g_{syn} がシナプスの最大コンダクタンスとなるのは s_{syn} の最大値を 1 に正規化する場合である。正規化は必須ではないので、単なる係数と思うのがよい。

性) と負の値(抑制性)を取るが, g_{syn} は正の値のみである, ということである^{*4}. Conductance-based モデルで興奮性と抑制性を決定しているのは, 平衡電位 V_{syn} である. 興奮性シナプスの平衡電位は高く, 抑制性シナプスの平衡電位は低いため, 膜電位を引いた符号はそれぞれ正と負になる.

3.3 指数関数型シナプスモデル

シナプスのモデルは複数あるが, 良く用いられるのが**指数関数型シナプスモデル** (exponential synapse model) である. このモデルは生理学的な過程を無視した現象論的モデルであることに注意しよう. 指数関数型シナプスモデルには2つの種類, **単一指数関数型モデル** (single exponential model) と **二重指数関数型モデル** (double exponential model) がある. 数式の説明の前にモデルの挙動を示す. 次図は2種類のモデルにおいて $t = 0$ でスパイクが生じてからのシナプス後電流の変化を示している. ただし, 実際のシナプス後電流はこれに**シナプス強度** (Synaptic strength)^{*5} を乗じて総和を取ったものとなる.

```
using PyPlot
rc("axes.spines", top=false, right=false)
```

```
td, tr = 2e-2, 2e-3 # synaptic decay time, synaptic rise time (sec)
dt, T = 5e-5, 0.1 # タイムステップ, シミュレーション時間 (sec)
nt = Int(T/dt) # シミュレーションの総ステップ

# 単一指数関数型シナプス
r_single = zeros(nt)
for t in 1:nt-1
    spike = ifelse(t == 1, 1, 0)
    r_single[t+1] = r_single[t]*(1-dt/td) + spike/td
    #r_single[t+1] = r_single[t]*exp(-dt/td) + spike/td
end

# 二重指数関数型シナプス
r_double, hr = zeros(nt), zeros(nt)
for t in 1:nt-1
    spike = ifelse(t == 1, 1, 0)
    r_double[t+1] = r_double[t]*(1-dt/tr) + hr[t]*dt
    hr[t+1] = hr[t]*(1-dt/td) + spike/(tr*td)
    #r_double[t+1] = r_double[t]*exp(-dt/tr) + hr[t]*dt
    #hr[t+1] = hr[t]*exp(-dt/td) + spike/(tr*td)
end
```

^{*4} これはコンダクタンスが電気抵抗の逆数であり, 基本的に抵抗は正の値しか取らないことからも分かる. なお電子回路においては素子の抵抗値が見かけ上, 負の値を取る場合もあり**負性抵抗** (negative resistance) と呼ばれる

^{*5} シナプス強度というのは便宜上の呼称で, 実際には神経伝達物質の種類や, その受容体の数など複数の要因によって決定されている. また, このシナプス強度はシナプス重みということもある. これはどちらかと言えば機械学習の表現に引っ張られたものである. このため, このサイトでは重みという語も使う.

```

time = (1:nt)*dt
fig, ax = subplots(figsize=(3, 2))
ax.plot(time, r_single, linestyle="dashed", label="single exp.")
ax.plot(time, r_double, label="double exp.")
ax.set_xlabel("Time (s)"); ax.set_ylabel("Post-synaptic\ncurrent (pA)")
ax.legend(); fig.tight_layout()

```

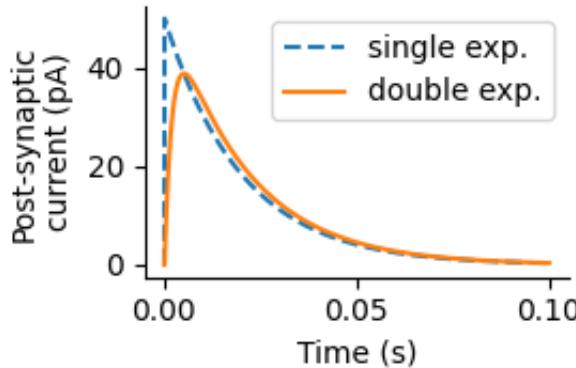


図 3.1 cell003.png

2種類の指数関数型シナプスの動態。破線は単一指数関数型シナプスで、実線は二重指数関数型シナプスである。

3.3.1 単一指数関数型モデル (Single exponential model)

シナプス前ニューロンにおいてスパイクが生じてからのシナプス後電流の変化はおおよそ指数関数的に減少する、というのが単一指数関数型モデルである^{*6}。式は次のようになる。

$$f(t) = \frac{1}{\tau_s} \exp\left(-\frac{t}{\tau_s}\right) \quad (3.5)$$

この関数を時間的なフィルターとして、過去の全てのスパイクについての総和を取る。

$$r(t) = \sum_{t_k < t} f(t - t_k) \quad (3.6)$$

ここで $r(t)$ は前節におけるシナプス動態 (s_{syn}) で、 t_k はあるニューロンの k 番目のスパイクの発生時刻である。 $t_k < t$ の意味は現在の時刻 t までに発生したスパイクについての和を取るという意味である。なお、スパイクが生じてから、ある程度の時間が経過した後はそのスパイクの影響はないと言なせるので、

^{*6} 薬学動態の静注 1 コンパートメントモデルと同じ式である。

一定の時間までの総和を取るのがよい。別の表記法としてスパイク列に対する畳み込みを行うというものもある。畳み込み演算子を $*$ とし、シナプス前細胞のスパイク列を $S(t) = \sum_{t_k < t} \delta(t - t_k)$ とする（ただし、 δ はDiracのdelta関数において $\delta(0) = 1$ とした関数）。このとき、 $r(t) = f * S(t)$ と表すことができる。畳み込み演算子を用いると簡略な表記ができるが、実装上は他と同じ手法を用いる。

微分方程式による表現

上の手法ではニューロンの発火時刻を記憶し、時間毎に全てのスパイクについての和を取る必要がある。そこで、実装する場合は次の等価な微分方程式を用いる。

$$\frac{dr}{dt} = -\frac{r}{\tau_s} + \frac{1}{\tau_s} \sum_{t_k < t} \delta(t - t_k) \quad (3.7)$$

ここで τ_s はシナプスの時定数(synaptic time constant)である。また、 $\delta(\cdot)$ はDiracのdelta関数です（ただし $\delta(0) = 1$ です）。これをEuler法で差分化すると

$$r(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_s}\right) r(t) + \frac{1}{\tau_s} \delta_{t,t_k} \quad (3.8)$$

となる。ここで δ_{t,t_k} はKroneckerのdelta関数で、 $t = t_k$ のときに1、それ以外は0となる。また減衰度として $(1 - \Delta t/\tau_d)$ の代わりに $\exp(-\Delta t/\tau_d)$ を用いる場合もある。

3.3.2 二重指数関数型モデル (Double exponential model)

2重の指数関数によりシナプス後電流の立ち上がりも考慮するのが、二重指数関数型モデル(Double exponential model)である^{*7}。 $t = 0$ にシナプス前細胞が発火したときのシナプス後電流の時間変化の関数は次のようになる。

$$f(t) = A \left[\exp\left(-\frac{t}{\tau_d}\right) - \exp\left(-\frac{t}{\tau_r}\right) \right] \quad (3.9)$$

ただし、 τ_r は立ち上がり時定数(synaptic rise time constant)、 τ_d は減衰時定数(synaptic decay time constant)である。 τ_d は τ_s と同じく神経伝達物質の減少速度を決定している。 A は規格化定数で次のように表される。

$$A = \frac{\tau_d}{\tau_d - \tau_r} \cdot \left(\frac{\tau_r}{\tau_d} \right)^{\frac{\tau_r}{\tau_r - \tau_d}} \quad (3.10)$$

規格化定数 A を乗じることで最大値が1となる。ただし、シミュレーションをする上で実際に規格化をする場合は少ない。

^{*7} 薬学動態の内服1コンパートメントモデルと同じ式である。

α 関数

上記の式において, $\tau = \tau_r = \tau_d$ の場合は **α 関数** (alpha function, alpha synapse) と呼ぶ (Rall, 1967). 式としては次のようになる.

$$\alpha(t) = \frac{t}{\tau} \exp\left(1 - \frac{t}{\tau}\right) \quad (3.11)$$

この式は二重指数関数型シナプスの式に単に代入するだけでは導出できない. これらの式の対応については後述する.

微分方程式による表現

ここで, 二重指数関数型シナプスの式に対応する, 補助変数 h を用いた微分方程式を導入する.

$$\frac{dr}{dt} = -\frac{r}{\tau_d} + h \quad (3.12)$$

$$\frac{dh}{dt} = -\frac{h}{\tau_r} + \frac{1}{\tau_r \tau_d} \sum_{t_k < t} \delta(t - t_k) \quad (3.13)$$

单一指数関数型シナプスの場合と同様に Euler 法で差分化すると

$$r(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_d}\right) r(t) + h(t) \cdot \Delta t \quad (3.14)$$

$$h(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_r}\right) h(t) + \frac{1}{\tau_r \tau_d} \delta_{t,t_{jk}} \quad (3.15)$$

となる. 念のため, 微分方程式と元の式が一致することを確認しておこう. $t = 0$ のときにシナプス前細胞が発火したとし, それ以降の発火はないとする. このとき, $h(0) = 1/\tau_r \tau_d$, $r(0) = 0$ である. h についての微分方程式の解は

$$h(t) = h(0) \cdot \exp\left(-\frac{t}{\tau_r}\right) \quad (3.16)$$

となるので, これを r についての式に代入して

$$\frac{dr}{dt} = -\frac{r}{\tau_d} + h(0) \cdot \exp\left(-\frac{t}{\tau_r}\right) \quad (3.17)$$

となる. これを解くには両辺に積分因子 $\exp(t/\tau_d)$ をかけてから積分をするか Laplace 変換をするかである. 今回は Laplace 変換を用いる. 右辺一項目を移行した後に両辺を Laplace 変換すると以下のよう

になる。

$$\mathcal{L} \left[\frac{dr}{dt} + r/\tau_d \right] = \mathcal{L} [h(0) \cdot \exp(-t/\tau_r)] \quad (3.18)$$

$$sF(s) - r(0) + \frac{1}{\tau_d} F(s) = \frac{h(0)}{s + 1/\tau_r} \quad (3.19)$$

$$F(s) = \frac{h(0)}{(s + 1/\tau_r)(s + 1/\tau_d)} \quad (3.20)$$

ただし $r(t)$ の Laplace 変換を $F(s)$ とした。ここで逆 Laplace 変換を行うと次のようになる。

$$r(t) = \mathcal{L}^{-1}(F(s)) \quad (3.21)$$

$$= \mathcal{L}^{-1} \left[\frac{h(0)}{(s + 1/\tau_r)(s + 1/\tau_d)} \right] \quad (3.22)$$

$$= \mathcal{L}^{-1} \left[\frac{h(0)}{1/\tau_r - 1/\tau_d} \left(\frac{1}{s + 1/\tau_d} - \frac{1}{s + 1/\tau_r} \right) \right] \quad (3.23)$$

$$= \frac{1}{\tau_d - \tau_r} [\exp(-t/\tau_d) - \exp(-t/\tau_r)] \quad (3.24)$$

この式の最大値 r_{\max} を求めておこう。 $r(t)$ を微分して 0 と置いた式の解 t_{\max} を代入すれば求められる。計算すると、

$$t_{\max} = \frac{\ln(\tau_d/\tau_r)}{1/\tau_r - 1/\tau_d}, \quad r_{\max} = \frac{1}{\tau_d} \cdot \left(\frac{\tau_r}{\tau_d} \right)^{\frac{\tau_r}{\tau_d - \tau_r}} \quad (3.25)$$

となる。なお、 α 関数の導出は逆 Laplace 変換をする前に $\tau = \tau_d = \tau_r$ とすればよく、

$$F_\alpha(s) = \frac{h(0)}{(s + 1/\tau)^2} \quad (3.26)$$

$$\alpha(t) = \frac{t}{\tau^2} \exp \left(-\frac{t}{\tau} \right) \quad (3.27)$$

となる。若干の係数の違いはあるが、同じ形の関数が導出された。

3.4 動力学モデル

3.4.1 チャネル動態の動力学的表現

指数関数型シナプスとモデルの振る舞いはほぼ同一だが、式の構成が少し異なるモデルとして**動力学モデル** (Kinetic model, または Markov kinetic model) がある (Destexhe et al., 1994)。動力学モデルは HH モデルのゲート変数の式と類似した式で表される。このモデルではチャネルが開いた状態 (Open) と閉じた状態 (Close), および神経伝達物質 (neurotransmitter) の放出状態 (T) の 2 つの要素に関する状態がある。また、閉 → 開の反応速度を α , 開 → 閉の反応速度を β とする。このとき、これらを表す状態遷移の式は次のようにある。



ここで、シナプス動態を r とすると

$$\frac{dr}{dt} = \alpha T(1 - r) - \beta r \quad (3.29)$$

となる。ただし、 T はシナプス前細胞が発火したときにインパルス的に 1だけ増加するとする。また、 α, β は速度なので、時定数の逆数であることに注意しよう。 $\alpha = 2000\text{ms}^{-1}$, $\beta = 200\text{ms}^{-1}$ とすると、シナプス動態は次のようになる。

```
using PyPlot
rc("axes.spines", top=false, right=false)
```

```
dt = 1e-4 # タイムステップ (sec)
α, β = 1/5e-4, 1/5e-3
T = 0.05 # シミュレーション時間 (sec)
nt = Int(T/dt) # シミュレーションの総ステップ

r = zeros(nt)

for t in 1:nt-1
    spike = ifelse(t == 1, 1, 0)
    r[t+1] = r[t] + dt * (α*spike*(1-r[t]) - β*r[t])
end
```

```
time = (1:nt)*dt
fig, ax = subplots(figsize=(3, 2))
ax.plot(time, r)
ax.set_xlabel("Time (s)"); ax.set_ylabel("Post-synaptic\ncurrent (pA)")
fig.tight_layout()
```

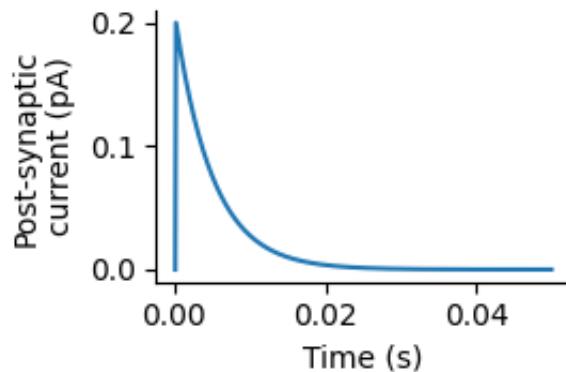


図 3.2 cell003.png

3.4.2 Hodgkin-Huxley モデルにおけるシナプスモデル

これまで明示的にスパイクの発生が表現されたモデルを用いてきたが、HH モデルでは単なる膜電位の変数があるのみである。ここでは前述した動力学的モデルを用いて HH モデルにおけるシナプス動態の記述を行う (Batista et al., 2014; Destexhe et al., 1994)。 r_j を j 番目のニューロンの pre-synaptic dynamics とすると、 r_j は次式に従う。

$$\frac{dr_j}{dt} = \left(\frac{1}{\tau_r} - \frac{1}{\tau_d} \right) \frac{1 - r_j}{1 + \exp(-V_j + V_0)} - \frac{r_j}{\tau_d} \quad (3.30)$$

ただし、時定数 $\tau_r = 0.5, \tau_d = 8$ (ms), 反転電位 $V_0 = -20$ (mV) とする。前節で既に r の描画は行ったが、パルス波を印加した場合の挙動を確認する。

```
using Parameters: @unpack # or using UnPack

abstract type Layer end
abstract type Neuron <: Layer end
abstract type SpikeNeuron <: Neuron end

abstract type Synapse <: Layer end

@kwdef struct HHParameter{FT}
    Cm::FT = 1 # 膜容量(uF/cm^2)
    gNa::FT = 120; gK::FT = 36; gL::FT = 0.3 # Na+, K+, leak の最大コンダクタンス(mS/cm^2)
    ENa::FT = 50; EK::FT = -77; EL::FT = -54 # Na+, K+, leak の平衡電位(mV)
end

@kwdef mutable struct HH{FT} <: SpikeNeuron
    num_neurons::UInt16
    dt::FT = 1e-3
    param::HHParameter = HHParameter{FT}()
    v)::Vector{FT} = fill(-65, num_neurons)
    m)::Vector{FT} = fill(0.05, num_neurons)
    h)::Vector{FT} = fill(0.6, num_neurons)
    n)::Vector{FT} = fill(0.32, num_neurons)
end

@kwdef struct HHKineticSynapseParameter{FT}
    tr::FT = 0.5; td::FT = 8 # ms
    tr^-1::FT = 1/tr; td^-1::FT = 1/td
    v0::FT = -20 # mV
end

@kwdef mutable struct HHKineticSynapse{FT} <: Synapse
    num_neurons::UInt16
    dt::FT = 1e-3
    param::HHKineticSynapseParameter = HHKineticSynapseParameter{FT}()
    r)::Vector{FT} = zeros(num_neurons)
end
```

```

function update!(neuron::HH, x::Vector)
    @unpack num_neurons, dt, v, m, h, n = neuron
    @unpack Cm, gNa, gK, gL, ENa, EK, EL = neuron.param
    @inbounds for i = 1:num_neurons
        m[i] += dt * ((0.1(v[i]+40)/(1 - exp(-0.1(v[i]+40))))*(1 - m[i]) - -
                      4exp(-(v[i]+65) / 18)*m[i])
        h[i] += dt * ((0.07exp(-0.05(v[i]+65)))*(1 - h[i]) - 1/(1 + -
                      exp(-0.1(v[i]+35)))*h[i])
        n[i] += dt * ((0.01(v[i]+55)/(1 - exp(-0.1(v[i]+55))))*(1 - n[i]) - -
                      (0.125exp(-0.0125(v[i]+65)))*n[i])
        v[i] += dt / Cm * (x[i] - gNa * m[i]^3 * h[i] * (v[i] - ENa) - gK * n[i]^4 * -
                           (v[i] - EK) -gL * (v[i] - EL))
    end
    return v
end

function update!(synapse::HHKineticSynapse, v::Vector)
    @unpack num_neurons, dt, r = synapse
    @unpack tr-1, td-1, v0 = synapse.param
    @inbounds for i = 1:num_neurons
        r[i] += dt * ((tr-1 - td-1) * (1 - r[i])/(1 + exp(-v[i] + v0)) - r[i] * td-1)
    end
    return r
end

(layer::Layer)(x) = update!(layer, x)

```

シミュレーションを実行する。

```

T = 50 # ms
dt = 0.01f0 # ms
nt = Int32(T/dt) # number of timesteps
num_neurons = 1 # ニューロンの数

# 入力刺激
time = Array{Float32}(1:nt)*dt
Ie = repeat(5*((time .> 10) - (time .> 15)), 1, num_neurons) # injection current

# 記録用
varr, rarr = zeros(Float32, nt, num_neurons), zeros(Float32, nt, num_neurons)

# modelの定義
hh_neurons = HH{Float32}(num_neurons=num_neurons, dt=dt)
hh_synapse = HHKineticSynapse{Float32}(num_neurons=num_neurons, dt=dt)

# simulation
@time for t = 1:nt
    v = hh_neurons(Ie[t, :])
    r = hh_synapse(v)
    varr[t, :], rarr[t, :] = v, r
end

```

結果を描画する。

```

fig, axes = subplots(3, 1, figsize=(5,3.5), sharex="all", height_ratios=[2, 2, 1])
axes[1].plot(time, varr[:, 1]); axes[1].set_ylabel("Membrane\n potential (mV)");
    axes[1].set_xlim(0, 50)
axes[2].plot(time, rarr[:, 1]); axes[2].set_ylabel("Pre-synaptic\n dynamics")
axes[3].plot(time, Ie[:, 1]); axes[3].set_xlabel("Times (ms)");
    axes[3].set_ylabel("Injection\n current (nA)")
fig.align_labels()
fig.tight_layout()

```

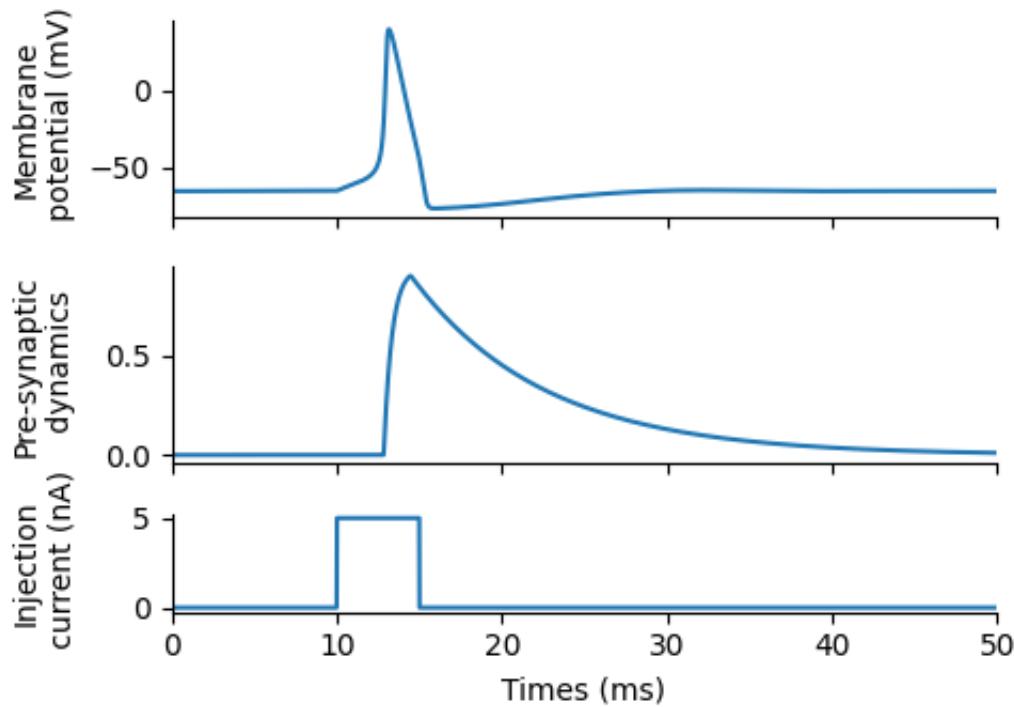


図 3.3 cell009.png

3.5 シナプス入力の重みづけ

ここまででは、シナプス前細胞と後細胞がそれぞれ1つずつである場合について考えていたが、実際には多数の細胞がネットワークを作っている。また、それぞれの入力は均等ではなく、異なるシナプス強度 (Synaptic strength) を持つ。この場合のシナプス入力の計算について述べておく。シナプス前細胞が N_{pre} 個、シナプス後細胞が N_{post} 個あるとする。このときシナプス前過程に注目したシナプス動態を $s_{\text{syn}} \in \mathbb{R}^{N_{\text{pre}}}$ 、シナプス後細胞の入力電流を $I_{\text{syn}} \in \mathbb{R}^{N_{\text{post}}}$ 、シナプス結合強度の行列を $W \in \mathbb{R}^{N_{\text{post}} \times N_{\text{pre}}}$

とすると, Current-based の場合は

$$\mathbf{I}_{\text{syn}}(t) = W \mathbf{s}_{\text{syn}} \quad (3.31)$$

となる. ただし, シナプス強度にシナプス効率が含まれるとした. また, Conductance-based の場合はシナプス後細胞の膜電位を $\mathbf{V}_m \in \mathbb{R}^{N_{\text{post}}}$ として,

$$\mathbf{I}_{\text{syn}}(t) = (\mathbf{V}_{\text{syn}} - \mathbf{V}_m(t)) \odot W \mathbf{s}_{\text{syn}} \quad (3.32)$$

となる. ただし, \odot は Hadamard 積である. これらの式は順序を入れ替えることも可能である. シナプス前細胞でスパイクが生じたことを表すベクトルを $\delta_{t,t_{\text{spike}}} \in \mathbb{R}^{N_{\text{pre}}}$ とする. ただし, t_{spike} は各ニューロンにおいてスパイクが生じた時刻である. \mathbf{s}_{syn} は $\delta_{t,t_{\text{spike}}}$ の関数であり, $\mathbf{s}_{\text{syn}}(\delta_{t,t_{\text{spike}}})$ と表せる. このときシナプス後過程に注目したシナプス動態を $\mathbf{s}'_{\text{syn}} \in \mathbb{R}^{N_{\text{post}}}$ とすると, Current-based の場合は

$$\mathbf{I}_{\text{syn}}(t) = \mathbf{s}'_{\text{syn}}(W \delta_{t,t_{\text{spike}}}) \quad (3.33)$$

Conductance-based の場合は

$$\mathbf{I}_{\text{syn}}(t) = (\mathbf{V}_{\text{syn}} - \mathbf{V}_m(t)) \odot \mathbf{s}'_{\text{syn}}(W \delta_{t,t_{\text{spike}}}) \quad (3.34)$$

と表すことができる. シナプス動態を前過程か後過程のどちらに注目したものとするかは, 実装によって様々である. シナプス入力の計算における中間の値を学習に用いるということもあるため, 単なる計算量の観点だけではどちらを選ぶかは決めることができない(計算量だけならシナプス変数に先に重み行列をかけた方がよい場合が多い). 実装の中で異なるのは計算順序と保持するベクトルの要素数である. 同じ実装の中で 2 つとも用いる場合もあるので注意してほしい.

3.6 動的シナプス

シナプス前活動に応じてシナプス伝達効率 (synaptic efficacy) が動的に変化する性質を短期的シナプス可塑性 (Short-term synaptic plasticity; STSP) といい, このような性質を持つシナプスを動的シナプス (dynamical synapses) と呼ぶ. シナプス伝達効率が減衰する現象を短期抑圧 (short-term depression; STD), 増強する現象を短期促通 (short-term facilitation; STF) という. さらにそれに対応するシナプスを減衰シナプス, 増強シナプスという. ここでは (Mongillo et al., 2008) および (Orhan and Ma, 2019) で用いられている定式化を使用する.

$$\frac{dx(t)}{dt} = \frac{1 - x(t)}{\tau_x} - u(t)x(t)r(t)\Delta t \quad (3.35)$$

$$\frac{du(t)}{dt} = \frac{U - u(t)}{\tau_u} + U(1 - u(t))r(t)\Delta t \quad (3.36)$$

ただし, x を利用可能な神経伝達物質の量, u を利用されている神経伝達物質の量 (the neurotransmitter utilization), τ_x は神経伝達物質の時定数, τ_u は utilization, U は increment, Δt を時間幅とする. ここで

は $\tau_x = (200 \text{ ms}/1,500 \text{ ms}; \text{facilitating/depressing})$, $\tau_u = (1,500 \text{ ms}/200 \text{ ms}; \text{facilitating/depressing})$, $U = (0.15/0.45; \text{facilitating/depressing})$, $\Delta t = 10\text{ms}$ とする。

```

using PyPlot
rc("axes.spines", top=false, right=false)

# ms
Uf, τfx, τfu = 0.15, 200, 1500
Ud, τdx, τdu = 0.45, 1500, 200
dt = 1
T = 4000
tarray = 1:dt:T;
nt = Int(T/dt);
s = zeros(nt) # stimuli
s[500:150:2000] .= 1;
s[2500:200:3000] .= 1;

# short-term synaptic plasticity
function stsp(dt, T, s, U, τx, τu, τs=30)
    nt = Int(T/dt);
    αx, αu, αs = dt/τx, dt/τu, dt/τs # 時定数を減衰率に変換

    u, x, r = zeros(nt), zeros(nt), zeros(nt)
    u[1], x[1] = U, 1

    for t in 1:nt-1
        x[t+1] = x[t] + αx*(1-x[t]) - u[t]*x[t]*s[t]
        u[t+1] = u[t] + αu*(U-u[t]) + U*(1-u[t])*s[t]
        x[t+1], u[t+1] = clamp.([x[t+1], u[t+1]], 0, 1) # for numerical stability
        r[t+1] = (1-αs)*r[t] + u[t]*x[t]*s[t]/U
    end
    return u, x, r
end;

# simulation
uf, xf, rf = stsp(dt, T, s, Uf, τfx, τfu)
ud, xd, rd = stsp(dt, T, s, Ud, τdx, τdu);

# compute synaptic efficacy
xuf = uf .* xf / Uf
xud = ud .* xd / Ud;

fig, axes = subplots(4, 2, figsize=(6, 5), sharex="col", height_ratios=[1, 2, 2, 2])
for i in 1:2
    axes[1,i].plot(tarray, s, "k");
    axes[2,i].set_ylim(0, 1.1);
end

```

```

axes[1,1].set_title("Facilitating synapse"); axes[1,1].set_ylabel("Presynaptic\nspike");
axes[1,2].set_title("Depressing synapse");
axes[2,1].plot(tarray, uf); axes[2,1].plot(tarray, xf, "tab:red");
axes[2,1].set_ylabel("Synaptic variables");
axes[2,2].plot(tarray, ud, label=L"$u$"); axes[2,2].plot(tarray, xd, "tab:red", label=L"$x$");
axes[2,2].legend(ncol=2)
axes[3,1].plot(tarray, xuf, "k"); axes[3,1].set_ylabel("Synaptic\n efficacy")
axes[3,2].plot(tarray, xud, "k");
axes[4,1].plot(tarray, rf, "k"); axes[4,1].set_xlabel("Time (ms)");
axes[4,1].set_ylabel("Postsynaptic current")
axes[4,2].plot(tarray, rd, "k"); axes[4,2].set_xlabel("Time (ms)")
fig.align_labels()
fig.tight_layout()

```

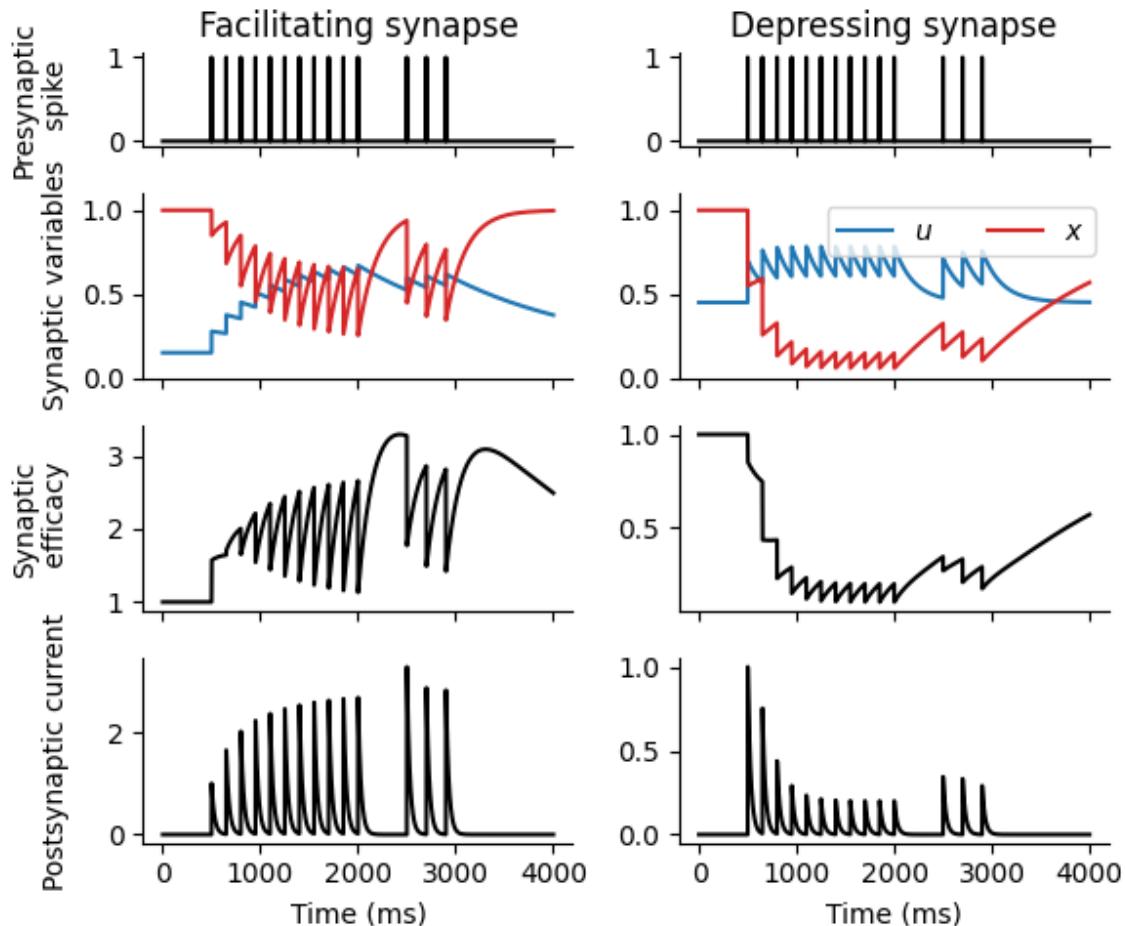


図 3.4 cell005.png

参考文献

- Batista, C. A. S. et al. (2014). “Dynamic range in small-world networks of Hodgkin–Huxley neurons with chemical synapses”. *Physica A: Statistical Mechanics and its Applications* 410, pp. 628–640.
- Cavallari, S., Panzeri, S., and Mazzoni, A. (2014). “Comparison of the dynamics of neural interactions between current-based and conductance-based integrate-and-fire recurrent networks”. *Front. Neural Circuits* 8, p. 12.
- Destexhe, A., Mainen, Z. F., and Sejnowski, T. J. (1994). “An Efficient Method for Computing Synaptic Conductances Based on a Kinetic Model of Receptor Binding”. *Neural Comput.* 6.1, pp. 14–18.
- Mongillo, G., Barak, O., and Tsodyks, M. (2008). “Synaptic Theory of Working Memory”. *Science* 319.5869, pp. 1543–1546.
- Orhan, A. E. and Ma, W. J. (2019). “A diverse range of factors affect the nature of neural representations underlying short-term memory”. *Nat. Neurosci.* 22.2, pp. 275–283.
- Rall, W. (1967). “Distinguishing theoretical synaptic potentials computed for different somadendritic distributions of synaptic input”. *J. Neurophysiol.* 30.5, pp. 1138–1168.

第4章

神経回路網の演算処理

4.1 ゲイン調節と四則演算

(Goldwyn et al., 2018) を実装. 神経演算 (neuronal arithmetic; (Angus Silver, 2010)) のモデル.

```
using Base: @kwdef
using Parameters: @unpack # or using UnPack
using PyPlot, ProgressMeter, Distributions
rc("axes.spines", top=false, right=false)

@kwdef struct HHIAParameter{FT}
    Cm::FT = 1 # 膜容量(uF/cm^2)
    gNa::FT = 37; gK::FT = 45; gA::FT = 20; gL::FT = 1 # Na, K, Ka, leakの最大コンダクタンス(mS/cm^2)
    ENa::FT = 55; EK::FT = -80; EL::FT = -70 #Na, K, leakの平衡電位(mV)
    gExc::FT = 0.5; gInh::FT = 1
    VExc::FT = 0; VInh::FT = -85
    βExc::FT = 0.2; βInh::FT = 0.18
    tr::FT = 0.5; td::FT = 8 # ms
    γ1::FT = 1/td; γ2::FT = 1/tr - 1/td
    v0::FT = -20 # mV
end

@kwdef mutable struct HHIA{FT}
    param::HHIAParameter = HHIAParameter{FT}()
    N::Int
    v::Vector{FT} = fill(-70, N); r::Vector{FT} = zeros(N)
    n::Vector{FT} = fill(1/(1 + exp(-(−70 + 32)/8)), N)
    a::Vector{FT} = fill(1/(1 + exp(-(−70 + 50)/20)), N)
    b::Vector{FT} = fill(1/(1 + exp((−70 + 70)/6)), N)
    sExc::Vector{FT} = zeros(N); sInh::Vector{FT} = zeros(N)
end

function update!(variable::HHIA, param::HHIAParameter, spikesExc::Vector, spikesInh::Vector, dt)
```

```

@unpack N, v, n, a, b, r, sExc, sInh = variable
@unpack Cm, gNa, gK, gL, gA, ENa, EK, EL, gExc, gInh, VExc, VInh, betaExc, betaInh, 
    y1, y2, v0 = param
@inbounds for i = 1:N
    m, h = 1 / (1 + exp(-(v[i]+30)/15)), 1 - n[i]

    n[i] += dt * 0.75(1/(1 + exp(-0.125(v[i] + 32))) - n[i]) / (1 + 100 / (1 +
        exp((v[i] + 80)/26)))
    a[i] += dt * 0.5(1/(1 + exp(-0.05(v[i] + 50))) - a[i])
    b[i] += dt * (1.0/(1 + exp((v[i] + 70)/6)) - b[i]) / 150

    sExc[i] += -sExc[i] * betaExc*dt + spikesExc[i]
    sInh[i] += -sInh[i] * betaInh*dt + spikesInh[i]
    IExc = gExc * sExc[i] * (v[i] - VExc)
    IIInh = gInh * sInh[i] * (v[i] - VInh)

    IL = gL * (v[i] - EL)
    IK = gK * n[i]^4 * (v[i] - EK)
    IA = gA * a[i]^3 * b[i] * (v[i] - EK)
    INa = gNa * m^3 * h * (v[i] - ENa)

    v[i] += dt/Cm * -(IL + IK + IA + INa + IExc + IIInh)
    r[i] += dt * (y2 * (1.0 - r[i])/(1.0 + exp(-v[i] + v0)) - r[i] * y1)
end
end

```

```

function GammaSpike(T, dt, n_neurons, fr, k)
    nt = Int(T/dt) # number of timesteps
    theta = 1/(k*(fr*dt*1e-3)) # fr = 1/(k*theta)

    isi = rand(Gamma(k, theta), Int(round(nt*1.5/fr)), n_neurons)
    spike_time = cumsum(isi, dims=1) # ISIを累積
    spike_time[spike_time .> nt - 1] .= 1 # ntを超える場合を1に
    spike_time = round.(Int, spike_time) # float to int
    spikes = zeros(Bool, nt, n_neurons) # スパイク記録変数

    for i=1:n_neurons
        spikes[spike_time[:, i], i] .= 1
    end

    spikes[1] = 0 # (spike_time=1)の発火を削除
    return spikes
end

```

```

function FIcurve(neurons, spikesExc, spikesInh, T=5000, dt=0.01)
    nt = Int(T/dt) # number of timesteps
    varr = zeros(Float32, nt, neurons.N)

    @showprogress for t = 1:nt
        update!(neurons, neurons.param, spikesExc[t, :], spikesInh[t, :], dt)
        varr[t, :] = neurons.v
    end

```

```

spike = (varr[1:nt-1, :] .< 0) .& (varr[2:nt, :] .> 0)
output_spikes = sum(spike, dims=1) / T*1e3
input_spikes = sum(spikesExc, dims=1) / T*1e3
return input_spikes, output_spikes
end

```

```

T, dt = 50000, 5e-2 # ms
nt = Int(T/dt)
N = 100
maxfrExc = 80; frInh = [0, 50];

```

```

function HHIAFIcurve_multi(gA, T, dt, N, maxfrExc, frInh)
    nInh = size(frInh)[1]
    input_spikes_arr, output_spikes_arr = zeros(nInh, N), zeros(nInh, N)
    nt = Int(T/dt) # number of timesteps
    frExc = rand(N) * maxfrExc
    spikesExc = zeros(Int, nt, N)
    for j = 1:N
        spikesExc[:, j] = rand(nt) .< frExc[j]*dt*1e-3
    end
    for i=1:nInh
        spikesInh = (frInh[i] == 0) ? zeros(Int, nt, N) : GammaSpike(T, dt, N, -
            frInh[i], 12)
        neurons = HHIA{Float32}(N=N, param=HHIAParameter{Float32}(gA=gA)) # ←
            modelの定義
        input_spikes_arr[i, :], output_spikes_arr[i, :] = FIcurve(neurons, -
            spikesExc, spikesInh, T, dt)
    end
    return input_spikes_arr, output_spikes_arr
end

```

```

input_spikes1, output_spikes1 = HHIAFIcurve_multi(20, T, dt, N, maxfrExc, frInh);
input_spikes2, output_spikes2 = HHIAFIcurve_multi(40, T, dt, N, maxfrExc, frInh);

```

```

figure(figsize=(10, 4))
subplot(1,3,1); title(L"Divisive inhibition (g$_A=20$)")
scatter(input_spikes1[1, :], output_spikes1[1, :], facecolor="white", -
    edgecolors="tab:red", label="no inhibition")
scatter(input_spikes1[2, :], output_spikes1[2, :], alpha=0.5, color="tab:red", -
    label="inhibition")
xlim(0, ); ylim(0, ); xlabel("Input spikes/s"); ylabel("Output spikes/s"); legend()
subplot(1,3,2); title(L"Subtractive inhibition (g$_A=40$)")
scatter(input_spikes2[1, :], output_spikes2[1, :], facecolor="white", -
    edgecolors="tab:blue", label="no inhibition")
scatter(input_spikes2[2, :], output_spikes2[2, :], alpha=0.5, color="tab:blue", -
    label="inhibition")
xlim(0, ); ylim(0, ); xlabel("Input spikes/s"); ylabel("Output spikes/s"); legend()
subplot(1,3,3);

```

```

scatter(output_spikes1[1, :], output_spikes1[2, :], alpha=0.5, color="tab:red", +
       label=L"g$_A$=20$")
scatter(output_spikes2[1, :], output_spikes2[2, :], alpha=0.5, color="tab:blue", +
       label=L"g$_A$=40$")
xlim(0, ); ylim(0, ); xlabel("Output spike/s\n No Inhibition"); ylabel("Output spike/s\n with Inhibition"); legend()
tight_layout()

```

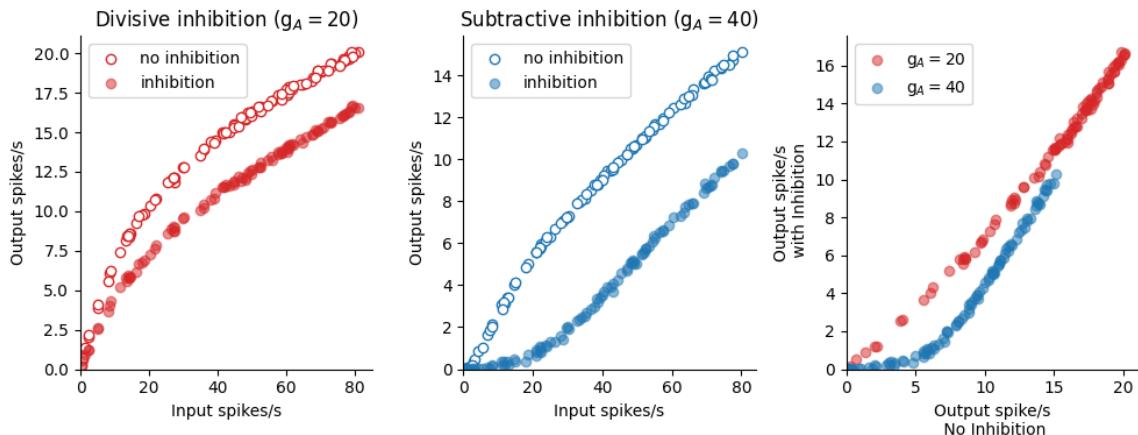


図 4.1 cell009.png

参考文献

- Angus Silver, R. (2010). “Neuronal arithmetic”. *Nat. Rev. Neurosci.* 11.7, pp. 474–489.
 Goldwyn, J. H. et al. (2018). “Gain control with A-type potassium current: IA as a switch between divisive and subtractive inhibition”. *PLoS Comput. Biol.* 14.7, e1006292.

第5章

局所学習則

5.1 Hebb 則と教師なし学習

5.1.1 Hebb 則

神経回路はどのようにして自己組織化するのだろうか。1940年代にカナダの心理学者 Donald O. Hebb により著書”The Organization of Behavior”(Hebb, 1949) で提案された学習則は「細胞 A が反復的または持続的に細胞 B の発火に関与すると、細胞 A が細胞 B を発火させる効率が向上するような成長過程または代謝変化が一方または両方の細胞に起こる」というものであった。すなわち、発火に時間的相関のある細胞間のシナプス結合を強化するという学習則である。これを **Hebb の学習則 (Hebbian learning rule)** あるいは **Hebb 則 (Hebb's rule)** という。Hebb 則は (Hebb 自身ではなく) Shatz により ”cells that fire together wire together” (共に活動する細胞は共に結合する) と韻を踏みながら短く言い換えられている (Shatz, 1992)。

Hebb 則の導出

数式で Hebb 則を表してみよう。 n 個のシナプス前細胞と m 個の後細胞の発火率をそれぞれ $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$ とする。前細胞と後細胞間のシナプス結合強度を表す行列を $\mathbf{W} \in \mathbb{R}^{m \times n}$ とし、 $\mathbf{y} = \mathbf{Wx}$ が成り立つとする。このようなモデルを線形ニューロンモデル (Linear neuron model) という。このとき、Hebb 則は

$$\tau \frac{d\mathbf{W}}{dt} = \phi(\mathbf{y})\varphi(\mathbf{x})^\top \quad (5.1)$$

として表される。ただし、 τ は時定数であり、 $\eta := 1/\tau$ は学習率 (learning rate) と呼ばれる学習の速さを決定するパラメータとなる。 $\varphi(\cdot)$ および $\phi(\cdot)$ は、それぞれシナプス前細胞および後細胞の活動量に応じて重みの変化量を決定する関数である。ただし、 $\varphi(\cdot), \phi(\cdot)$ は基本的に恒等関数に設定される場合が多い。この場合、Hebb 則は $\tau \frac{d\mathbf{W}}{dt} = \mathbf{yx}^\top = (\text{post}) \cdot (\text{pre})^\top$ と簡潔に表現される。この Hebb 則は数学的に導出されたものではないが、特定の目的関数を神経活動及び重みを変化させて最適化するよ

うなネットワークを構築すれば自然に出現する。このようなネットワークを**エネルギーベースモデル (energy-based models)**といい、次章で扱う。エネルギーベースモデルでは、先にエネルギー関数（あるいはコスト関数） \mathcal{E} を定義し、その目的関数を最小化するような神経活動 \mathbf{z} および重み行列 \mathbf{W} のダイナミクスをそれぞれ、

$$\frac{d\mathbf{z}}{dt} \propto -\frac{\partial \mathcal{E}}{\partial \mathbf{z}}, \quad \frac{d\mathbf{W}}{dt} \propto -\frac{\partial \mathcal{E}}{\partial \mathbf{W}} \quad (5.2)$$

として導出する。この手順の逆を行う、すなわち先に神経細胞の活動ダイナミクスを定義し、神経活動で積分することで神経回路のエネルギー関数 \mathcal{E} を導出し、さらに \mathcal{E} を重み行列で微分することで Hebb 則が導出できる (Isomura and Friston, 2020)。Hebb 則の導出を連続時間線形ニューロンモデル $\frac{dy}{dt} = \mathbf{Wx}$ を例にして考えよう。ここで $\frac{\partial \mathcal{E}}{\partial \mathbf{y}} := -\frac{dy}{dt}$ となるようなエネルギー関数 $\mathcal{E}(\mathbf{x}, \mathbf{y}, \mathbf{W})$ を仮定すると、

$$\mathcal{E}(\mathbf{x}, \mathbf{y}, \mathbf{W}) = - \int \mathbf{Wx} dy = -\mathbf{y}^\top \mathbf{Wx} \in \mathbb{R} \quad (5.3)$$

となる。これをさらに \mathbf{W} で微分すると、

$$\frac{\partial \mathcal{E}}{\partial \mathbf{W}} = -\mathbf{yx}^\top \Rightarrow \frac{d\mathbf{W}}{dt} = -\frac{\partial \mathcal{E}}{\partial \mathbf{W}} = \mathbf{yx}^\top \quad (5.4)$$

となり、Hebb 則が導出できる（簡単のため時定数は 1 とした）。

5.1.2 Hebb 則の安定化と LTP/LTD

BCM 則

Hebb 則には問題点があり、シナプス結合強度が際限なく増大するか、0 に近づくこととなってしまう。これを数式で確認しておこう。前細胞と後細胞がそれぞれ 1 つの場合を考える。2 細胞間の結合強度を $w (> 0)$ とし、 $y = wx$ が成り立つとすると、Hebb 則は $\frac{dw}{dt} = \eta yx = \eta x^2 w$ となる。この場合、 $\eta x^2 > 1$ なら $\lim_{t \rightarrow \infty} w = \infty$ 、 $\eta x^2 < 1$ なら $\lim_{t \rightarrow \infty} w = 0$ となる。当然、生理的にシナプス結合強度が無限大となることはあり得ないが、不安定なほど大きくなってしまう可能性があることに違いはない。このため、Hebb 則を安定化させるための修正が必要とされた。Cooper, Liberman, Oja らにより頭文字をとって **CLO 則 (CLO rule)** が提案された (Cooper et al., 1979)。その後、Bienenstock, Cooper, Munro らにより提案された学習則は同様に頭文字をとって **BCM 則 (BCM rule)** と呼ばれている (Bienenstock et al., 1982) (Cooper and Bear, 2012)。 $\mathbf{x} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d, y \in \mathbb{R}$ とし、単一の出力 $y = \mathbf{w}^\top \mathbf{x} = \mathbf{x}^\top \mathbf{w}$ を持つ線形ニューロンを仮定する。重みの更新則は次のようにする。

$$\frac{d\mathbf{w}}{dt} = \eta_w \mathbf{x} \phi(y, \theta_m) \quad (5.5)$$

ここで関数 ϕ は $\phi(y, \theta_m) = y(y - \theta_m)$ などとする。また $\theta_m := \mathbb{E}[y^2]$ は閾値を決定するパラメータ、**修正閾値 (modification threshold)** であり、

$$\frac{d\theta_m}{dt} = \eta_\theta (y^2 - \theta_m) \quad (5.6)$$

として更新される。ToDo: 詳細

```
using PyPlot, Random, Distributions, LinearAlgebra, FFTW
rc("axes.spines", top=false, right=false)
rc("font", family="Meiryo")
```

```
phi(y, theta_m) = y * (y - theta_m);
```

```
y = 0:0.1:2;
theta_m = 1.0
props = Dict("boxstyle" => "round", "facecolor" => "wheat", "alpha" => 0.5)
figure(figsize=(3, 2))
plot(y, 1.5*y, label="Hebb則")
plot(y, phi.(y, theta_m), label="BCM則")
xlim(0,);
annotate(text="", xy=(0.8,0), xytext=(1.2,0), arrowprops=Dict("arrowstyle" => "<->", "color" => "tab:purple"))
axvline(theta_m, linestyle="dashed", color="tab:purple")
axhline(0, linestyle="dashed", color="tab:gray")
xticks([]); yticks([]); xlabel(L"\$y\$ \"*(シナプス後細胞の活動)"")
text(0, 3.5, L"\$\phi(y, \theta_m)\$", ha="center", va="center")
text(2.2, 3, "Hebb則", color="tab:blue", fontsize=10)
text(2.2, 2, "BCM則", color="tab:orange", fontsize=10)
text(0.5, 0.2, L"\">\theta_m", color="tab:purple", fontsize=11)
text(-0.4, -0.3, "LTD", fontsize=11, color="tab:blue", ha="center", va="center", bbox=props);
text(-0.4, 1.8, "LTP", fontsize=11, color="tab:red", ha="center", va="center", bbox=props);
tight_layout()
```

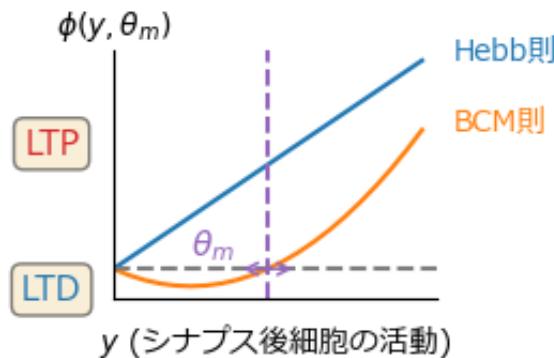


図 5.1 cell004.png

Hebb 則の生理的機序

ここで Hebb 則および BCM 則の生理的基盤について触れておこう。LTP の実験的発見 (Bliss and Lomo, 1973) (Dudek and Bear, 1992) ToDo: 実験的発見の survey

Oja 則

Hebb 則を安定化させる別のアプローチとして、結合強度を正規化するという手法が考えられる。BCM 則と同様に $\mathbf{x} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d, y \in \mathbb{R}$ とし、単一の出力 $y = \mathbf{w}^\top \mathbf{x} = \mathbf{x}^\top \mathbf{w}$ を持つ線形ニューロンを仮定する。 η を学習率とするとき、 $\mathbf{w} \leftarrow \frac{\mathbf{w} + \eta \mathbf{x}y}{\|\mathbf{w} + \eta \mathbf{x}y\|}$ とすれば正規化できる。ここで、 $f(\eta) := \frac{\mathbf{w} + \eta \mathbf{x}y}{\|\mathbf{w} + \eta \mathbf{x}y\|}$ とし、 $\eta = 0$ において Taylor 展開を行うと、

$$f(\eta) \approx f(0) + \eta \left. \frac{df(\eta^*)}{d\eta^*} \right|_{\eta^*=0} + \mathcal{O}(\eta^2) \quad (5.7)$$

$$= \frac{\mathbf{w}}{\|\mathbf{w}\|} + \eta \left(\frac{\mathbf{x}y}{\|\mathbf{w}\|} - \frac{y^2 \mathbf{w}}{\|\mathbf{w}\|^3} \right) + \mathcal{O}(\eta^2) \quad (5.8)$$

ここで $\|\mathbf{w}\| = 1$ として、1次近似すれば $f(\eta) \approx \mathbf{w} + \eta (\mathbf{x}y - y^2 \mathbf{w})$ となる。重みの変化が連続的であるとすると、

$$\frac{d\mathbf{w}}{dt} = \eta (\mathbf{x}y - y^2 \mathbf{w}) \quad (5.9)$$

として重みの更新則が得られる。これを Oja 則 (Oja's rule) と呼ぶ (Oja, 1982)。こうして得られた学習則において $\|\mathbf{w}\| \rightarrow 1$ となることを確認しよう。

$$\frac{d\|\mathbf{w}\|^2}{dt} = 2\mathbf{w}^\top \frac{d\mathbf{w}}{dt} = 2\eta y^2 (1 - \|\mathbf{w}\|^2) \quad (5.10)$$

より、 $\frac{d\|\mathbf{w}\|^2}{dt} = 0$ のとき、 $\|\mathbf{w}\| = 1$ となる。

恒常的可塑性

Oja 則は更新時の即時的な正規化から導出されたものであるが、恒常的可塑性 (synaptic scaling) により安定化しているという説がある (Turrigiano, 2008)(Yee et al., 2017)。しかし、この過程は遅すぎるため、Hebb 則の不安定化を安定化するに至らない (Zenke et al., 2017) ToDo: 恒常的可塑性の詳細 Johansen, Joshua P., Lorenzo Diaz-Mataix, Hiroki Hamanaka, Takaaki Ozawa, Edgar Ycu, Jenny Koivumaa, Ashwani Kumar, et al. 2014. “Hebbian and Neuromodulatory Mechanisms Interact to Trigger Associative Memory Formation.” Proceedings of the National Academy of Sciences 111 (51): E5584 – 92.

5.1.3 Hebb 則と主成分分析

Oja 則を用いることで主成分分析 (Principal component analysis; PCA) という処理をニューラルネットワークにおいて実現できる。主成分分析とは- ToDo: 主成分分析の説明

```
d = MvNormal([0,0], [1.0 0.5; 0.5 1.0]) # multivariate normal distribution
N = 300 # sample size
Random.seed!(0) # set seed
X = rand(d, N); # generate toy data
```

```
U, S, V = svd(X*X')
```

```
figure(figsize=(3,3))
scatter(X[1,:], X[2,:], alpha=0.5)
arrow(0, 0, V[1,1], V[2,1], head_width=0.2, color="tab:red", -
      length_includes_head=true, label="PC1")
arrow(0, 0, V[1,2], V[2,2], head_width=0.2, color="tab:orange", -
      length_includes_head=true, label="PC2")
θc = 0:1e-2:2pi
plot(cos.(θc), sin.(θc), "k--", alpha=0.8)
xlabel(L"\$X_1\$"); ylabel(L"\$X_2\$")
legend(); tight_layout()
```

Oja 則による PCA の実行

ここで Oja 則が主成分分析を実行できることを示す。重みの変化量の期待値を取る。

$$\frac{d\mathbf{w}}{dt} = \eta (\mathbf{x}\mathbf{y} - \mathbf{y}^2\mathbf{w}) = \eta (\mathbf{x}\mathbf{x}^\top\mathbf{w} - [\mathbf{w}^\top \mathbf{x}\mathbf{x}^\top \mathbf{w}] \mathbf{w}) \quad (5.11)$$

$$\mathbb{E} \left[\frac{d\mathbf{w}}{dt} \right] = \eta (\mathbf{C}\mathbf{w} - [\mathbf{w}^\top \mathbf{C}\mathbf{w}] \mathbf{w}) \quad (5.12)$$

$\mathbf{C} := \mathbb{E}[\mathbf{x}\mathbf{x}^\top] \in \mathbb{R}^{d \times d}$ とする。 \mathbf{x} の平均が 0 の場合、 \mathbf{C} は分散共分散行列である。 $\mathbb{E} \left[\frac{d\mathbf{w}}{dt} \right] = 0$ となる \mathbf{w} が収束する固定点 (fixed point) では次の式が成り立つ。

$$\mathbf{C}\mathbf{w} = \lambda\mathbf{w} \quad (5.13)$$

これは固有値問題であり、 $\lambda := \mathbf{w}^\top \mathbf{C}\mathbf{w}$ は固有値、 \mathbf{w} は固有ベクトル (eigen vector) になる。ここでサンプルサイズを n とし、 $\mathbf{X} \in \mathbb{R}^{d \times n}, \mathbf{y} = \mathbf{X}^\top \mathbf{w} \in \mathbb{R}^n$ とする。標本平均で近似して $\mathbf{C} \simeq \mathbf{X}\mathbf{X}^\top$ とする。この場合、

$$\mathbb{E} \left[\frac{d\mathbf{w}}{dt} \right] \simeq \eta (\mathbf{X}\mathbf{X}^\top\mathbf{w} - [\mathbf{w}^\top \mathbf{X}\mathbf{X}^\top \mathbf{w}] \mathbf{w}) \quad (5.14)$$

$$= \eta (\mathbf{X}\mathbf{y} - [\mathbf{y}^\top \mathbf{y}] \mathbf{w}) \quad (5.15)$$

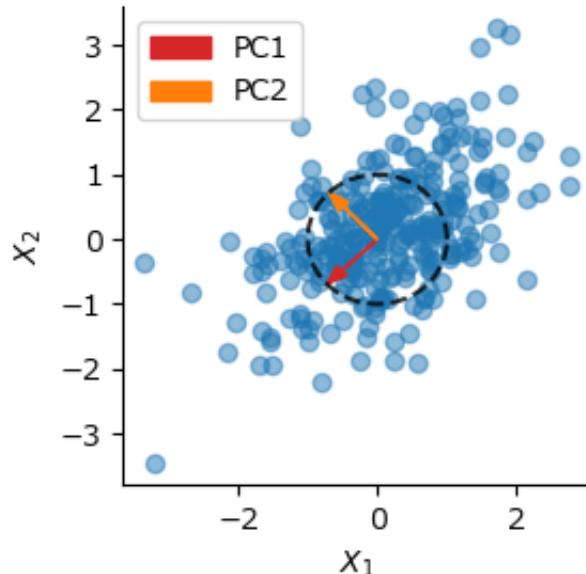


図 5.2 cell011.png

となる。

```
w = randn(2) # initialize weight
w ./= sqrt.(sum(w.^2)) # L2 normalize
initw = copy(w) # save initial weight
η = 1e-3 # learning rate
for _ in 1:200
    y = X' * w
    w += η * (X * y - y' * y * w) # Oja's rule
end
```

```
figure(figsize=(3,3))
scatter(X[1,:], X[2,:], alpha=0.5)
arrow(0,0,initw[1],initw[2], head_width=0.2, color="k", length_includes_head=true, ~
      label=L"Init. \$w$")
arrow(0,0,w[1],w[2], head_width=0.2, color="tab:red", length_includes_head=true, ~
      label=L"Opt. \$w$")
plot(cos.(θc), sin.(θc), "k--", alpha=0.8)
xlabel(L"\$X\_1\$"); ylabel(L"\$X\_2\$")
tight_layout()
legend(); tight_layout()
```

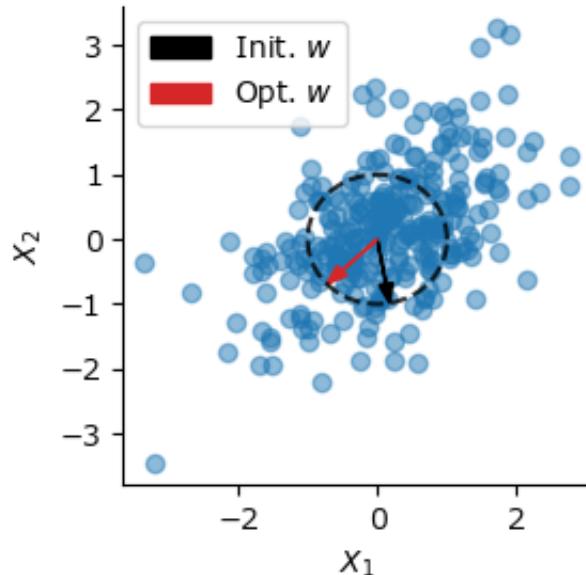


図 5.3 cell014.png

後のために Oja 則においてネットワークが q 個の複数出力を持つ場合を考えよう。重み行列を $\mathbf{W} \in \mathbb{R}^{q \times d}$, 出力を $\mathbf{y} = \mathbf{Wx} \in \mathbb{R}^q$, $\mathbf{Y} = \mathbf{WX} \in \mathbb{R}^{q \times n}$ とする。この場合の更新則は

$$\frac{d\mathbf{W}}{dt} = \eta (\mathbf{yx}^\top - \text{Diag} [\mathbf{yy}^\top] \mathbf{W}) \quad (5.16)$$

となる。ただし、 $\text{Diag}(\cdot)$ は行列の対角成分からなる対角行列を生み出す作用素である。

Sanger 則

Oja 則に複数の出力を持たせた場合であっても、出力が直交しないため、PCA の第 1 主成分しか求めることができない。Sanger 則 (Sanger's rule), あるいは一般化 Hebb 則 (generalized Hebbian algorithm; GHA) は、Oja 則に Gram - Schmidt の正規直交化法 (Gram - Schmidt orthonormalization) を組み合わせた学習則であり、次式で表される。

$$\frac{d\mathbf{W}}{dt} = \eta (\mathbf{yx}^\top - \text{LT} [\mathbf{yy}^\top] \mathbf{W}) \quad (5.17)$$

$\text{LT}(\cdot)$ は行列の対角成分より上側の要素を 0 にした下三角行列 (lower triangular matrix) を作り出す作用素である。Sanger 則を用いれば PCA の第 2 主成分以降も求めることができる。

```
W = randn(2, 2) # initialize weight
W ./= sqrt.(sum(W.^2, dims=2)) # normalize
```

```

initW = copy(W) # save initial weight
for _ in 1:200
    Y = W * X
    W += η * (Y * X' - LowerTriangular(Y * Y') * W) # Sanger's rule
end

```

```

figure(figsize=(3,3))
scatter(X[1,:], X[2,:], alpha=0.5)
arrow(0, 0, W[1,1], W[1,2], head_width=0.2, color="tab:red", -
      length_includes_head=true, label=L"$w_1$")
arrow(0, 0, W[2,1], W[2,2], head_width=0.2, color="tab:orange", -
      length_includes_head=true, label=L"$w_2$")
plot(cos.(θc), sin.(θc), "k--", alpha=0.8)
xlabel(L"$X_1$"); ylabel(L"$X_2$")
legend(); tight_layout()

```

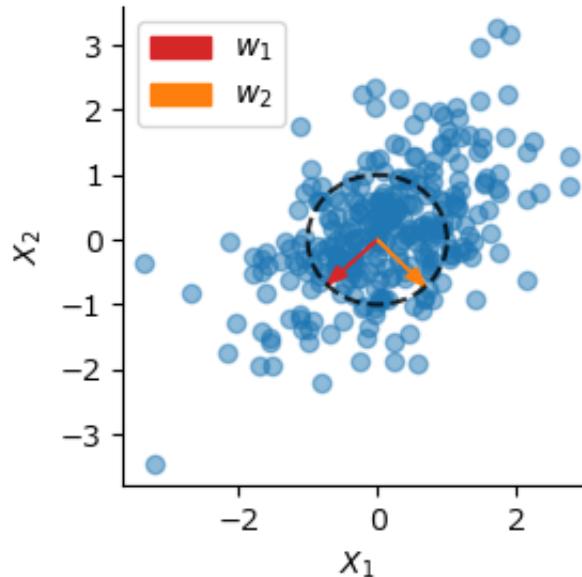


図 5.4 cell018.png

Oja 則, Sanger 則をまとめて一つの関数にしておこう. `identity()` は恒等関数である.

```

function HebbianPCA(X; n_components=10, η=1e-6, maxiter=200, func=identity, -
                     orthogonal=true)
    # X : n x m -> Y : n_components x m
    n = size(X)[1]
    η /= n
    X = (X .- mean(X, dims=2)) ./ std(X, dims=2) # normalization

```

```

Y = nothing
W = randn(n_components, n) # initialize weight
W ./= sqrt(sum(W.^2, dims=2)) # normalization
for _ in 1:maxiter
    Y = func.(W * X)
    if orthogonal
        W .+= η * (Y * X' - LowerTriangular(Y * Y') * W) # Sanger's rule
    else
        W .+= η * (Y * X' - Diagonal(Y * Y') * W) # Oja's rule
    end
end
return Y, W
end;

```

5.1.4 非線形 Hebb 学習

出力 y に非線形関数 $g(\cdot)$ を適用し, $y \rightarrow g(y)$ として置き換えることで非線形 Hebb 学習となる (Oja, 1997)(Brito and Gerstner, 2016). 関数 HebbianPCA の `func` 引数に非線形関数を渡すことで実現できる。ToDo: 詳細

非負主成分分析によるグリッドパターンの創発

内側嗅内皮質 (MEC) にあるグリッド細胞 (grid cells) は六角形格子状の発火パターンにより自己位置等を符号化するのに貢献している。この発火パターンを生み出すモデルは多数あるが、場所細胞 (place cells) の発火パターンを非負主成分分析 (nonnegative principal component analysis) で次元削減するとグリッド細胞のパターンが生まれるというモデルがある (Dordek et al., 2016)。非線形 Hebb 学習を用いてこのモデルを実装しよう。なお、同様のことは非負値行列因子分解 (NMF: nonnegative matrix factorization) でも可能である。

場所細胞の発火パターン まず、訓練データとなる場所細胞の発火パターンを人工的に作成する。場所細胞の発火パターンは Difference of Gaussians (DoG) で近似する。DoG は大きさの異なる 2 つのガウス関数の差分を取った関数であり、画像に適応すれば band-pass フィルタとして機能する。また、DoG は網膜神経節細胞等の受容野の ON 中心 OFF 周辺型受容野のモデルとしても用いられる。受容野中央では活動が大きく、その周辺では活動が抑制される、という特性を持つ。2 次元のガウス関数と DoG 関数を実装する。

```

function gaussian2d(center, width, height, step, sigma, scale=1)
    x, y = range(-width/2, width/2, length=step), range(-height/2, height/2, -
        length=step)
    f(x,y) = exp(-((x-center[1])^2 + (y-center[2])^2) / (2.0*scale*(sigma^2)))
    gau = f.(x', y)
    return gau ./ sum(gau)
end

```

```

function DoG(center, width=2.2, height=2.2, step=55, sigma=0.12, surround_scale=2)
    g1 = gaussian2d(center, width, height, step, sigma)
    g2 = gaussian2d(center, width, height, step, sigma, surround_scale)
    return g1 - g2
end

```

モデルのパラメータを設定する。

```

sqNp = 32          # 場所細胞の数の平方根: Np=sqNp^2
Ng = 9             # 格子細胞の数
sigma = 0.12       # 場所細胞のtuning curveの幅 [m]
surround_scale = 2 # DoGの $\sigma^2$ の比率
box_width = 2.2    # 箱の横幅 [m]
box_height = 2.2   # 箱の縦幅 [m]
step = 45;         # 空間位置の離散化数

```

先にガウス関数と DoG 関数がどのような見た目になるか確認しよう。

```

c_eg = zeros(2)
gau_eg = gaussian2d(c_eg, box_width, box_height, step, sigma)
dog_eg = DoG(c_eg, box_width, box_height, step, sigma, surround_scale);

```

```

fig, ax = subplots(2,2,figsize=(4,4),sharex="all", sharey="row")
ax[1,1].set_title("Gaussian")
ax[1,1].imshow(gau_eg, cmap="turbo", extent=(-box_width/2, box_width/2, -box_height/2, box_height/2))
ax[1,1].set_ylabel("y [m]")
ax[1,2].set_title("Difference of\n Gaussians (DoG)")
ax[1,2].imshow(dog_eg, cmap="turbo", extent=(-box_width/2, box_width/2, -box_height/2, box_height/2))
x_pos = range(-box_width/2, box_width/2, length=step)
ax[2,1].plot(x_pos, gau_eg/div(step, 2), :]/maximum(gau_eg))
ax[2,1].set_xlabel("x [m]"; ax[2,1].set_ylabel(L"$y=0$* の形状 (正規化)")
ax[2,2].plot(x_pos, dog_eg/div(step, 2), :]/maximum(dog_eg))
ax[2,2].set_xlabel("x [m]")
tight_layout()

```

場所細胞の活動パターンを生み出す。それぞれの場所受容野の中心は空間を均等に覆うように作成する（一様分布で生み出してもよい）。

```

x_pos = range(-box_width/2, box_width/2, length=sqNp)
y_pos = range(-box_height/2, box_height/2, length=sqNp)
centers = [[i, j] for i in x_pos for j in y_pos]
X_place = hcat([DoG(c, box_width, box_height, step, sigma, surround_scale)[:] for c in centers]);

```

線形 PCA の場合

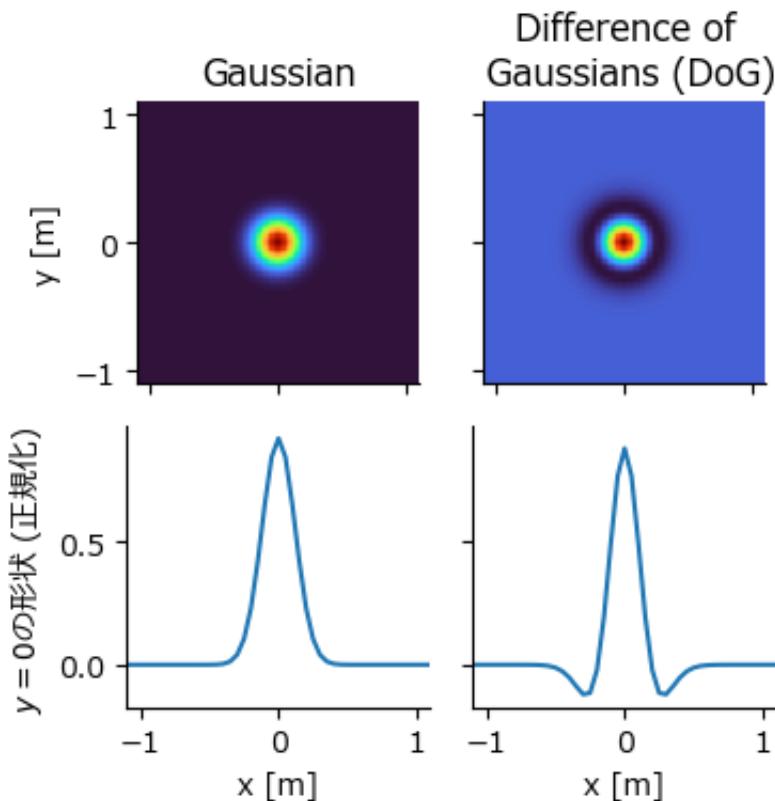


図 5.5 cell029.png

```
@time Y_pca, W_pca = HebbianPCA(X_place, n_components=Ng, η=1e-2, maxiter=5000, ~
    orthogonal=true)
Y_pca = reshape(Y_pca, (Ng, step, step));
```

```
figure(figsize=(3,3.5))
suptitle("次元削減された活動 (PCA)")
for i in 1:Ng
    subplot(3,3,i)
    imshow(Y_pca[i, :, :], cmap="turbo")
    axis("off")
end
tight_layout()
```

自己相関マップ (autocorrelation map) を確認する。ToDo: 相関の計算の説明

次元削減された活動 (PCA)

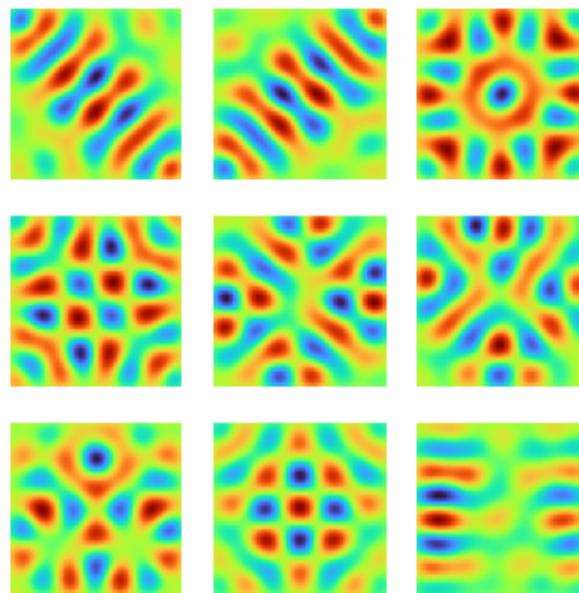


図 5.6 cell034.png

```
function correlate_fft(x, y)
    corr = fftshift(real(iffft(fft(x) .* conj(fft(y)))))
    return corr / maximum(corr)
end;
```

```
corr_pca = [correlate_fft(Y_pca[i, :, :], Y_pca[i, :, :]) for i in 1:Ng];
```

```
figure(figsize=(3,3.5))
suptitle("自己相関マップ (PCA)")
for i in 1:Ng
    subplot(3,3,i)
    imshow(corr_pca[i], cmap="turbo")
    axis("off")
end
tight_layout()
```

非負 PCA の場合

```
relu(x) = max(x, 0)
```

自己相関マップ (PCA)

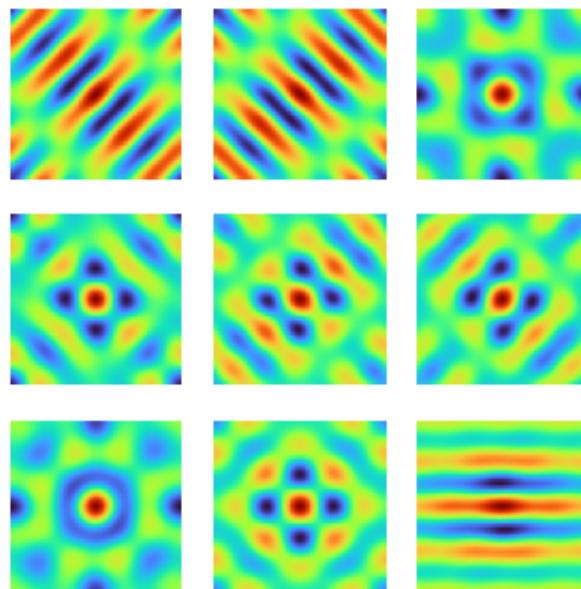


図 5.7 cell038.png

```
@time Y_npca, W_npca = HebbianPCA(X_place; n_components=Ng, η=1e-2, maxiter=5000, ~
    func=relu, orthogonal=true);
Y_npca = reshape(Y_npca, (Ng, step, step));
```

```
figure(figsize=(3,3.5))
suptitle("次元削減された活動 (非負PCA)")
for i in 1:Ng
    subplot(3,3,i)
    imshow(Y_npca[i, :, :], cmap="turbo")
    axis("off")
end
tight_layout()
```

```
corr_npca = [correlate_fft(Y_npca[i, :, :], Y_npca[i, :, :]) for i in 1:Ng];
```

```
figure(figsize=(3,3.5))
suptitle("自己相関マップ (非負PCA)")
```

次元削減された活動 (非負PCA)

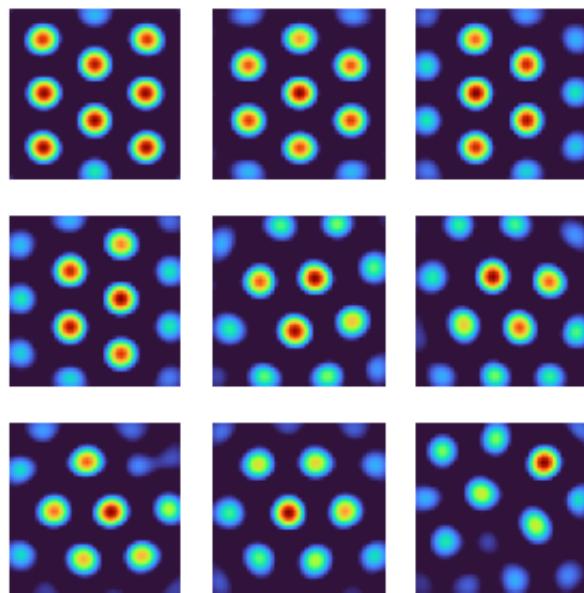


図 5.8 cell042.png

```

for i in 1:Ng
    subplot(3,3,i)
    imshow(corr_npca[i], cmap="turbo")
    axis("off")
end
tight_layout()

```

Place cell の受容野を DoG に設定したが、これが無いと格子状の受容野は出現しない。path integration を RNN で実行する場合も同様。一方で、DoG は場所細胞の受容野としては不適切である。No Free Lunch from Deep Learning in Neuroscience: A Case Study through Models of the Entorhinal-Hippocampal Circuit <https://openreview.net/forum?id=mx1xKzNFr> ToDo: 他の grid cells のモデルについて

5.2 Slow Feature Analysis (SFA)

Slow Feature Analysis (SFA) とは、複数の時系列データの中から低速に変化する成分 (slow feature) を抽出する教師なし学習のアルゴリズムである (Wiskott and Sejnowski, 2002; Wiskott et al.,

自己相関マップ (非負PCA)

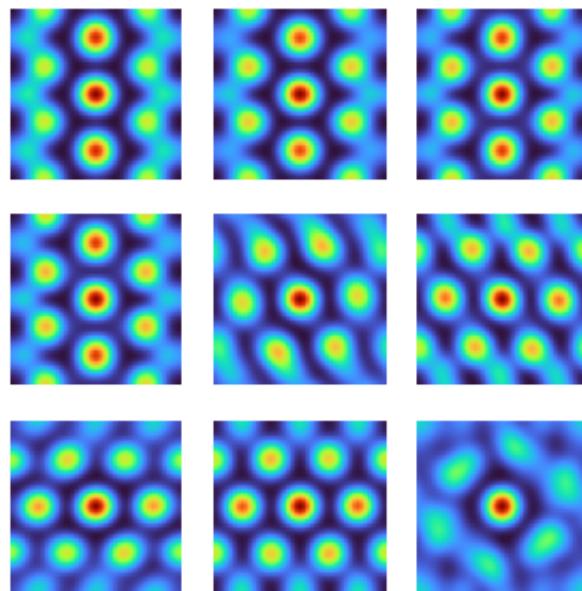


図 5.9 cell044.png

2011). 潜在変数 y の時間変化の 2 乗である $\left(\frac{dy}{dt}\right)^2$ を最小にするように教師なし学習を行う。初期視覚野の受容野 (Berkes2005-i) や格子細胞・場所細胞などのモデルに応用がされている (Franzius et al., 2007)。生理学的妥当性についてはいくつかの検討がされている。(Sprekeler et al., 2007) では STDP 則により SFA が実現できることを報告している。古典的な線形 Recurrent neural network での実装も提案されている (Lipshutz et al., 2020)。まずデータセットの生成を行う。(Wiskott and Sejnowski, 2002) で用いられているトイデータを用いる。

```
using PyPlot, Statistics, LinearAlgebra
rc("axes.spines", top=false, right=false)
```

```
# create the input signal
nt = 5000;
t = range(0, 2π, length=nt)

x1 = sin.(t) + 2*cos.(11*t).^2;
x2 = cos.(11*t);

X = [x1 x2];
```

```

figure(figsize=(5, 2))
subplot2grid((2, 2), (0, 0), rowspan=2)
plot(x2, x1)
xlabel(L" $x_2$ ")
ylabel(L" $x_1$ ")
subplot2grid((2, 2), (0, 1))
plot(t, x1)
ylabel(L" $x_1$ ")
subplot2grid((2, 2), (1, 1))
plot(t, x2)
xlabel("Time")
ylabel(L" $x_2$ ")
xlim(0, 2 $\pi$ )
tight_layout()

```

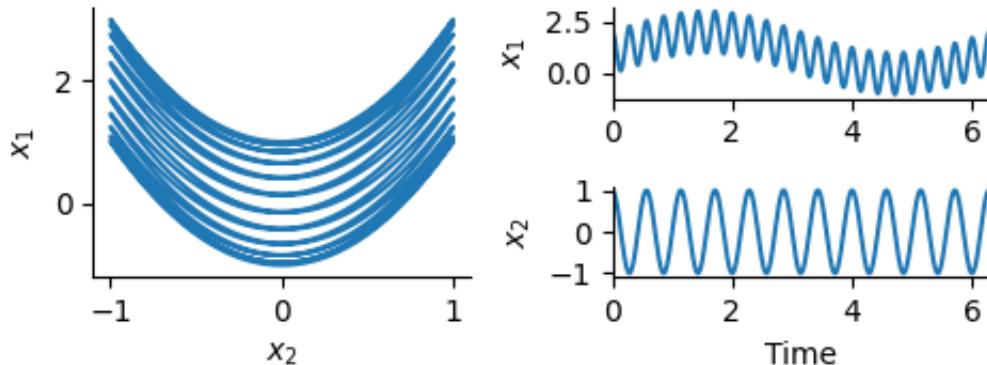


図 5.10 cell004.png

SFA の前処理として多項式展開 (polynomial expansion) が用いられる.

```

monomials(n, d) = [t for t in Base.product(ntuple(i->0:d, Val{n}())...) if sum(t)<=d &
&& sum(t) > 0]
polynomial_expand(X, d) = hcat([[prod(X[i, :] .^ m) for m in monomials(size(X)[2], d)] for i in 1:size(X)[1]]...)
whiten(X) = (X .- mean(X, dims=1)) ./ std(X, dims=1);

```

時間的にずらして時系列データの次元を増やす前処理も行われる.

```
time_frames(X, d) = hcat([X[i:end-d+i] for i in 1:d]...);
```

SFA の実装をする.

```

# Linear slow feature analysis
function linsfa(X)
    # X ∈ R^(dims x timesteps)
    Xw = whiten(X)

```

```

    -, -, V = svd(diff(Xw, dims=1))
    return Xw[1:end-1, :] * V; # V means weight matrix of X to Y
end

```

実行と結果表示を行う。

```
Y = linsfa(polynomial_expand(X, 2));
```

```

figure(figsize=(10, 3))
subplot2grid((3, 3), (0, 0), rowspan=2)
plot(t[1:end-1], whiten(Y[:, end]), label="Estimated")
plot(t[1:end-1], whiten(sin.(t[1:end-1])), "--", label="Actual")
ylabel("SF1"); xlim(0, 2π); legend(loc="upper right");
for i in 1:4
    if i == 1
        subplot2grid((3, 3), (2, 0))
        xlabel("Time");
    else
        subplot2grid((3, 3), (i-2, 1))
    end
    plot(t[1:end-1], whiten(Y[:, end-i]))
    ylabel("SF$(i+1)$"); xlim(0, 2π)
end
xlabel("Time")
tight_layout()

```

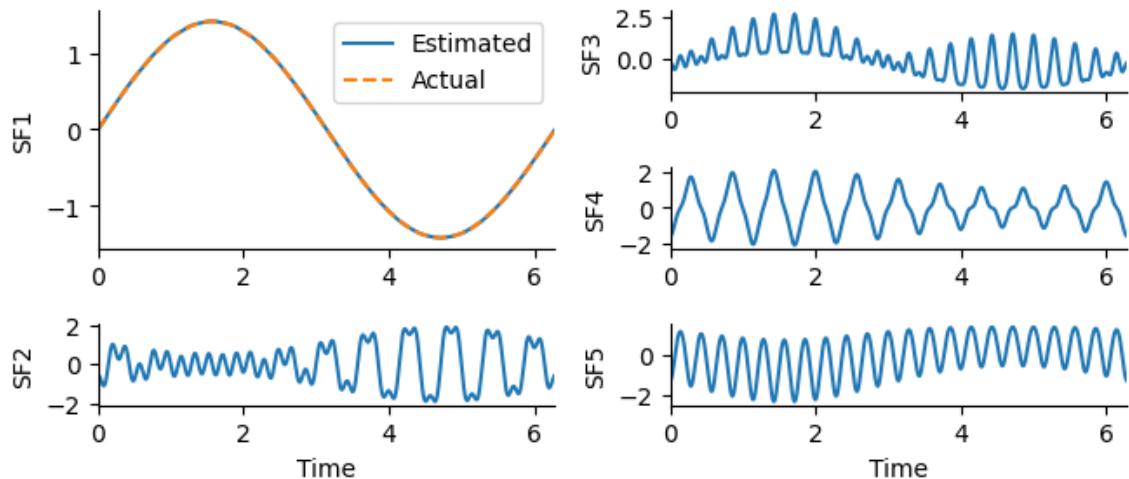


図 5.11 cell013.png

5.3 自己組織化マップと視覚野の構造

視覚野にはコラム構造が存在する。こうした構造は神経活動依存的な発生 (activity dependent development) により獲得される。本節では視覚野のコラム構造を生み出す数理モデルの中で、**自己組織化マップ (self-organizing map)** (Kohonen, 1982), (Kohonen, 2013) を取り上げる。自己組織化マップを視覚野の構造に適応したのは (Obermayer et al., 1990) (N. V. Swindale, 1998) などの研究である。視覚野マップの数理モデルとして自己組織化マップは受容野を考慮しないなどの簡略化がなされているが、単純な手法にして視覚野の構造に関する良い予測を与える。他の数理モデルとしては自己組織化マップと発想が類似している **Elastic net** (Durbin and D. Willshaw, 1987) (Durbin and Mitchison, 1990) (Carreira-Perpiñán et al., 2005) (ここで Elastic net は正則化手法としての Elastic net regularization とは異なる) や受容野を明示的に設定した (Tanaka et al., 2004), (Ringach, 2007) などのモデルがある。総説としては (Das, 2005), (Goodhill, 2007)、数理モデル同士の関係については (田中, 2002) が詳しい。自己組織化マップでは「抹消から中枢への伝達過程で損失される情報量」、および「近い性質を持ったニューロン同士が結合するような配線長」の両者を最小化するような学習が行われる。包括性 (coverage) と連続性 (continuity) のトレードオフとも呼ばれる (Carreira-Perpiñán et al., 2005) (Elastic net は両者を明示的に計算し、線形結合で表されるエネルギー関数を最小化する。Elastic net は本書では取り扱わないが、MATLAB 実装が公開されている <https://faculty.ucmerced.edu/mcarreira-perpinan/research/EN.html>)。連続性と関連する事項として、近い性質を持つ細胞が脳内で近傍に存在するような発生/発達過程を**トポグラフィックマッピング (topographic mapping)** と呼ぶ。トポグラフィックマッピングの数理モデルの初期の研究としては (Malsburg, 1973) (D. J. Willshaw et al., 1976) (Takeuchi and S. Amari, 1979)などがある。発生の数理モデルに関する総説 (Ooyen, 2011), (Goodhill, 2018)

5.3.1 単純なデータセット

SOM における n 番目の入力を $\mathbf{v}(t) = \mathbf{v}_n \in \mathbb{R}^D (n = 1, \dots, N)$, m 番目のニューロン ($m = 1, \dots, M$) の重みベクトル (または活動ベクトル、参照ベクトル) を $\mathbf{w}_m(t) \in \mathbb{R}^D$ とする (Kohonen, 2013)。また、各ニューロンの物理的な位置を \mathbf{x}_m とする。このとき、 $\mathbf{v}(t)$ に対して $\mathbf{w}_m(t)$ を次のように更新する。まず、 $\mathbf{v}(t)$ と $\mathbf{w}_m(t)$ の間の距離が最も小さい (類似度が最も大きい) ニューロンを見つける。距離や類似度としてはユークリッド距離やコサイン類似度などが考えられる。

$$[\text{ユークリッド距離}] : c = \operatorname{argmin}_m [\|\mathbf{v}(t) - \mathbf{w}_m(t)\|^2] \quad (5.18)$$

$$[\text{コサイン類似度}] : c = \operatorname{argmax}_m \left[\frac{\mathbf{w}_m(t)^\top \mathbf{v}(t)}{\|\mathbf{w}_m(t)\| \|\mathbf{v}(t)\|} \right] \quad (5.19)$$

この、 c 番目のニューロンを**勝者ユニット (best matching unit; BMU)** と呼ぶ。コサイン類似度において、 $\mathbf{w}_m(t)^\top \mathbf{v}(t)$ は線形ニューロンモデルの出力となる。このため、コサイン距離を採用する方が生理学的に妥当であり SOM の初期の研究ではコサイン類似度が用いられている (Kohonen, 1982)。しかし、コサイン類似度を用いる場合は \mathbf{w}_m および \mathbf{v} を正規化する必要がある。ユークリッド距離を用いると正規化なしでも学習できるため、SOM を応用する上ではユークリッド距離が採用される事が多い。ユークリッド距離を用いる場合、 \mathbf{w}_m は重みベクトルではなくなるため、活動ベクトルや参照ベクトルと呼ばれる。ここでは結果の安定性を優先してユークリッド距離を用いることとする。こうして得られた c を用いて \mathbf{w}_m を次のように更新する。

$$\mathbf{w}_m(t+1) = \mathbf{w}_m(t) + h_{cm}(t)[\mathbf{v}(t) - \mathbf{w}_m(t)] \quad (5.20)$$

ここで $h_{cm}(t)$ は近傍関数 (neighborhood function) と呼ばれ、 c 番目と m 番目のニューロンの距離が近いほど大きな値を取る。ガウス関数を用いるのが一般的である。

$$h_{cm}(t) = \alpha(t) \exp\left(-\frac{\|\mathbf{x}_c - \mathbf{x}_m\|^2}{2\sigma^2(t)}\right) \quad (5.21)$$

ここで \mathbf{x} はニューロンの位置を表すベクトルである。また、 $\alpha(t), \sigma(t)$ は単調に減少するように設定する。^{*1}

```
using Random, PyPlot, ProgressMeter
using PyPlot: matplotlib
rc("font", family="Arial")
```

```
using PyCall
@pyimport numpy as np
```

ToDo: dims を v, w で修正

```
# inputs
Random.seed!(1234);
σv, σw = 0.1, 0.05
dims = 2 # dims of inputs and neurons
num_v = 300 # num of inputs
num_blobs = 5 # num. cluster of dataset
num_w_sqrt = 15 # must be int
num_w = num_w_sqrt^2
init_w = σw*randn(num_w, dims);
```

^{*1} Generative topographic map (GTM) を用いれば $\alpha(t), \sigma(t)$ の縮小は必要ない。また、SOM と GTM の間を取ったモデルとして S-map がある。

```

# 単位円上に等間隔にならんだクラスターによるtoy datasetを作成する
function make_blobs(num_samples, num_blobs, dims, σ)
    n = Int(num_samples/num_blobs) # number of samples in each
    data = vcat([σ*randn(n, dims) .+ [cos(i/num_blobs*2π), sin(i/num_blobs*2π)]' for i in 0:num_blobs-1]...)
    label = repeat(1:num_blobs, inner=n)
    return data, label
end

v, v_labels = make_blobs(num_v, num_blobs, dims, σv);

function plot_som(v, w, ax; vcolor="tab:blue")
    num_w, dims = size(w)
    num_w_sqrt = Int(sqrt(num_w))
    rw = reshape(w, (num_w_sqrt, num_w_sqrt, dims))
    ax.scatter(v[:, 1], v[:, 2], s=10, color=vcolor)
    ax.plot(rw[:, :, 1], rw[:, :, 2], "k", alpha=0.5);
    ax.plot(rw[:, :, 1]', rw[:, :, 2]', "k", alpha=0.5)
    ax.scatter(w[:, 1], w[:, 2], s=5, fc="white", ec="k", zorder=99) # w[i, j, 1]とw[i, j, 2]の点をプロット
end;

```

近傍関数 (neighborhood function) のための二次元ガウス関数を実装する。Winner ニューロンからの距離に応じて値が減弱する関数である。ここでは一つの入力に対して全てのニューロンの活動ベクトルを更新するということはせず、winner neuron の近傍のニューロンのみ更新を行う。つまり、更新においては global ではなく local な処理のみを行うということである (Winner neuron の決定には WTA による global な評価が必要ではあるが)。自己組織化マップのメインとなる関数を書く。ナイーブに実装する。この方法だと空間が円、球体やトーラスのように周期性を持つ場合にも適応できる。

```

function som(v, init_w; α0=1.0, σ0=6, T=500, dist_mat=nothing, return_history=true)
    # α0: update rate, σ0 : width, T : training steps
    w = copy(init_w)
    num_w = size(init_w)[1]
    num_w_sqrt = Int(sqrt(num_w))
    num_v = size(v)[1]

    if return_history
        w_history = [copy(init_w)] # history of w
    end

    if dist_mat == nothing
        pos = hcat([[i, j] for i in 1:num_w_sqrt for j in 1:num_w_sqrt]...)
        dist_mat = hcat([sum((pos .- pos[:, i]) .^2, dims=1)' for i in 1:num_w]...); #
    end

    @showprogress for t in 1:T

```

```

α = α₀ * (1 - t/T); # update rate
σ = max(α₀ * (1 - t/T), 1); # decay from large to small (linearly decreased, ↴
    avoid zero)
exp_dist_mat = exp.(-dist_mat / (2.0(σ^2)))
exp_dist_mat ./= maximum(sum(exp_dist_mat, dims=1))
# loop for the num_v inputs
for i in 1:num_v
    dist = sum((v[i, :]' .- w).^2, dims=2) # distance between input and ↴
        neurons
    win_idx = argmin(dist)[1] # winner index
    # update the winner & neighbor neuron
    η = α * exp_dist_mat[win_idx, :]
    w[:, :] += η .* (v[i, :]' .- w)
end
if return_history
    append!(w_history, [copy(w)]) # save w
end
if return_history
    return w_history
else
    return w
end
end;

```

今回のように2次元のみを扱う場合はwinner neuronの周辺だけをsliceで抜き出して重み更新する方が高速である。

```

# Gaussian mask for inputs
function gaussian_mask(sizex=9, sizey=9; σ=5)
    x, y = 0:sizex-1, 0:sizey-1
    X, Y = ones(sizey) * x', y * ones(sizex)'
    x0, y0 = (sizex-1) / 2, (sizey-1) / 2
    mask = exp.(-((X .- x0) .^2 + (Y .- y0) .^2) / (2.0(σ^2)))
    return mask ./ sum(mask)
end;

function som_2d(v, init_w; α₀=1.0, σ₀=6, T=500, return_history=true)
    # α₀: update rate, σ₀ : width, T : training steps
    w = copy(init_w)
    num_w, dims = size(init_w)
    num_w_sqrt = Int(sqrt(num_w))
    num_v = size(v)[1]

    w_history = [copy(w)] # history of w

    w_2d = reshape(w, (num_w_sqrt, num_w_sqrt, dims))

    if return_history
        w_history = [copy(init_w)] # history of w
    end

```

```

@showprogress for t in 1:T
    α = α₀ * (1 - t/T); # update rate
    σ = max(α₀ * (1 - t/T), 1); # decay from large to small (linearly decreased, ←
        avoid zero)
    w̄m = ceil(Int, σ)
    h = gaussian_mask(2w̄m+1, 2w̄m+1, σ=σ);
    # loop for the num_v inputs
    for i in 1:num_v
        dist = sum([(v[i, j] .- w_2d[:, :, j]).^2 for j in 1:dims]) # distance ←
            between input and neurons
        win_idx = argmin(dist) # winner index
        idx = [max(1,win_idx[j] - w̄m):min(num_w_sqrt, win_idx[j] + w̄m) for j in ←
            1:2] # neighbor indices
        # update the winner & neighbor neuron
        η = α * h[1:length(idx[1]), 1:length(idx[2])]
        for j in 1:dims
            w_2d[idx..., j] += η .* (v[i, j] .- w_2d[idx..., j])
        end
    end
    if return_history
        w = reshape(w_2d, (num_w, dims))
        append!(w_history, [copy(w)]) # save w
    end
end
if return_history
    return w_history
else
    w = reshape(w_2d, (num_w, dims))
    return w
end
end;

```

w_history = som(v, init_w, α₀=2, σ₀=10, T=100);
#w_history = som_2d(v, init_w, α₀=2, σ₀=10, T=100);

赤色の点がデータ位置 v , 白色の点が重み位置 w である. 黒線はニューロン間の位置関係を表す (これは Weight unfolding diagrams と呼ばれる). 下段のヒートマップは w の一番目の次元を表す. 学習が進むとともに近傍のニューロンが近い活動ベクトルを持つことがわかる.

```

cm = get_cmap(:Reds)
vcolors = cm.(v_labels / num_blobs);

indices = [1, 50, 100]
fig, axes = subplots(2, length(indices), figsize=(6, 4), sharey="row", ←
    subplot_kw=Dict("box_aspect"=>1))
for (i, idx) in enumerate(indices)
    wh = w_history[idx]
    axes[1, i].set_title("Epoch : "*string(idx))
    axes[1, i].spines["right"].set_visible(false)
    axes[1, i].spines["top"].set_visible(false)

```

```

plot_som(v, wh, axes[1, i], vcolor=vcolors);
axes[2, i].imshow(reshape(wh[:, 1], (num_w_sqrt, num_w_sqrt)));
end
axes[1, 1].set_ylabel("Weight unfolding\nin data space")
axes[2, 1].set_ylabel("1st dim. weight")
tight_layout()

```

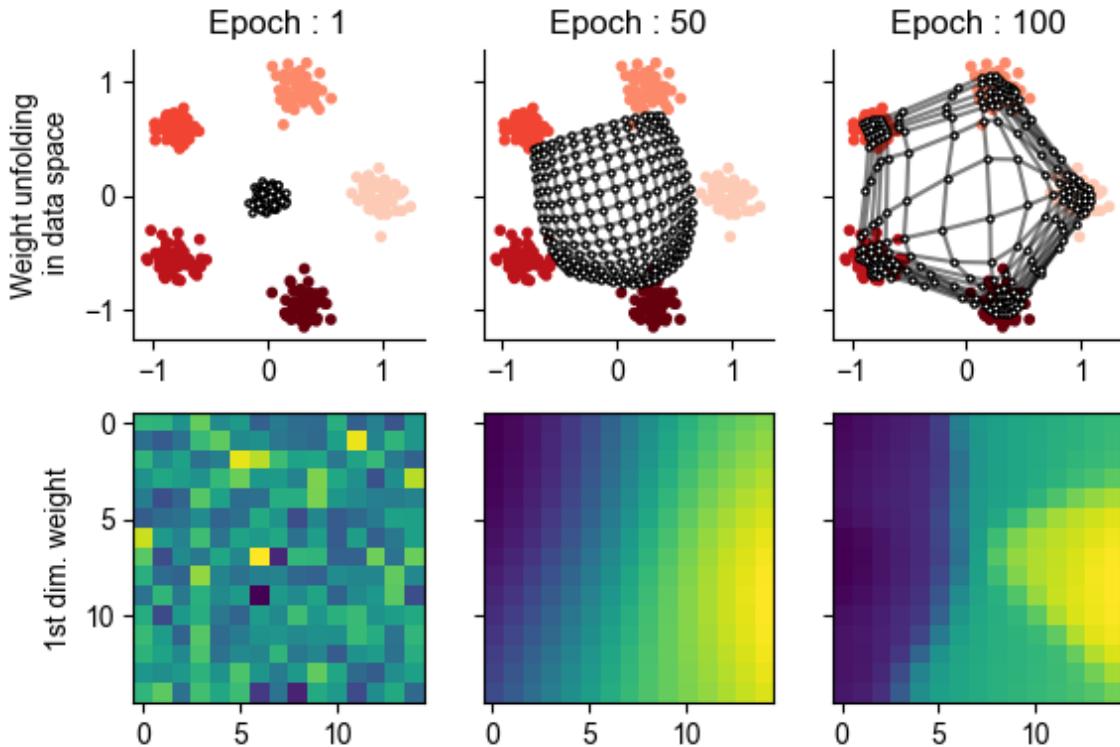


図 5.12 cell018.png

次に U -matrix (unified distance matrix) という隣接ユニットの重み間の距離を定義する。 U -matrix の i 番目の要素 U_i ($i = 1, \dots, M$) は次式で定義される：

$$U_i := \sqrt{\frac{1}{|\mathcal{A}_i|} \sum_{j \in \mathcal{A}_i} \|\mathbf{w}_i - \mathbf{w}_j\|^2} \quad (5.22)$$

ここで $\mathcal{A}_i = \{m \in \{1, \dots, M\} \mid d(\mathbf{x}_i, \mathbf{x}_m) = 1\}$ であり、 $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$ とする。隣接する要素とは位置の差の絶対値が 1 であることを利用する。 U -matrix を計算することでクラスター境界を可視化できる。

```

function u_matrix2d(w)
    num_w = size(w)[1]
    num_w_sqrt = Int(sqrt(num_w))
    pos = hcat([[i, j] for i in 1:num_w_sqrt for j in 1:num_w_sqrt]...)
    abs_dist_mat = hcat([sum(abs.(pos .- pos[:, i])), dims=1]' for i in 1:num_w]...)
    adj_indices = [findall(x -> x == 1, abs_dist_mat[i, :]) for i in 1:num_w] # ←
        adjacent indices
    U = [sqrt(sum((w[adj_indices[i], :] .- w[i, :]) .^2)) / size(adj_indices[i])[1]) -
        for i in 1:num_w]
    U = reshape(U, (num_w_sqrt, num_w_sqrt));
    return U
end

# find best matching unit
function find_bmu(v, w)
    num_v, dims = size(v)
    num_w = size(init_w)[1]
    num_w_sqrt = Int(sqrt(num_w))

    pos = hcat([[i, j] for i in 1:num_w_sqrt for j in 1:num_w_sqrt]...)
    mapped_vpos = zeros(num_v, dims);
    for i in 1:num_v
        dist = sum((v[i, :]'. - w).^2, dims=2) # distance between input and neurons
        win_idx = argmin(dist)[1] # winner index
        mapped_vpos[i, :] = pos[:, win_idx]'. - 1
    end
    return mapped_vpos
end

U = u_matrix2d(w_history[end]);

mapped_vpos = find_bmu(v, w_history[end]);
unique_mapped_vpos, indices, counts = np.unique(mapped_vpos, axis=0, ←
    return_index=true, return_counts=true);

fig, ax = subplots(figsize=(3, 3), subplot_kw=Dict("box_aspect"=>1))
ax.set_title(L"\$U\$-matrix")
ax.pcolormesh(U, cmap="Blues", edgecolors="w", linewidth=0.5)
ax.scatter(unique_mapped_vpos[:, 1].+0.5, unique_mapped_vpos[:, 2].+0.5, ←
    s=counts*10, color=vcolors[indices.+1])
tight_layout()

```

5.3.2 視覚野マップ

集合の直積を配列として返す関数 `product` と極座標を直交座標に変換する関数 `pol2cart` を用意する。

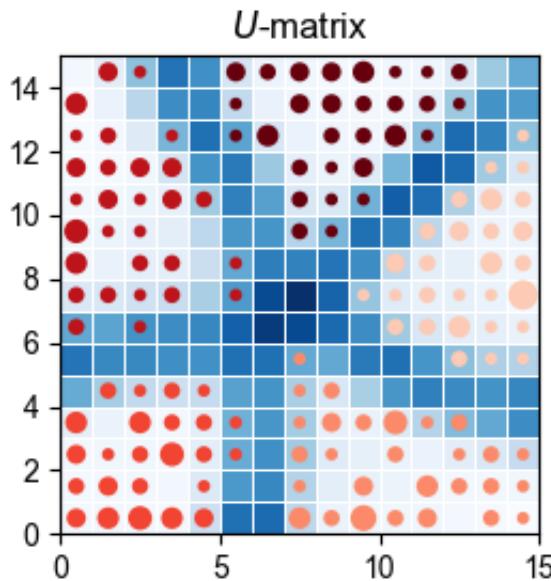


図 5.13 cell024.png

```
product(sets...) = hcat([collect(x) for x in Iterators.product(sets...)])' # ←
    Array of Cartesian product of sets
pol2cart(θ, r) = r*[cos(θ), sin(θ)];
```

刺激と初期の活動ベクトルは (Carreira-Perpiñán et al., 2005) を参考に作成。SOM の学習に用いる刺激は画像ではなく、刺激間の距離が保たれるようなパラメータのみを使用する。刺激空間は 5 つの次元を持つ。視野における受容野の中心座標 ($x, y \in [0, 1]$), 眼優位性 (ocular dominance; OD) $\in [-l_{OD}, l_{OD}]$, 方位 (orientation preference) から成る。方位選択性における方位 θ は $[-\pi/2, \pi/2]$ の範囲の値を取る。ここで, $f(-\pi/2) = f(\pi/2)$ かつ, $\Delta f(\theta) = f(\theta + \Delta\theta) - f(\theta)$ が任意の θ で等しくなるという 2 条件を満たすには, $f(\theta; r) = [r \cos(2\theta), r \sin(2\theta)]$ ($r > 0$) とすればよい。実質的に単位円上から均等に点をサンプリングすることとなる。これらの刺激から直積 `product` で全ての組の入力を作成する。

```
# generate stimulus
Random.seed!(1234);
Nx, Ny, NOD, NOR = 10, 10, 2, 12
dims = 5 # dims of inputs
l, r = 0.14, 0.17

rx, ry = range(0, 1, length=Nx), range(0, 1, length=Ny)
rOD = range(-l, l, length=NOD)
rORθ = range(-π/2, π/2, length=NOR)
```

```
# stimuli
r0Rxy = hcat(pol2cart.(2r0Rθ, r)...)
v = product(rx, ry, r0D, r0Rxy[1, :], r0Rxy[2, :]);
```



```
# initial neurons
num_w_sqrt = 64
num_w = num_w_sqrt^2
init_w = product(range(0, 1, length=num_w_sqrt), range(0, 1, length=num_w_sqrt))
init_w += (rand(size(init_w)...).- 1) * 0.05;
init_w = [init_w l*(2bitrand(num_w).- 1) hcat(pol2cart.(2π*rand(num_w), r)...)'];
```

w_history を用いてアニメーションを作成すると発達の過程が可視化される。

```
w = som_2d(v, init_w, α0=1.5, σ0=5.0, T=50, return_history=false); # faster
#w = som(v, init_w, α0=1.5, σ0=5.0, T=50, return_history=false);
```

```
rw = reshape(w, (num_w_sqrt, num_w_sqrt, dims))
ORmap = atan.(rw[:, :, 5], rw[:, :, 4]) / 2; # get angle of polar
x = 0:num_w_sqrt-1
X = ones(num_w_sqrt) * x';
```

```
fig, axes = subplots(2,2, figsize=(7, 6), subplot_kw=Dict("box_aspect"=>1))#, ...
    adjustable="box", aspect=1)
fig.subplots_adjust(hspace=0)
axes[1,1].set_title("Retinotopic map")
plot_som(v, w, axes[1,1])
axes[1,2].set_title("Ocular dominance (OD) map")
od_map = axes[1,2].imshow(rw[:, :, 3], cmap="gray", origin="lower")
ins1 = axes[1,2].inset_axes([1.05,0,0.05,1])
fig.colorbar(od_map, cax=ins1, aspect=40, pad=0.08, shrink=0.6)
ins1.text(0, -0.16, "Left", ha="left", va="center")
ins1.text(0, 0.16, "Right", ha="left", va="center")
axes[2,1].set_title("Contours of OD and OR")
axes[2,1].contour(X, X', ORmap, cmap="hsv")
axes[2,1].contour(X, X', rw[:, :, 3], colors="k", levels=1)
axes[2,2].set_title("Orientation (OR) angle map")
axes[2,2].imshow(ORmap, cmap="hsv", origin="lower")

cm = get_cmap(:hsv)
lines, colors = [], []
for i in 1:9
    θ = (i-1)/8*π - pi/2
    c, s = cos(θ), sin(θ)
    push!(lines, [(-c/2, 15-1.5i -s/2), (c/2, 15-1.5i + s/2)])
    push!(colors, cm(1/8*(i-1)))
end

ins2 = axes[2,2].inset_axes([1,0,0.2,1])
ins2.add_collection(matplotlib.collections.LineCollection(lines, ...
    linewidths=3,color=colors))
```

```
ins2.set_aspect("equal")
ins2.axis("off")
ins2.set_xlim(-1, 1); ins2.set_ylim(0, 15)
tight_layout()
```

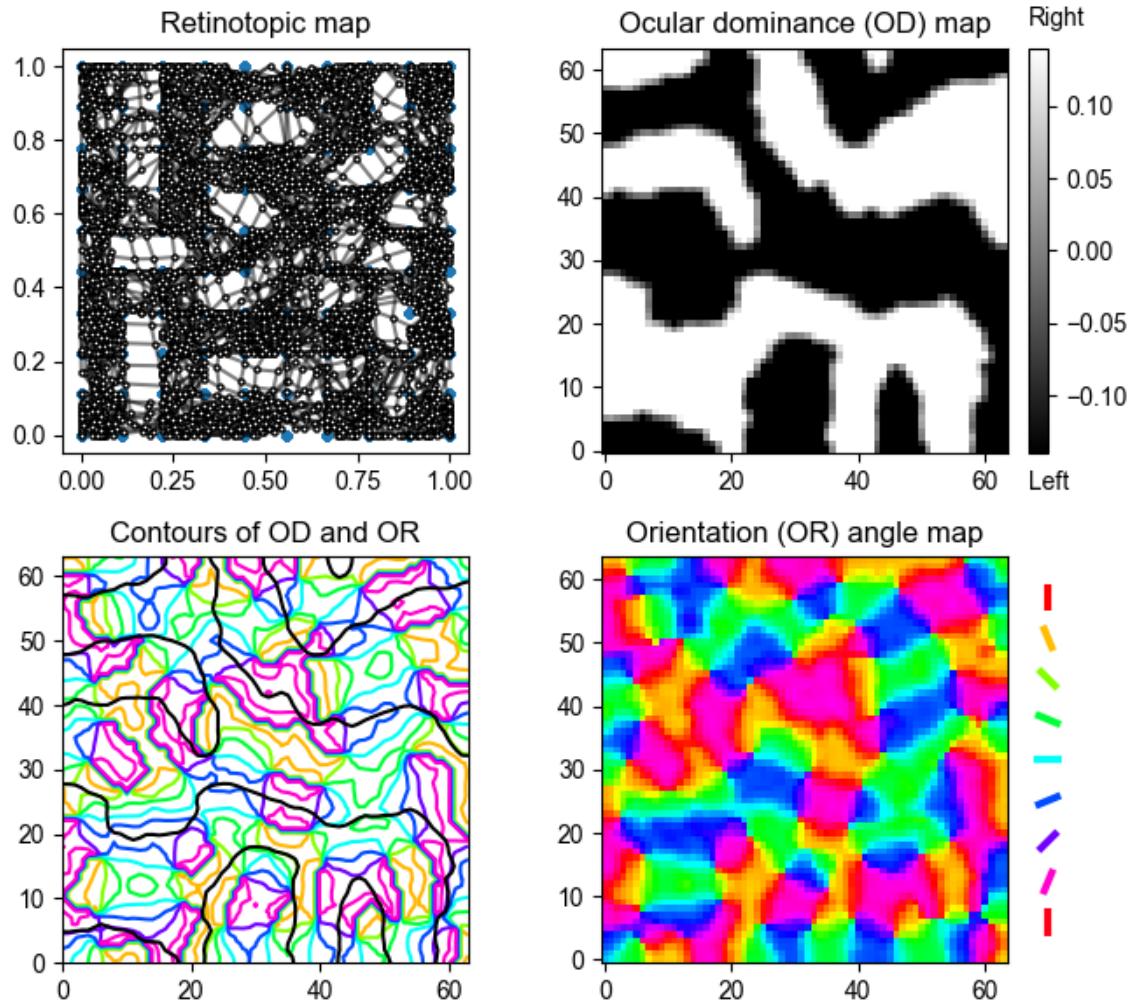


図 5.14 cell033.png

方位選択性マップにはピンホイール (pinwheel) という特異点 (singular point) が存在する。

```
px = range(-1,1,length=50)
pX = ones(length(px)) * px';
fig, axes = subplots(1,2, figsize=(3.5, 2), subplot_kw=Dict("box_aspect"=>1))
```

```

axes[1].imshow(atan.(pX, pX'), cmap="hsv")
axes[2].imshow(atan.(-pX, pX'), cmap="hsv")
axes[1].set_title("CCW (pos.) pinwheel")
axes[2].set_title("CW (neg.) pinwheel")
axes[1].axis("off")
axes[2].axis("off")
fig.tight_layout()

```

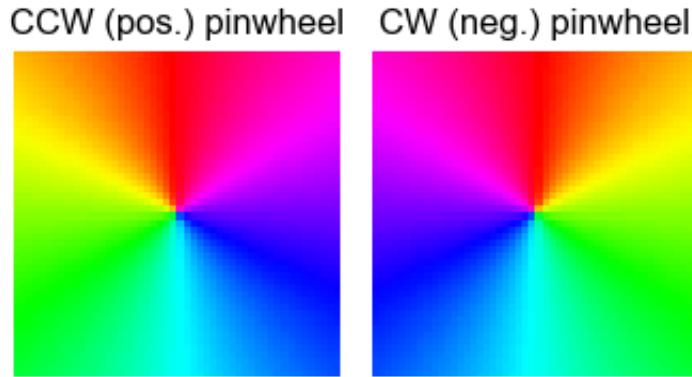


図 5.15 cell035.png

参考文献

- Bienenstock, E. L., Cooper, L. N., and Munro, P. W. (1982). “Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex”. *J. Neurosci.* 2.1, pp. 32–48.
- Bliss, T. V. and Lomo, T. (1973). “Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path”. *J. Physiol.* 232.2, pp. 331–356.
- Brito, C. S. N. and Gerstner, W. (2016). “Nonlinear Hebbian Learning as a Unifying Principle in Receptive Field Formation”. *PLoS Comput. Biol.* 12.9, e1005070.
- Carreira-Perpiñán, M. A., Lister, R. J., and Goodhill, G. J. (2005). “A computational model for the development of multiple maps in primary visual cortex”. *Cereb. Cortex* 15.8, pp. 1222–1233.
- Cooper, L. N. and Bear, M. F. (2012). “The BCM theory of synapse modification at 30: interaction of theory with experiment”. *Nat. Rev. Neurosci.* 13.11, pp. 798–810.
- Cooper, L. N., Liberman, F., and Oja, E. (1979). “A theory for the acquisition and loss of neuron specificity in visual cortex”. *Biol. Cybern.* 33.1, pp. 9–28.

- Das, A. (2005). "Cortical Maps: Where Theory Meets Experiments". *Neuron* 47.2, pp. 168–171.
- Dordek, Y. et al. (2016). "Extracting grid cell characteristics from place cell inputs using non-negative principal component analysis". *Elife* 5, e10094.
- Dudek, S. M. and Bear, M. F. (1992). "Homosynaptic long-term depression in area CA1 of hippocampus and effects of N-methyl-D-aspartate receptor blockade". *Proc. Natl. Acad. Sci. U. S. A.* 89.10, pp. 4363–4367.
- Durbin, R. and Willshaw, D. (1987). "An analogue approach to the travelling salesman problem using an elastic net method". *Nature* 326.6114, pp. 689–691.
- Durbin, R. and Mitchison, G. (1990). "A dimension reduction framework for understanding cortical maps". *Nature* 343.6259, pp. 644–647.
- Franzius, M., Sprekeler, H., and Wiskott, L. (2007). "Slowness and sparseness lead to place, head-direction, and spatial-view cells". *PLoS Comput. Biol.* 3.8, e166.
- Goodhill, G. J. (2007). "Contributions of theoretical modeling to the understanding of neural map development". *Neuron* 56.2, pp. 301–311.
- (2018). "Theoretical Models of Neural Development". *iScience* 8, pp. 183–199.
- Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory*. Wiley & Sons.
- Isomura, T. and Friston, K. (2020). "Reverse-Engineering Neural Networks to Characterize Their Cost Functions". *Neural Comput.* 32.11, pp. 2085–2121.
- Kohonen, T. (2013). "Essentials of the self-organizing map". *Neural Netw.* 37, pp. 52–65.
- (1982). "Self-organized formation of topologically correct feature maps". *Biol. Cybern.* 43.1, pp. 59–69.
- Lipshutz, D. et al. (2020). "A biologically plausible neural network for Slow Feature Analysis". *Adv. Neural Inf. Process. Syst.* abs/2010.12644.
- Malsburg, C. von der (1973). "Self-organization of orientation sensitive cells in the striate cortex". *Kybernetik* 14.2, pp. 85–100.
- N. V. Swindale, H.-U. B. (1998). "Application of Kohonen's self-organizing feature map algorithm to cortical maps of orientation and direction preference". *Proceedings of the Royal Society B: Biological Sciences* 265.1398, p. 827.
- Obermayer, K., Ritter, H., and Schulten, K. (1990). "A principle for the formation of the spatial structure of cortical feature maps". *Proc. Natl. Acad. Sci. U. S. A.* 87.21, pp. 8345–8349.
- Oja, E. (1982). "A simplified neuron model as a principal component analyzer". *J. Math. Biol.* 15.3, pp. 267–273.
- Oja, E. (1997). "The nonlinear PCA learning rule in independent component analysis". *Neurocomputing* 17.1, pp. 25–45.

- Ooyen, A. van (2011). "Using theoretical models to analyse neural development". *Nat. Rev. Neurosci.* 12.6, pp. 311–326.
- Ringach, D. L. (2007). "On the origin of the functional architecture of the cortex". *PLoS One* 2.2, e251.
- Shatz, C. J. (1992). "The developing brain". *Sci. Am.* 267.3, pp. 60–67.
- Sprekeler, H., Michaelis, C., and Wiskott, L. (2007). "Slowness: an objective for spike-timing-dependent plasticity?" *PLoS Comput. Biol.* 3.6, e112.
- Takeuchi, A. and Amari, S. (1979). "Formation of topographic maps and columnar microstructures in nerve fields". *Biol. Cybern.* 35.2, pp. 63–72.
- Tanaka, S., Miyashita, M., and Ribot, J. (2004). "Roles of visual experience and intrinsic mechanism in the activity-dependent self-organization of orientation maps: theory and experiment". *Neural Netw.* 17.8–9, pp. 1363–1375.
- Turrigiano, G. G. (2008). "The self-tuning neuron: synaptic scaling of excitatory synapses". *Cell* 135.3, pp. 422–435.
- Willshaw, D. J., Von Der Malsburg, C., and Longuet-Higgins, H. C. (1976). "How patterned neural connections can be set up by self-organization". *Proceedings of the Royal Society of London. Series B. Biological Sciences* 194.1117, pp. 431–445.
- Wiskott, L. and Sejnowski, T. J. (2002). "Slow feature analysis: unsupervised learning of invariances". *Neural Comput.* 14.4, pp. 715–770.
- Wiskott, L. et al. (2011). "Slow feature analysis". *Scholarpedia J.* 6.4, p. 5282.
- Yee, A. X., Hsu, Y.-T., and Chen, L. (2017). "A metaplasticity view of the interaction between homeostatic and Hebbian plasticity". *Philos. Trans. R. Soc. Lond. B Biol. Sci.* 372.1715.
- Zenke, F., Gerstner, W., and Ganguli, S. (2017). "The temporal paradox of Hebbian learning and homeostatic plasticity". *Curr. Opin. Neurobiol.* 43, pp. 166–176.
- 田中, 繁. (2002). "神経情報科学サマースクール NISS2001 講義録 視覚野神経回路の自己組織化". **日本神経回路学会誌** 9.1, pp. 41–48.

第 6 章

生成モデルとエネルギーベースモデル

6.1 エネルギーベースモデル

本章ではエネルギーベースモデル (energy-based models; EBMs) という枠組みに含まれるモデルを紹介する。エネルギーベースモデルではネットワークの状態をスカラー値に変換するエネルギー関数 (あるいはコスト関数) を定義し、推論時と学習時の双方においてエネルギーを最小化するようにネットワークの状態を更新する。(LeCun et al., 2006) 入力 $\mathbf{x} \in \mathbb{R}^d$, エネルギー関数 $E_\theta : \mathbb{R}^d \rightarrow \mathbb{R}$ を考える。

$$p_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta} \quad (6.1)$$

$$Z_\theta = \int \exp(-E_\theta(\mathbf{x})) d\mathbf{x} \quad (6.2)$$

Z_θ は分配関数。

6.2 Hopfield モデル

(Hopfield, 1982) で提案。始めは 1 と 0 の状態を取った。Hopfield モデルと呼ばれることが多いが、Amari の先駆的研究 (S.-I. Amari, 1972) を踏まえ Amari-Hopfield モデルと呼ばれることもある。次のような連続時間線形モデルを考える。シナプス前活動を $\mathbf{x} \in \mathbb{R}^n$, 後活動を $\mathbf{y} \in \mathbb{R}^m$, 重み行列を $\mathbf{W} \in \mathbb{R}^{m \times n}$ とする。

$$\frac{d\mathbf{y}}{dt} = -\mathbf{y} + \mathbf{W}\mathbf{x} + \mathbf{b} \quad (6.3)$$

ここで $\frac{\partial \mathcal{L}}{\partial \mathbf{y}} := -\frac{d\mathbf{y}}{dt}$ となるような $\mathcal{L} \in \mathbb{R}$ を仮定すると,

$$\mathcal{L} = \int (\mathbf{y} - \mathbf{W}\mathbf{x} - \mathbf{b}) d\mathbf{y} = \frac{1}{2} \|\mathbf{y}\|^2 - \mathbf{y}^\top \mathbf{W}\mathbf{x} - \mathbf{y}^\top \mathbf{b} \quad (6.4)$$

となる。これをさらに \mathbf{W} で微分すると,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = -\mathbf{y}\mathbf{x}^\top \Rightarrow \frac{d\mathbf{W}}{dt} = -\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{y}\mathbf{x}^\top = (\text{post}) \cdot (\text{pre})^\top \quad (6.5)$$

となり、Hebb 則が導出できる。

6.2.1 モデルの定義

モデルを定義する。

```
using Parameters: @unpack
using LinearAlgebra, PyPlot, Random, Distributions, Statistics, ImageTransformations, TestImages, ColorTypes

@kwdef mutable struct AmariHopfieldModel
    W::Array # weights
    θ::Vector # thresholds
end

# Training weights & definition of model
function AmariHopfieldModel(inputs; σθ=1e-2)
    num_data, num_units = size(inputs) # inputs : num_data x num_unit
    inputs = mapslices(x -> x .- mean(x), inputs, dims=2)
    W = (inputs' * inputs) / num_data # hebbian rule
    W -= diagm(diag(W)) # Set the diagonal of weights to zero
    return AmariHopfieldModel(W=W, θ=σθ*randn(num_units))
end;
```

```
binarize(img) = 2.0((img .- mean(img)) .> 0) .- 1; # img to {-1, 1}

function corrupted(img, p=0.3)
    mask = rand(Binomial(1, p), size(img));
    return img .* (1 .- mask) - img .* mask;
end
```

データセットの作成

```
testimagelist = ["cameraman", "jetplane", "house", "mandril_gray", "lake_gray"]; # ←
gray & size(512 x 512)
num_data = length(testimagelist)

imgs = [convert(Array{Float64}, imresize(Gray.(testimage(imagename)), ratio=1/8)) ~
        for imagename in testimagelist];
imgs_binarized = map(binarize, imgs);
imgs_corrupted = map(corrupted, imgs_binarized);

input_train = reshape(stack(imgs_binarized), (:, num_data))';
input_test = reshape(stack(imgs_corrupted), (:, num_data));
```

```

figure(figsize=(8, 3))
for i in 1:num_data
    subplot(2, num_data, i); imshow(imgs[i], cmap="gray");
    xticks([]); yticks([]); if i==1 ylabel("Original", fontsize=14) end;
    subplot(2, num_data, i+num_data); imshow(imgs_binarized[i], cmap="gray");
    xticks([]); yticks([]); if i==1 ylabel("Binarized", fontsize=14) end;
end
tight_layout()

```

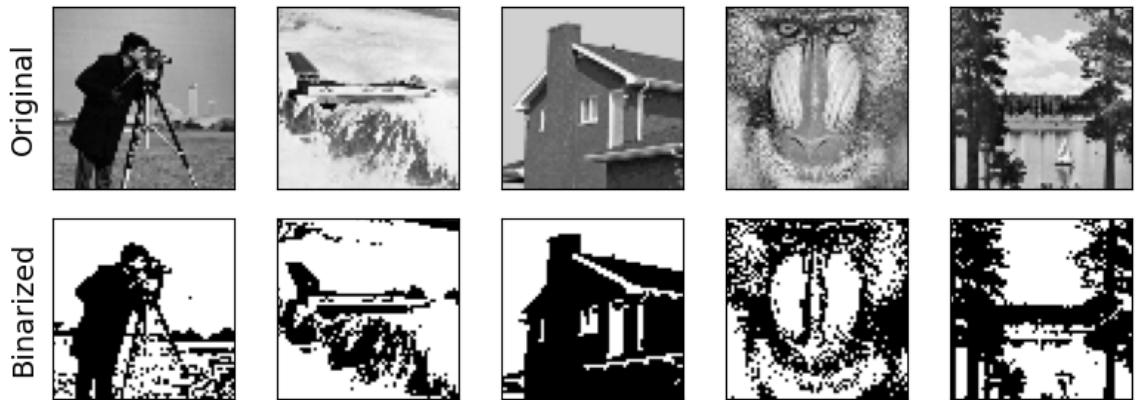


図 6.1 cell007.png

モデルの定義と訓練

```
model = AmariHopfieldModel(input_train);
```

```

figure(figsize=(3, 3))
title("Weight matrix"); imshow(model.W);
tight_layout()

```

画像の復元を行う。エネルギー関数

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i = -\frac{1}{2} \mathbf{s}^\top \mathbf{W} \mathbf{s} + \boldsymbol{\theta}^\top \mathbf{s} \quad (6.6)$$

を最小化するように内部状態 \mathbf{s} を更新。

$$\mathbf{s} \leftarrow \text{sign}(\mathbf{W}\mathbf{s} - \boldsymbol{\theta}) \quad (6.7)$$

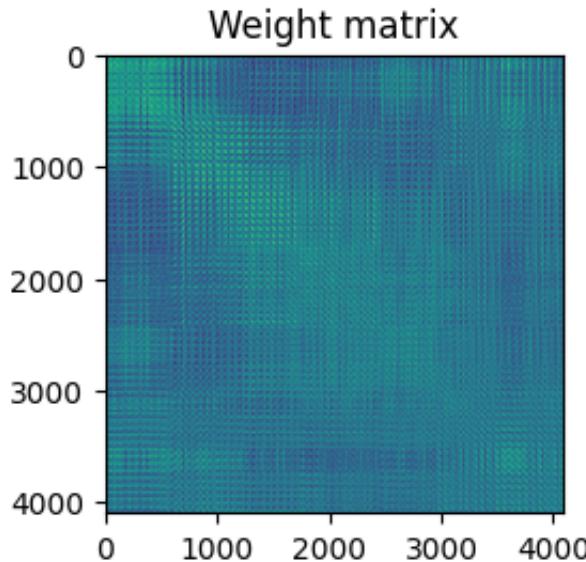


図 6.2 cell010.png

```

energy(W, s, θ) = -0.5s' * W * s + θ' * s

# Synchronous update
function prediction(model::AmariHopfieldModel, init_s, max_iter=100)
    @unpack W, θ = model
    s, e = init_s, energy(W, init_s, θ)
    for t in 1:max_iter
        s = sign.(W * s - θ)      # update s
        e_tp1 = energy(W, s, θ) # compute state (t+1) energy
        if abs(e_tp1-e) < 1e-3  # convergence
            return s
        end
    end
    return s
end;

imgs_predicted = [reshape(prediction(model, input_test[i, :]), (64, 64)) for i in 1:num_data];

figure(figsize=(8, 3))
for i in 1:num_data
    subplot(2, num_data, i); imshow(imgs_corrupted[i], cmap="gray");
    xticks([]); yticks([]); if i==1 ylabel("Inputs", fontsize=14) end;
    subplot(2, num_data, i+num_data); imshow(imgs_predicted[i], cmap="gray");
end;

```

```
xticks([]); yticks([]); if i==1 ylabel("Outputs", fontsize=14) end;  
end  
tight_layout()
```

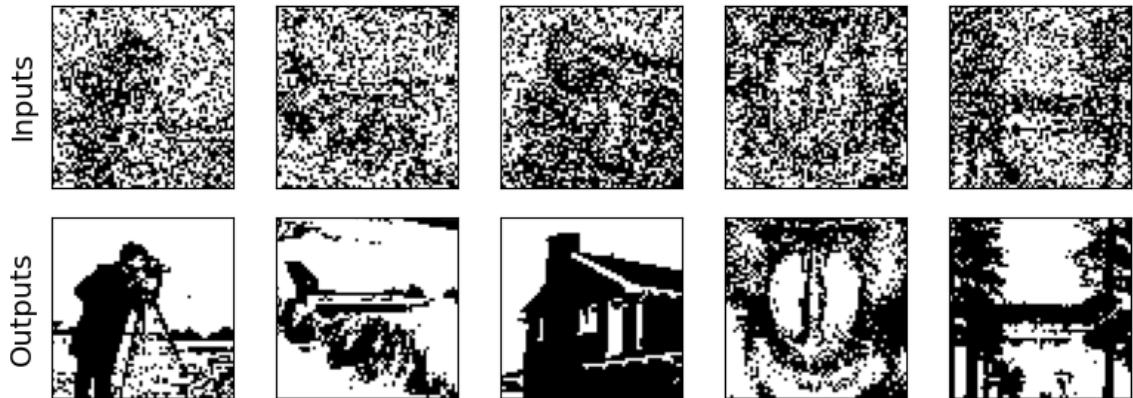


図 6.3 cell014.png

6.2.2 稠密連想記憶 (dense associative memory) モデル

Dense Associative Memory (DAM) モデル (Modern Hopfield networks とも呼ばれる) .

- Krotov, Dmitry, and John J. Hopfield. 2016. “Dense Associative Memory for Pattern Recognition.” arXiv. arXiv. <http://arxiv.org/abs/1606.01164>.
- Krotov, Dmitry, and John Hopfield. 2018. “Dense Associative Memory Is Robust to Adversarial Inputs.” Neural Computation 30 (12): 3151 – 67.
- Krotov, Dmitry, and John J. Hopfield. 2019. “Unsupervised Learning by Competing Hidden Units.” Proceedings of the National Academy of Sciences of the United States of America 116 (16): 7723 – 31.
- Ramsauer, Hubert, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, et al. 2020. “Hopfield Networks Is All You Need.” arXiv. arXiv. <http://arxiv.org/abs/2008.02217>.

深層ニューラルネットワークへの応用.

- Krotov, Dmitry, and John J. Hopfield. 2020. “Large Associative Memory Problem in Neurobiology and Machine Learning.” https://openreview.net/pdf?id=X4y_100X-hX

“Hopfield Networks Is All You Need.” の論文における非生理学的 3 ニューロン相互作用の緩和.

6.3 Boltzmann マシン

6.3.1 Boltzmann マシン

(Boltzmann machine)

6.3.2 制限 Boltzmann マシン

(Restricted Boltzmann machine) (cf.) <http://deeplearning.net/tutorial/rbm.html>
データの読み込み

```
using MLDatasets
using PyPlot
using Random
using ProgressMeter

train_x, _ = MNIST.traindata()
size(train_x)

figure(figsize=(4, 1.5))
for i in 1:4
    subplot(1,4,i)
    imshow(train_x[:, :, i]', cmap="gray")
    axis("off")
end
tight_layout()
```



図 6.4 cell005.png

```
num_data = 100
input_size = 28*28
data = train_x[:, :, 1:num_data]
```

```

data = reshape(data, (input_size, num_data))'
println(size(data))

width = 28 # MNIST dataの幅
num_v = input_size # visible variables
num_h = 100 # hidden variables
num_units = num_v + num_h # all units
η = 0.01 # learning rate
num_epoch = 50 # epoch of learning
num_draws = 20 # The number of samples to draw

```

離散の観測変数 (visible variable) \mathbf{v} , 潜在変数 (hidden variable) \mathbf{h} とする. 各ユニットの値は $\{0, 1\}$ の 2 値 (binary) である. エネルギー関数を

$$E_\theta(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^T \mathbf{v} - \mathbf{c}^T \mathbf{h} + \mathbf{v}^T \mathbf{W} \mathbf{h} \quad (6.8)$$

とする. ただし, $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$

```

# sigmoid function
sigmoid(x) = 1 / (1+exp(-x))

# Initial parameters
W = 0.2 * randn(num_h, num_v)
hbias = 0.2 * randn(num_h, 1)
vbias = 0.2 * randn(num_v, 1)

println(size(W), size(hbias), size(vbias))

```

シグモイド関数を

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \quad (6.9)$$

とする.

訓練データで学習

$$p_\theta(\mathbf{h}|\mathbf{v}) = \prod_i p_\theta(h_i = 1|\mathbf{v}) = \prod_i \sigma(c_i + W_i \mathbf{v}) \quad (6.10)$$

$$p_\theta(\mathbf{v}|\mathbf{h}) = \prod_j p_\theta(v_j = 1|\mathbf{h}) = \prod_j \sigma(b_j + W_j^T \mathbf{h}) \quad (6.11)$$

```

@showprogress "Computing..." for epoch in 1:num_epoch
    for i in 1:num_data
        input = data[i, :]
    end
end

```

```

h_given_v = sigmoid.(W * input + hbias)
v = 0.5 * ones(num_v, 1) # init state
h = 0.5 * ones(num_h, 1) # init state
sum_v = zeros(num_v, 1)
sum_h = zeros(num_h, 1)
outerprod = zeros(num_h, num_v)

for _ in 1:num_draws
    h = 1.0f0 * (sigmoid.(W * v + hbias) .≥ rand(num_h, 1)) # hidden
    v = 1.0f0 * (sigmoid.(W' * h + vbias) .≥ rand(num_v, 1)) # visible
    #h = floor.(sigmoid.(W * v + hbias) + rand(num_h, 1)) # hidden
    #v = floor.(sigmoid.(W' * h + vbias) + rand(num_v, 1)) # visible
    sum_h += h
    sum_v += v
    outerprod += h * v'
end

sum_h /= num_draws
sum_v /= num_draws
outerprod /= num_draws

# update parameters
W += η * (h_given_v * input' - outerprod)
hbias += η * (h_given_v - sum_h)
vbias += η * (input - sum_v)
end
end

```

二項分布 (bernoulli distribution) のサンプリングには 2 通りある。`1.0f0` を乗じているのは Bool 変数から Float への変換のため。詳細は tips. テストデータで確認

```

num_draws_test = 50 # draws for in test
num_see = 392 # Visible units in test
noise_scale = 0.1 # テスト時のノイズレベル
numtestdata = 4

testdata = data[1:numtestdata, :] + noise_scale * randn(numtestdata, input_size)
testdata[:, num_see+1:num_v] .= 0.5

figure(figsize=(4, 1.5))
for i in 1:4
    subplot(1,4,i)
    imshow(reshape(testdata[i, :], (width, width))', cmap="gray")
    axis("off")
end
tight_layout()

energy(v, h) = -v' * vbias - h' * hbias - h' * W * v
# free_energy(v) = -v' * vbias .. sum(log.(1 .+ exp.(W * v + hbias)))

```



図 6.5 cell016.png

```
# Results of Test data
energy_arr = zeros(numtestdata, num_draws_test)
figure(figsize=(4, 1.5))

for i in 1:numtestdata
    v = 0.5 * ones(num_v, 1) # init state
    h = 0.5 * ones(num_h, 1) # init state
    sum_v = zeros(num_v, 1)
    for j in 1:num_draws_test
        v[1:num_see, 1] = testdata[i, 1:num_see]'
        h = 1.0f0 * (sigmoid.(W * v + hbias) .≥ rand(num_h, 1))
        v = 1.0f0 * (sigmoid.(W' * h + vbias) .≥ rand(num_v, 1))
        sum_v += v
        energy_arr[i, j] = energy(v, h)[1]
    end
    sum_v /= num_draws_test

    # show
    subplot(1,4,i)
    imshow(reshape(sum_v, (width, width))', cmap="gray")
    axis("off")
end

tight_layout()
```



図 6.6 cell018.png

エネルギーの変化を見る

```

figure(figsize=(4,3))
ylabel("energy")
xlabel("num. of sampling")
for i in 1:4
    plot(energy_arr[i, :])
end

```

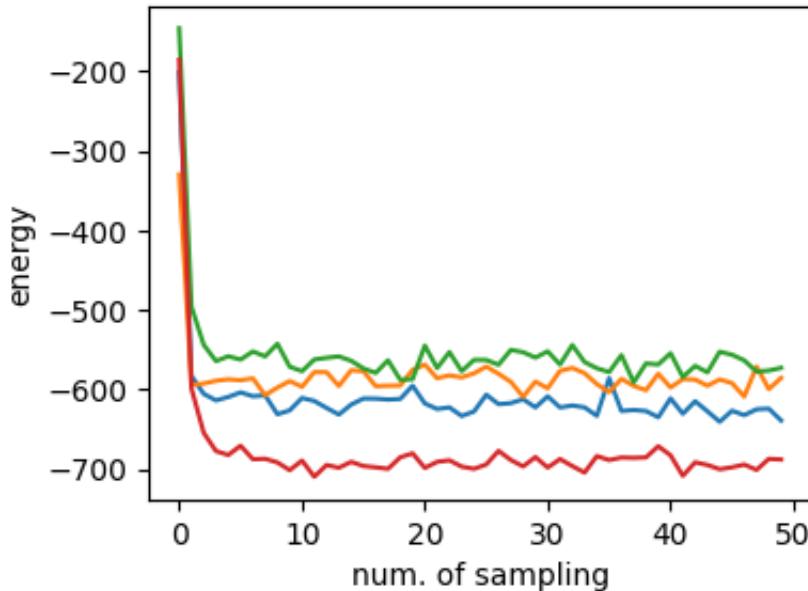


図 6.7 cell020.png

受容野の可視化

```

# Plot Receptive fields
figure(figsize=(5, 5))
subplots_adjust(hspace=0.1, wspace=0.1)
for i in 1:num_h
    subplot(10, 10, i)
    imshow(reshape(W[i, :], (width, width))', cmap="gray")
    axis("off")
end
subtitle("Receptive fields", fontsize=14)
subplots_adjust(top=0.9)

```

Receptive fields



図 6.8 cell022.png

6.4 スパース符号化

6.4.1 Sparse coding と生成モデル

Sparse coding モデル (Olshausen and Field, 1996) (Olshausen and Field, 1997) は V1 のニューロンの応答特性を説明する線形生成モデル (linear generative model) である。まず、画像パッチ \mathbf{x} が基底関数 (basis function) $= [\phi_j]$ のノイズを含む線形和で表されるとする (係数は $\mathbf{r} = [r_j]$ とする)。

$$\mathbf{x} = \sum_j r_j \phi_j + \epsilon = \mathbf{r} + \epsilon \quad (6.12)$$

ただし, $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$ である. このモデルを神経ネットワークのモデルと考えると, ϵ は重み行列, 係数 \mathbf{r} は入力よりも高次の神経細胞の活動度を表していると解釈できる. ただし, r_j は負の値も取るので単純に発火率と捉えられないのはこのモデルの欠点である. Sparse coding では神経活動 \mathbf{r} が潜在変数の推定量を表現しているという仮定の下, 少数の基底で画像 (や目的変数) を表すことを目的とする. 要は上式において, ほとんどが 0 で, 一部だけ 0 以外の値を取るという疎 (=sparse) な係数 \mathbf{r} を求めたい.

確率的モデルの記述

入力される画像パッチ \mathbf{x}_i ($i = 1, \dots, N$) の真の分布を $p_{data}(\mathbf{x})$ とする. また, \mathbf{x} の生成モデルを $p(\mathbf{x}|)$ とする. さらに潜在変数 \mathbf{r} の事前分布 (prior) を $p(\mathbf{r})$, 画像パッチ \mathbf{x} の尤度 (likelihood) を $p(\mathbf{x}|\mathbf{r},)$ とする. このとき,

$$p(\mathbf{x}|) = \int p(\mathbf{x}|\mathbf{r},)p(\mathbf{r})d\mathbf{r} \quad (6.13)$$

が成り立つ. $p(\mathbf{x}|\mathbf{r},)$ は, (1) 式においてノイズ項を $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$ としたことから,

$$p(\mathbf{x}|\mathbf{r},) = \mathcal{N}(\mathbf{x}|\mathbf{r}, \sigma^2 \mathbf{I}) = \frac{1}{Z_\sigma} \exp\left(-\frac{\|\mathbf{x} - \mathbf{r}\|^2}{2\sigma^2}\right) \quad (6.14)$$

と表せる. ただし, Z_σ は規格化定数である.

事前分布の設定

事前分布 $p(\mathbf{r})$ としては, 0 においてピークがあり, 補の重い (heavy tail) を持つ sparse distribution あるいは **super-Gaussian distribution** (Laplace 分布や Cauchy 分布など Gaussian 分布よりも kurtotic な分布) を用いるのが良い. このような分布では, \mathbf{r} の各要素 r_i はほとんど 0 に等しく, ある入力に対しては大きな値を取る. $p(\mathbf{r})$ は一般化して式 (4), (5) のように表記する.

$$p(\mathbf{r}) = \prod_j p(r_j) \quad (6.15)$$

$$p(r_j) = \frac{1}{Z_\beta} \exp[-\beta S(r_j)] \quad (6.16)$$

ただし, β は逆温度 (inverse temperature), Z_β は規格化定数 (分配関数) である. これらの用語は統計力学における正準分布 (Boltzmann 分布) から来ている. $S(x)$ と分布の関係をまとめた表が以下となる.

$S(r)$	$\frac{dS(r)}{dr}$	$p(r)$	分布名	尖度 (kurtosis)
r^2	$2r$	$\frac{1}{\alpha\sqrt{2\pi}} \exp\left(-\frac{r^2}{2\alpha^2}\right)$	Gaussian 分布	0
$ r $	$\text{sign}(r)$	$\frac{1}{2\alpha} \exp\left(-\frac{ r }{\alpha}\right)$	Laplace 分布	3.0
$\ln(\alpha^2 + r^2)$	$\frac{2r}{\alpha^2 + r^2}$	$\frac{\alpha}{\pi} \frac{1}{\alpha^2 + r^2} = \frac{\alpha}{\pi} \exp[-\ln(\alpha^2 + r^2)]$	Cauchy 分布	-

分布 $p(r)$ や $S(r)$ を描画すると次のようになる。

```
using PyPlot

x = range(-5, 5, length=300)
figure(figsize=(7,3))
subplot(1,2,1)
title(L" $p(x)$ ")
plot(x, 1/sqrt(2pi)*exp.(-(x.^2)/2), color="black", linestyle="--", label="Gaussian")
plot(x, 1/2*exp.(-abs.(x)), label="Laplace")
plot(x, 1 ./ (pi*(1 .+ x.^2)), label="Cauchy")
xlim(-5, 5);
xlabel(L" $x$ ")
legend()

subplot(1,2,2)
title(L" $S(x)$ ")
plot(x, x.^2, color="black", linestyle="--", label="Gaussian")
plot(x, abs.(x), label="Laplace")
plot(x, log.(1 .+ x.^2), label="Cauchy")
xlim(-5, 5); ylim(0, 5)
xlabel(L" $x$ ")
tight_layout()
```

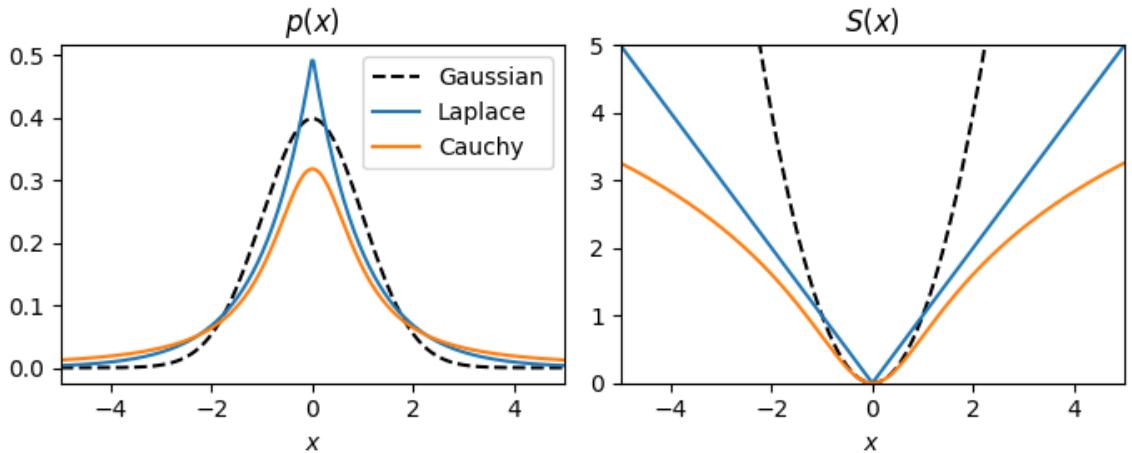


図 6.9 cell003.png

6.4.2 目的関数の設定と最適化

最適な生成モデルを得るために、入力される画像パッチの真の分布 $p_{data}(\mathbf{x})$ と \mathbf{x} の生成モデル $p(\mathbf{x}|)$ を近づける。このために、2つの分布の Kullback-Leibler ダイバージェンス $D_{KL}(p_{data}(\mathbf{x})\| p(\mathbf{x}|))$ を最

小化したい。しかし、真の分布は得られないので、経験分布

$$\hat{p}_{data}(\mathbf{x}) := \frac{1}{N} \sum_{i=1}^N \delta(\mathbf{x} - \mathbf{x}_i) \quad (6.17)$$

を近似として用いる ($\delta(\cdot)$ は Dirac のデルタ関数である)。ゆえに $D_{KL}(\hat{p}_{data}(\mathbf{x}) \| p(\mathbf{x}))$ を最小化する。

$$D_{KL}(\hat{p}_{data}(\mathbf{x}) \| p(\mathbf{x})) = \int \hat{p}_{data}(\mathbf{x}) \log \frac{\hat{p}_{data}(\mathbf{x})}{p(\mathbf{x})} d\mathbf{x} \quad (6.18)$$

$$= \mathbb{E}_{\hat{p}_{data}} \left[\ln \frac{\hat{p}_{data}(\mathbf{x})}{p(\mathbf{x})} \right] \quad (6.19)$$

$$= \mathbb{E}_{\hat{p}_{data}} [\ln \hat{p}_{data}(\mathbf{x})] - \mathbb{E}_{\hat{p}_{data}} [\ln p(\mathbf{x})] \quad (6.20)$$

が成り立つ。 (7) 式の 1 番目の項は一定なので、 $D_{KL}(\hat{p}_{data}(\mathbf{x}) \| p(\mathbf{x}))$ を最小化するには $\mathbb{E}_{\hat{p}_{data}} [\ln p(\mathbf{x})]$ を最大化すればよい。ここで、

$$\mathbb{E}_{\hat{p}_{data}} [\ln p(\mathbf{x})] = \sum_{i=1}^N \hat{p}_{data}(\mathbf{x}_i) \ln p(\mathbf{x}_i) = \frac{1}{N} \sum_{i=1}^N \ln p(\mathbf{x}_i) \quad (6.21)$$

が成り立つ。また、(2) 式より

$$\ln p(\mathbf{x}) = \ln \int p(\mathbf{x}|\mathbf{r},)p(\mathbf{r})d\mathbf{r} \quad (6.22)$$

が成り立つのので、近似として $\int p(\mathbf{x}|\mathbf{r},)p(\mathbf{r})d\mathbf{r}$ を $p(\mathbf{x}|\mathbf{r},)p(\mathbf{r}) (= p(\mathbf{x}, \mathbf{r}))$ で評価する。これらの近似の下、最適な $=^*$ は次のようにして求められる。

$$* = \arg \min_{\mathbf{r}} \min_{\mathbf{x}} D_{KL}(\hat{p}_{data}(\mathbf{x}) \| p(\mathbf{x})) \quad (6.23)$$

$$= \arg \max_{\mathbf{r}} \max_{\mathbf{x}} \mathbb{E}_{\hat{p}_{data}} [\ln p(\mathbf{x})] \quad (6.24)$$

$$= \arg \max_{\mathbf{r}} \sum_{i=1}^N \max_{\mathbf{r}_i} \ln p(\mathbf{x}_i) \quad (6.25)$$

$$\approx \arg \max_{\mathbf{r}} \sum_{i=1}^N \max_{\mathbf{r}_i} \ln p(\mathbf{x}_i | \mathbf{r}_i) p(\mathbf{r}_i) \quad (6.26)$$

$$= \arg \min_{\mathbf{r}} \sum_{i=1}^N \min_{\mathbf{r}_i} E(\mathbf{x}_i, \mathbf{r}_i) \quad (6.27)$$

ただし、 \mathbf{x}_i に対する神経活動を \mathbf{r}_i とした。また、 $E(\mathbf{x}, \mathbf{r})$ はコスト関数であり、次式のように表される。

$$E(\mathbf{x}, \mathbf{r}) := -\ln p(\mathbf{x}|\mathbf{r},)p(\mathbf{r}) \quad (6.28)$$

$$= \underbrace{\|\mathbf{x} - \mathbf{r}\|^2}_{\text{preserve information}} + \lambda \underbrace{\sum_j S(r_j)}_{\text{sparserness of } r_j} \quad (6.29)$$

ただし, $\lambda = 2\sigma^2\beta$ は正則化係数 (この式から逆温度 β が正則化の度合いを調整するパラメータであることがわかる。) であり, 1行目から2行目へは式(3), (4), (5)を用いた。ここで, 第1項が復元損失, 第2項が罰則項(正則化項)となっている。式(9)で表される最適化手順を最適な \mathbf{r} を求める過程に分割しよう。まず, を固定した下で $E(\mathbf{x}_n, \mathbf{r}_i)$ を最小化する $\mathbf{r}_i = \hat{\mathbf{r}}_i$ を求める。

$$\hat{\mathbf{r}}_i = \arg \min_{\mathbf{r}_i} E(\mathbf{x}_i, \mathbf{r}_i) \quad \left(= \arg \max_{\mathbf{r}_i} p(\mathbf{r}_i | \mathbf{x}_i) \right) \quad (6.30)$$

これは \mathbf{r} について **MAP推定** (maximum a posteriori estimation) を行うことに等しい。次に $\hat{\mathbf{r}}$ を用いて

$${}^* = \arg \min \sum_{i=1}^N E(\mathbf{x}_i, \hat{\mathbf{r}}_i) \quad \left(= \arg \max \prod_{i=1}^N p(\mathbf{x}_i | \hat{\mathbf{r}}_i) \right) \quad (6.31)$$

とすることにより, を最適化する。こちらは について **最尤推定** (maximum likelihood estimation) を行うことに等しい。

6.4.3 Locally competitive algorithm (LCA)

\mathbf{r} の勾配法による更新則は, E の微分により次のように得られる。

$$\frac{d\mathbf{r}}{dt} = -\frac{\eta_r}{2} \frac{\partial E}{\partial \mathbf{r}} = \eta_r \cdot \left[{}^\top (\mathbf{x} - \mathbf{r}) - \frac{\lambda}{2} S'(\mathbf{r}) \right] \quad (6.32)$$

ただし, η_r は学習率である。この式により \mathbf{r} が収束するまで最適化するが, 単なる勾配法ではなく, (Olshausen and Field, 1996) では**共役勾配法** (conjugate gradient method) を用いている。しかし, 共役勾配法は実装が煩雑で非効率であるため, より効率的かつ生理学的な妥当性の高い学習法として, **LCA** (locally competitive algorithm) が提案されている (Rozell et al., 2008)。LCA は**側抑制** (local competition, lateral inhibition) と**閾値関数** (thresholding function) を用いる更新則である。LCA による更新を行う RNN は通常の RNN とは異なり, コスト関数(またはエネルギー関数)を最小化する動的システムである。このような機構は Hopfield network で用いられているために, Olshausen は **Hopfield trick** と呼んでいる。

軟判定閾値関数を用いる場合 (ISTA)

$S(x) = |x|$ とした場合の閾値関数を用いる手法として **ISTA** (Iterative Shrinkage Thresholding Algorithm) がある。ISTA は L1-norm 正則化項に対する近接勾配法で, 要は Lasso 回帰に用いる勾配法である。解くべき問題は次式で表される。

$$\mathbf{r} = \arg \min_{\mathbf{r}} \left\{ \|\mathbf{x} - \mathbf{r}\|_2^2 + \lambda \|\mathbf{r}\|_1 \right\} \quad (6.33)$$

詳細は後述するが, 次のように更新することで解が得られる。

- $\mathbf{r}(0)$ を要素が全て 0 のベクトルで初期化 : $\mathbf{r}(0) = \mathbf{0}$
- $\mathbf{r}_*(t+1) = \mathbf{r}(t) + \eta_{\mathbf{r}} \cdot {}^\top(\mathbf{x} - \mathbf{r}(t))$
- $\mathbf{r}(t+1) = \Theta_\lambda(\mathbf{r}_*(t+1))$
- \mathbf{r} が収束するまで 2 と 3 を繰り返す

ここで $\Theta_\lambda(\cdot)$ は軟判定閾値関数 (Soft thresholding function) と呼ばれ、次式で表される。

$$\Theta_\lambda(y) = \begin{cases} y - \lambda & (y > \lambda) \\ 0 & (-\lambda \leq y \leq \lambda) \\ y + \lambda & (y < -\lambda) \end{cases} \quad (6.34)$$

$\Theta_\lambda(\cdot)$ を関数として定義すると次のようになる。また、ReLU (ランプ関数) は $\max(x, 0)$ で実装できる。この点から考えれば ReLU を軟判定非負閾値関数 (soft nonnegative thresholding function) と捉えることもできる (Papyan et al., 2018)。

```
# thresholding function of S(x)=|x|
soft_thres(x, λ) = max(x - λ, 0) - max(-x - λ, 0)
soft_nonneg_thres(x, λ) = max(x - λ, 0) # relu(x-λ)
```

次に $\Theta_\lambda(\cdot)$ を描画すると次のようになる。

```
xmin, xmax = -5, 5
x = range(xmin, xmax, length=100)
y = soft_thres.(x, 1)

figure(figsize=(4,4.5))
subplot(2,2,1)
title(L"$S(x)=|x|$")
plot(x, abs.(x))
xlim(xmin, xmax); ylim(0, 10)
hlines(y=xmax, xmin=xmin, xmax=xmax, color="k", alpha=0.2)
vlines(x=0, ymin=0, ymax=xmax*2, color="k", alpha=0.2)

subplot(2,2,2)
title(L"\frac{\partial S(x)}{\partial x}")
plot(x, x, "k--")
plot(x, sign.(x))
xlim(xmin, xmax); ylim(xmin, xmax)
hlines(y=0, xmin=xmin, xmax=xmax, color="k", alpha=0.2)
vlines(x=0, ymin=xmin, ymax=xmax, color="k", alpha=0.2)

subplot(2,2,3)
title(L"$f_\lambda(x)=x+\lambda\cdot\frac{\partial S(x)}{\partial x}$")
plot(x, x, "k--")
plot(x, x + 1*sign.(x))
xlabel(L"$x$")
xlim(-5, 5); ylim(-5, 5)
hlines(y=0, xmin=xmin, xmax=xmax, color="k", alpha=0.2)
vlines(x=0, ymin=xmin, ymax=xmax, color="k", alpha=0.2)
```

```
subplot(2,2,1)
title(L"\Theta_\lambda(x)")
plot(x, x, "k--")
plot(x, y)
xlabel(L"$x$")
xlim(-5, 5); ylim(-5, 5)
hlines(y=0, xmin=xmin, xmax=xmax, color="k", alpha=0.2)
vlines(x=0, ymin=ymin, ymax=ymax, color="k", alpha=0.2)
tight_layout()
```

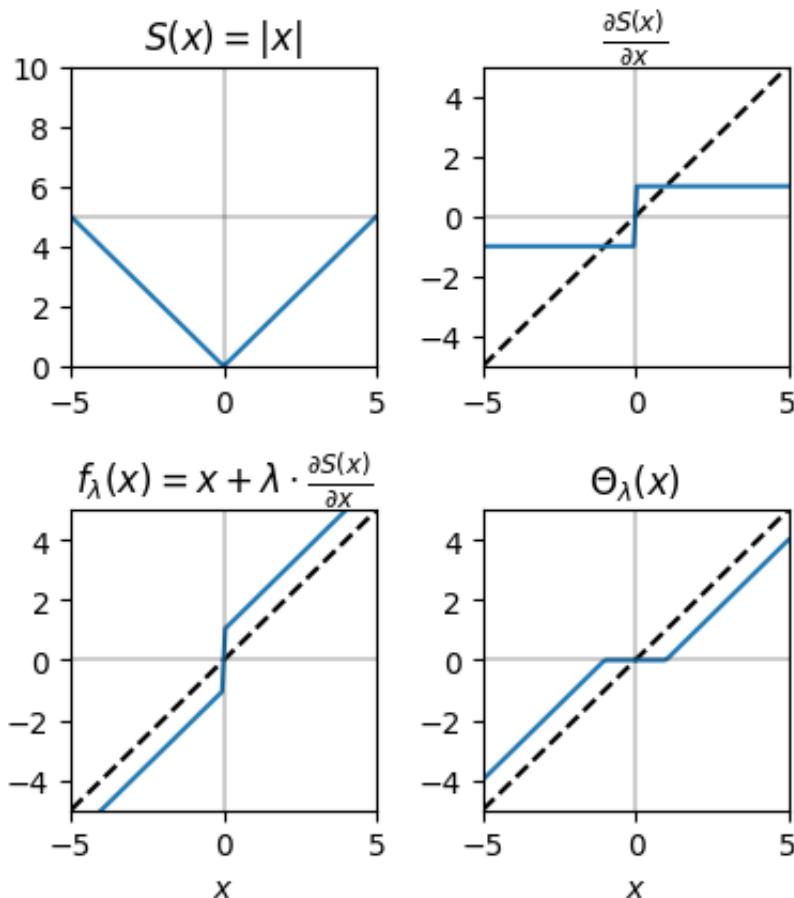


図 6.10 cell009.png

なお、軟判定閾値関数は次の目的関数 C を最小化する x を求めることで導出できる。

$$C = \frac{1}{2}(y - x)^2 + \lambda|x| \quad (6.35)$$

ただし、 x, y, λ はスカラー値とする。 $|x|$ が微分できないが、これは場合分けを考えることで解決する。 $x \geq 0$ を考えると、(6) 式は

$$C = \frac{1}{2}(y - x)^2 + \lambda x = \{x - (y - \lambda)\}^2 + \lambda(y - \lambda) \quad (6.36)$$

となる。(7) 式の最小値を与える x は場合分けをして考えると、 $y - \lambda \geq 0$ のとき二次関数の頂点を考えて $x = y - \lambda$ となる。一方で $y - \lambda < 0$ のときは $x \geq 0$ において単調増加な関数となるので、最小となるのは $x = 0$ のときである。同様の議論を $x \leq 0$ に対しても行うことで(5)式が得られる。なお、閾値関数としては軟判定閾値関数だけではなく、硬判定閾値関数や $y = x - \tanh(x)$ (Tanh-shrink) など様々な関数を用いることができる。

6.4.4 重み行列の更新則

\mathbf{r} が収束したら勾配法により \mathbf{r} を更新する。

$$\Delta\phi_i(\mathbf{x}) = -\eta \frac{\partial E}{\partial} = \eta \cdot [([\mathbf{x} - \mathbf{r}] \mathbf{r}^\top] \quad (6.37)$$

6.4.5 Sparse coding network の実装

ネットワークは入力層を含め 2 層の単純な構造である。今回は、入力はランダムに切り出した $16 \times 16 (= 256)$ の画像パッチとし、これを入力層の 256 個のニューロンが受け取るとする。入力層のニューロンは次層の 100 個のニューロンに投射するとする。100 個のニューロンが入力を Sparse に符号化するようにその活動および重み行列を最適化する。

画像データの読み込み

データは <http://www.rctn.org/bruno/sparsenet/> からダウンロードできる。これはアメリカ北西部で撮影された自然画像であり、van Hateren's Natural Image Dataset <http://bethgelab.org/datasets/vanhateren/> から取得されたものである。IMAGES_RAW.mat は 10 枚の自然画像で、IMAGES.mat はそれを白化したものである。mat ファイルの読み込みには MAT.jl <https://github.com/JuliaIO/MAT.jl> を用いる。

```
using MAT
#using PyPlot
```

```
# datasets from http://www.rctn.org/bruno/sparsenet/
mat_images_raw = matopen("../static/datasets/IMAGES_RAW.mat")
imgs_raw = read(mat_images_raw, "IMAGESt")

mat_images = matopen("../static/datasets/IMAGES.mat")
imgs = read(mat_images, "IMAGES")

close(mat_images_raw)
close(mat_images)
```

画像データを描画する。

```
figure(figsize=(8, 3))
subplots_adjust(hspace=0.1, wspace=0.1)
for i=1:10
    subplot(2, 5, i)
    imshow(imgs_raw[:, :, i], cmap="gray")
    axis("off")
end
suptitle("Natural Images", fontsize=12)
subplots_adjust(top=0.9)
```

Natural Images



図 6.11 cell017.png

モデルの定義

必要なパッケージを読み込む。

```
using Parameters: @unpack # or using UnPack
using LinearAlgebra, Random, Statistics, ProgressMeter
Random.seed!(0)
rc("axes.spines", top=false, right=false)
```

モデルを定義する。

```
@kwdef struct OFParameter{FT}
    lr_r::FT = 1e-2 # learning rate of r
    lr_Phi::FT = 1e-2 # learning rate of Phi
    λ::FT = 5e-3 # regularization parameter
end

@kwdef mutable struct OlshausenField1996Model{FT}
    param::OFParameter = OFParameter{FT}()
    num_inputs::Int32
    num_units::Int32
    batch_size::Int32
    r::Array{FT} = zeros(batch_size, num_units) # activity of neurons
    Phi::Array{FT} = randn(num_inputs, num_units) .* sqrt(1/num_units)
end
```

パラメータを更新する関数を定義する。今回はより生理学的に妥当にするため、軟判定非負閾値関数を用いる。

```
function updateOF!(variable::OlshausenField1996Model, param::OFParameter, ~
    inputs::Array, training::Bool)
    @unpack num_inputs, num_units, batch_size, r, Phi = variable
    @unpack lr_r, lr_Phi, λ = param

    # Updates
    error = inputs .- r * Phi'
    r_ = r + lr_r .* error * Phi

    #r[:, :] = soft_thres.(r_, λ)
    r[:, :] = soft_nonneg_thres.(r_, λ)

    if training
        error = inputs - r * Phi'
        dPhi = error' * r
        Phi[:, :] += lr_Phi * dPhi
    end

    return error
end
```

行ごとに正規化する関数を定義する。

```
function normalize_rows(A::Array)
    return A ./ sqrt.(sum(A.^2, dims=1) .+ 1e-8)
end
```

損失関数を定義する。

```
function calculate_total_error(error, r, λ)
    recon_error = mean(error.^2)
    sparsity_r = λ*mean(abs.(r))
    return recon_error + sparsity_r
end
```

シミュレーションを実行する関数を定義する。外側の `for loop` では画像パッチの作成と `r` の初期化を行う。内側の `for loop` では `r` が収束するまで更新を行い、収束したときに重み行列 `Phi` を更新する。

```
function run_simulation(imgs, num_iter, nt_max, batch_size, sz, num_units, eps)
    H, W, num_images = size(imgs)
    num_inputs = sz^2

    model = OlshausenField1996Model{Float32}(num_inputs=num_inputs, -
        num_units=num_units, batch_size=batch_size)
    errorarr = zeros(num_iter) # Vector to save errors

    # Run simulation
    @showprogress "Computing..." for iter in 1:num_iter
        # Get the coordinates of the upper left corner of clopping image randomly.
        beginx = rand(1:W-sz, batch_size)
        beginy = rand(1:H-sz, batch_size)

        inputs = zeros(batch_size, num_inputs) # Input image patches

        # Get images randomly
        for i in 1:batch_size
            idx = rand(1:num_images)
            img = imgs[:, :, idx]
            clop = img[beginy[i]:beginy[i]+sz-1, beginx[i]:beginx[i]+sz-1,:]
            inputs[i, :] = clop .- mean(clop)
        end

        model.r = zeros(batch_size, num_units) # Reset r states
        model.Phi = normalize_rows(model.Phi) # Normalize weights
        # Input image patches until latent variables are converged
        r_tm1 = zeros(batch_size, num_units) # set previous r (t minus 1)

        for t in 1:nt_max
            # Update r without update weights
            error = updateOF!(model, model.param, inputs, false)

            dr = model.r - r_tm1

            # Compute norm of r
            dr_norm = sqrt(sum(dr.^2)) / sqrt(sum(r_tm1.^2)) + 1e-8
            r_tm1 .= model.r # update r_tm1

            # Check convergence of r, then update weights
            if dr_norm < eps
                error = updateOF!(model, model.param, inputs, true)
            end
        end
    end
end
```

```

        errorarr[iter] = calculate_total_error(error, model.r, +
            model.param.λ) # Append errors
        break
    end

    # If failure to convergence, break and print error
    if t >= nt_max-1
        print("Error at patch:", iter_, dr_norm)
        errorarr[iter] = calculate_total_error(error, model.r, +
            model.param.λ) # Append errors
        break
    end
end
# Print moving average error
if iter % 100 == 0
    moving_average_error = mean(errorarr[iter-99:iter])
    println("iter: ", iter, "/", num_iter, ", Moving average error:", +
        moving_average_error)
end
end
return model, errorarr
end

```

`r_tm1 .= model.r` の部分は、要素ごとのコピーを実行している。`r_tm1 = copy(model.r)`でもよいが、新たなメモリ割り当てが生じるので避けている。`@.r_tm1 = model.r` としてもよい。シミュレーションの実行をする。

```

# Simulation constants
num_iter = 500 # number of iterations
nt_max = 1000 # Maximum number of simulation time
batch_size = 250 # Batch size

sz = 16 # image patch size
num_units = 100 # number of neurons (units)
eps = 1e-2 # small value which determines convergence

model, errorarr = run_simulation(imgs, num_iter, nt_max, batch_size, sz, num_units, +
    eps);

```

訓練中の損失の描画

訓練中の損失の変化を描画してみよう。損失が低下し、学習が進行したことが分かる。

```

# Plot error
figure(figsize=(4, 2))
ylabel("Error")
xlabel("Iterations")
plot(1:num_iter, errorarr)
tight_layout()

```

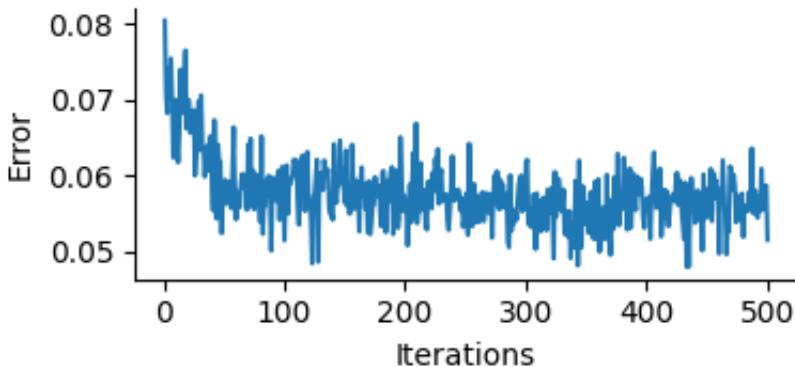


図 6.12 cell033.png

重み行列 (受容野) の描画

学習後の重み行列 $\Phi()$ を可視化してみよう.

```
# Plot Receptive fields
figure(figsize=(4.2, 4))
subplots_adjust(hspace=0.1, wspace=0.1)
for i in 1:num_units
    subplot(10, 10, i)
    imshow(reshape(model.Phi[:, i], (sz, sz)), cmap="gray")
    axis("off")
end
suptitle("Receptive fields", fontsize=14)
subplots_adjust(top=0.925)
```

白色が **ON** 領域 (興奮), 黒色が **OFF** 領域 (抑制) を表す. Gabor フィルタ様の局所受容野が得られており, これは一次視覚野 (V1) における単純型細胞 (simple cells) の受容野に類似している.

画像の再構成

学習したモデルを用いて入力画像が再構成されるか確認しよう.

```
H, W, num_images = size(imgs)
num_inputs = sz^2

# Get the coordinates of the upper left corner of clopping image randomly.
beginx = rand(1:W-sz, batch_size)
beginy = rand(1:H-sz, batch_size)

inputs = zeros(batch_size, num_inputs) # Input image patches

# Get images randomly
```

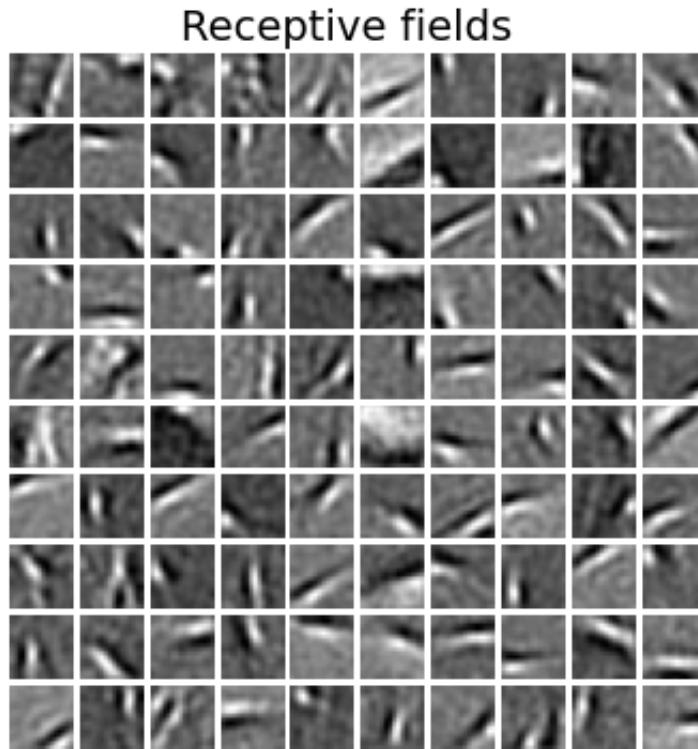


図 6.13 cell035.png

```

for i in 1:batch_size
    idx = rand(1:num_images)
    img = imgs[:, :, idx]
    clop = img[beginy[i]:beginy[i]+sz-1, beginx[i]:beginx[i]+sz-1][:]
    inputs[i, :] = clop .- mean(clop)
end

model.r = zeros(batch_size, num_units) # Reset r states

# Input image patches until latent variables are converged
r_tm1 = zeros(batch_size, num_units) # set previous r (t minus 1)

for t in 1:nt_max
    # Update r without update weights
    error = updateOF!(model, model.param, inputs, false)

    dr = model.r - r_tm1

    # Compute norm of r
    dr_norm = sqrt(sum(dr.^2)) / sqrt(sum(r_tm1.^2) + 1e-8)
    r_tm1 .= model.r # update r_tm1

```

```
# Check convergence of r, then update weights
if dr_norm < eps
    break
end
end;
```

神経活動 r がスペースになっているか確認しよう。

```
figure(figsize=(3, 2))
hist(model.r[:,], bins=50)
xlim(0, 0.5)
tight_layout()
```

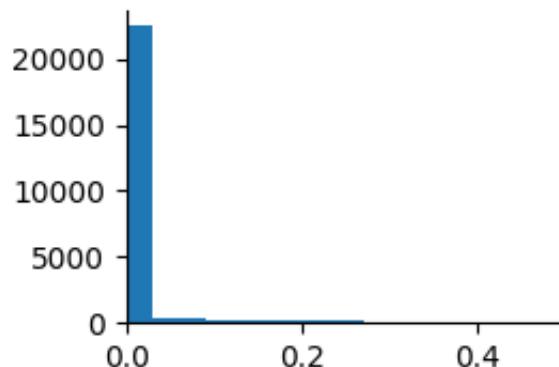


図 6.14 cell040.png

要素がほとんど 0 のスペースなベクトルになっていることがわかる。次に画像を再構成する。

```
reconst = model.r * model.Phi'
println(size(reconst))
```

再構成した結果を描画する。

```
figure(figsize=(7.5, 3))
subplots_adjust(hspace=0.1, wspace=0.1)
num_show = 5
for i in 1:num_show
    subplot(2, num_show, i)
    imshow(reshape(inputs[i, :], (sz, sz)), cmap="gray")
    xticks([]); yticks([]);
    if i == 1
        ylabel("Input\n images")
    end
end
```

```
subplot(2, num_show, num_show+i)
imshow(reshape(reconst[i, :], (sz, sz)), cmap="gray")
xticks([]); yticks([]);
if i == 1
    ylabel("Reconstructed\n images")
end
end
```

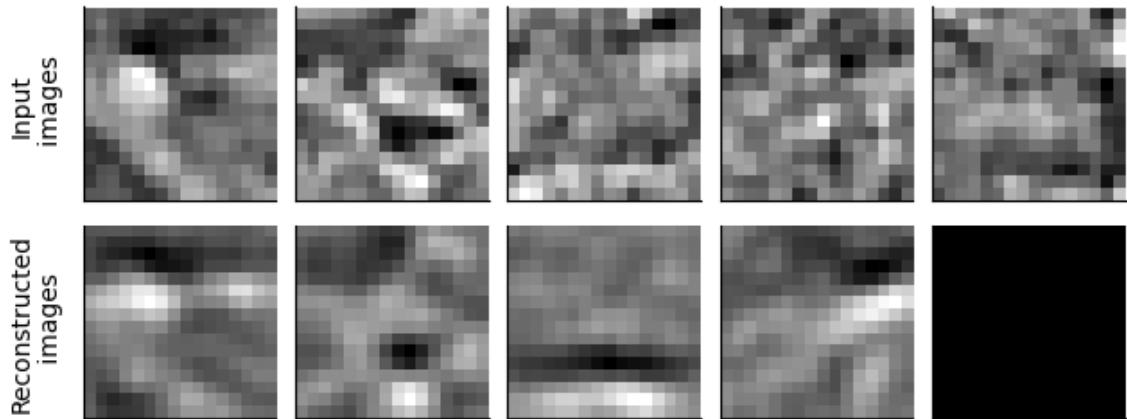


図 6.15 cell044.png

上段が入力画像、下段が再構成された画像である。差異はあるものの、概ね再構成されていることがわかる。論文以外の参考資料

- http://www.scholarpedia.org/article/Sparse_coding
- Bruno Olshausen: “Sparse coding in brains and machines” (<https://talks.stanford.edu/bruno-olshausen-sparse-coding-in-brains-and-machines/>), <http://www.rctn.org/bruno/public/Simons-sparse-coding.pdf>
- <https://redwood.berkeley.edu/wp-content/uploads/2018/08/sparse-coding-ICA.pdf>
- <https://redwood.berkeley.edu/wp-content/uploads/2018/08/sparse-coding-LCA.pdf>
- https://redwood.berkeley.edu/wp-content/uploads/2018/08/Dylan-lca-overcompleteness_09-27-2018.pdf

6.5 予測符号化

6.5.1 観測世界の階層的予測

階層的予測符号化 (hierarchical predictive coding; HPC) は (Rao and Ballard, 1999) により導入された。構築するネットワークは入力層を含め、3層のネットワークとする。LGNへの入力として画像 $\mathbf{x} \in \mathbb{R}^{n_0}$ を考える。画像 \mathbf{x} の観測世界における隠れ変数、すなわち潜在変数 (latent variable) を $\mathbf{r} \in \mathbb{R}^{n_1}$ とし、ニューロン群によって発火率で表現されているとする (真の変数と \mathbf{r} は異なるので文字を分けるべきだが簡単のためにこう表す)。このとき、

$$\mathbf{x} = f(\mathbf{Ur}) + \epsilon \quad (6.38)$$

が成立しているとする。ただし、 $f(\cdot)$ は活性化関数 (activation function)、 $\mathbf{U} \in \mathbb{R}^{n_0 \times n_1}$ は重み行列である。 $\epsilon \in \mathbb{R}^{n_0}$ は $\mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$ からサンプリングされるとする。潜在変数 \mathbf{r} はさらに高次 (higher-level) の潜在変数 \mathbf{r}^h により、次式で表現される。

$$\mathbf{r} = \mathbf{r}^{td} + \epsilon^{td} = f(\mathbf{U}^h \mathbf{r}^h) + \epsilon^{td} \quad (6.39)$$

ただし、Top-down の予測信号を $\mathbf{r}^{td} := f(\mathbf{U}^h \mathbf{r}^h)$ とした。また、 $\mathbf{r}^{td} \in \mathbb{R}^{n_1}$, $\mathbf{r}^h \in \mathbb{R}^{n_2}$, $\mathbf{U}^h \in \mathbb{R}^{n_1 \times n_2}$ である。 $\epsilon^{td} \in \mathbb{R}^{n_1}$ は $\mathcal{N}(\mathbf{0}, \sigma_{td}^2 \mathbf{I})$ からサンプリングされるとする。話は飛ぶが、Predictive coding のネットワークの特徴は

- 階層的な構造
- 高次による低次の予測 (Feedback or Top-down 信号)
- 低次から高次への誤差信号の伝搬 (Feedforward or Bottom-up 信号)

である。ここまででは高次表現による低次表現の予測、という Feedback 信号について説明してきたが、この部分は Sparse coding でも同じである。それでは Predictive coding のもう一つの要となる、低次から高次への予測誤差の伝搬という Feedforward 信号はどのように導かれるのだろうか。結論から言えば、これは復元誤差 (reconstruction error) の最小化を行う再帰的ネットワーク (recurrent network) を考慮することで自然に導かれる。

6.5.2 損失関数と学習則

事前分布の設定

\mathbf{r} の事前分布 $p(\mathbf{r})$ は Cauchy 分布を用いる。 $p(\mathbf{r})$ の負の対数事前分布を $g(\mathbf{r}) := -\log p(\mathbf{r})$ としておく。

$$p(\mathbf{r}) = \prod_i p(r_i) = \prod_i \exp[-\alpha \ln(1 + r_i^2)] \quad (6.40)$$

$$g(\mathbf{r}) = -\ln p(\mathbf{r}) = \alpha \sum_i \ln(1 + r_i^2) \quad (6.41)$$

$$g'(\mathbf{r}) = \frac{\partial g(\mathbf{r})}{\partial \mathbf{r}} = \left[\frac{2\alpha r_i}{1 + r_i^2} \right]_i \quad (6.42)$$

次に重み行列 \mathbf{U} の事前分布 $p(\mathbf{U})$ は Gaussian 分布とする。 $p(\mathbf{U})$ の負の対数事前分布を $h(\mathbf{U}) := -\ln p(\mathbf{U})$ とすると、次のように表される。

$$p(\mathbf{U}) = \exp(-\lambda \|\mathbf{U}\|_F^2) \quad (6.43)$$

$$h(\mathbf{U}) = -\ln p(\mathbf{U}) = \lambda \|\mathbf{U}\|_F^2 \quad (6.44)$$

$$h'(\mathbf{U}) = \frac{\partial h(\mathbf{U})}{\partial \mathbf{U}} = 2\lambda \mathbf{U} \quad (6.45)$$

ただし、 $\|\cdot\|_F^2$ はフロベニウスノルムを意味する。

損失関数の設定

Sparse coding と同様に考えることにより、損失関数 E を次のように定義する。

$$E = \underbrace{\frac{1}{\sigma^2} \|\mathbf{x} - f(\mathbf{Ur})\|^2 + \frac{1}{\sigma_{td}^2} \|\mathbf{r} - f(\mathbf{U}^h \mathbf{r}^h)\|^2}_{\text{reconstruction error}} + \underbrace{g(\mathbf{r}) + g(\mathbf{r}^h) + h(\mathbf{U}) + h(\mathbf{U}^h)}_{\text{sparsity penalty}} \quad (6.46)$$

潜在変数 \mathbf{r}, \mathbf{r}^h と 重み行列 \mathbf{U}, \mathbf{U}^h のそれぞれに事前分布を仮定しているため、これらについての MAP 推定を行うことに相当する。

再帰ネットワークの更新則

簡単のために $\mathbf{z} := \mathbf{Ur}, \mathbf{z}^h := \mathbf{U}^h \mathbf{r}^h$ とする。

$$\frac{d\mathbf{r}}{dt} = -\frac{k_1}{2} \frac{\partial E}{\partial \mathbf{r}} = k_1 \cdot \left(\frac{1}{\sigma^2} \mathbf{U}^T \left[\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \odot \underbrace{(\mathbf{x} - f(\mathbf{z}))}_{\text{bottom-up error}} \right] - \frac{1}{\sigma_{td}^2} \underbrace{(\mathbf{r} - f(\mathbf{z}^h))}_{\text{top-down error}} - \frac{1}{2} g'(\mathbf{r}) \right) \quad (6.47)$$

$$\frac{d\mathbf{r}^h}{dt} = -\frac{k_1}{2} \frac{\partial E}{\partial \mathbf{r}^h} = k_1 \cdot \left(\frac{1}{\sigma_{td}^2} (\mathbf{U}^h)^T \left[\frac{\partial f(\mathbf{z}^h)}{\partial \mathbf{z}^h} \odot \underbrace{(\mathbf{r} - f(\mathbf{z}^h))}_{\text{bottom-up error}} \right] - \frac{1}{2} g'(\mathbf{r}^h) \right) \quad (6.48)$$

ただし, k_1 は更新率 (updating rate) である. または, 発火率の時定数を $\tau := 1/k_1$ として, k_1 は発火率の時定数 τ の逆数であると捉えることもできる. ここで 1 番目の式において, 中間表現 \mathbf{r} のダイナミクスは bottom-up error と top-down error で記述されている. このように bottom-up error が \mathbf{r} への入力となることは自然に導出される. なお, top-down error に関しては高次からの予測 (prediction) の項 $f(\mathbf{x}^h)$ と leaky-integrator としての項 $-\mathbf{r}$ に分割することができる. また $\mathbf{U}^\top, (\mathbf{U}^h)^\top$ は重み行列の転置となっており, bottom-up と top-down の投射において対称な重み行列を用いることを意味している. $-g'(\mathbf{r})$ は発火率を抑制してスペースにすることを目的とする項だが, 無理やり解釈をすると自己再帰的な抑制と言える.

画像データの読み込み

「スペース符号化」と同様にデータは <http://www.rctn.org/bruno/sparsenet/> からダウンロードできるファイルを用いる.

```
using MAT

# datasets from http://www.rctn.org/bruno/sparsenet/
mat_images = matopen("../_static/datasets/IMAGES.mat")
imgs = read(mat_images, "IMAGES")

close(mat_images)
```

モデルの定義

必要なパッケージを読み込む.

```
using Parameters: @unpack # or using UnPack
using LinearAlgebra, Random, Statistics, PyPlot, ProgressMeter
```

モデルを定義する.

```
@kwdef struct RBParameter{FT}
    α::FT = 1.0
    αh::FT = 0.05
    σ²::FT = 1.0
    σ²td::FT = 10
    σ⁻²::FT = 1/σ²
    σ⁻²td::FT = 1/σ²td
    k₁::FT = 0.3 # k_1: update rate
    λ::FT = 0.02 # regularization parameter
end

@kwdef mutable struct RaoBallard1999Model{FT}
    param::RBParameter = RBParameter{FT}()
    num_units_lv0::UInt16 = 256 # number of units of level0
    num_units_lv1::UInt16 = 32
```

```

num_units_lv2::UInt16 = 128
num_lv1::UInt16 = 3
k2::FT = 0.2 # k_2: learning rate
r::Array{FT} = zeros(num_lv1, num_units_lv1) # activity of neurons
rh::Array{FT} = zeros(num_units_lv2) # activity of neurons
U::Array{FT} = randn(num_units_lv0, num_units_lv1) .* sqrt(2.0 / ~
    (num_units_lv0+num_units_lv1))
Uh::Array{FT} = randn(num_lv1*num_units_lv1, num_units_lv2) .* sqrt(2.0 / ~
    (num_lv1*num_units_lv1+num_units_lv2))
end

```

パラメータを更新する関数を定義する。

```

function update!(variable::RaoBallard1999Model, param::RBParameter, inputs::Array, ~
    training::Bool)
    @unpack num_units_lv0, num_units_lv1, num_units_lv2, num_lv1, k2, r, rh, U, Uh = ~
        variable
    @unpack α, αh, σ⁻², σ⁻²td, k₁, λ = param

    r_reshaped = r[:] # (96)

    fx = r * U' # (3, 256)
    fxh = Uh * rh # (96, )

    # Calculate errors
    error = inputs - fx # (3, 256)
    errorh = r_reshaped - fxh # (96, )
    errorh_reshaped = reshape(errorh, (num_lv1, num_units_lv1)) # (3, 32)

    g_r = α * r ./ (1.0 .+ r .^ 2) # (3, 32)
    g_rh = αh * rh ./ (1.0 .+ rh .^ 2) # (64, )

    # Update r and rh
    dr = k₁ * (σ⁻² * error * U - σ⁻²td * errorh_reshaped - g_r)
    drh = k₁ * (σ⁻²td * Uh' * errorh - g_rh)

    r[:, :] += dr
    rh[:] += drh

    if training
        U[:, :] += k₂ * (σ⁻² * error' * r - num_lv1 * λ * U)
        Uh[:, :] += k₂ * (σ⁻²td * errorh * rh' - λ * Uh)
    end
end

return error, errorh, dr, drh
end

```

入力に乘じる Gaussian フィルタを定義する。

```

# Gaussian mask for inputs
function gaussian_2d(sizex=16, sizey=16, sigma=5)
    x, y = 0:sizex-1, 0:sizey-1
    x0, y0 = (sizex-1)/2, (sizey-1)/2

```

```

f(x,y) = exp(-((x-x0)^2 + (y-y0)^2) / (2.0*(sigma^2)))
gau = f.(x', y)
return gau ./ sum(gau)
end

```

```

gau = gaussian_2d()
figure(figsize=(2,2))
title("Gaussian mask")
imshow(gau)
tight_layout()

```

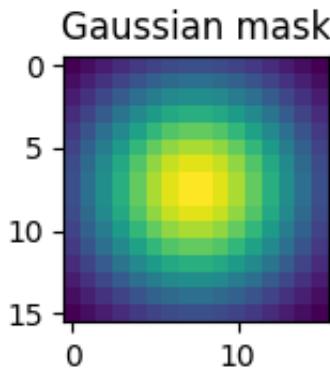


図 6.16 cell012.png

損失関数を定義する。

```

function calculate_total_error(error, errorh, variable::RaoBallard1999Model, +
param::RBParameter)
@unpack r, rh, U, Uh = variable
@unpack α, αh, σ⁻², σ⁻²td, k₁, λ = param
recon_error = σ⁻² * sum(error.^2) + σ⁻²td * sum(errorh.^2)
sparsity_r = α * sum(r.^2) + αh * sum(rh.^2)
sparsity_U = λ * (sum(U.^2) + sum(Uh.^2))
return recon_error + sparsity_r + sparsity_U
end;

```

シミュレーションを実行する関数を定義する。外側の **for** loop では画像パッチの作成と **r** の初期化を行う。内側の **for** loop では **r** が収束するまで更新を行い、収束したときに重み行列 **Phi** を更新する。

```

function run_simulation(imgs, num_iter, nt_max, eps)
# Define model
model = RaoBallard1999Model{Float32}()

# Simulation constants

```

```

H, W, num_images = size(imgs)
input_scale = 40 # scale factor of inputs
gmask = gaussian_2d() # Gaussian mask
errorarr = zeros(num_iter) # Vector to save errors

# Run simulation
@showprogress "Computing..." for iter in 1:num_iter
    # Get images randomly
    idx = rand(1:num_images)
    img = imgs[:, :, idx]

    # Get the coordinates of the upper left corner of clopping image randomly.
    beginx = rand(1:W-27)
    beginy = rand(1:H-17)
    img_clopped = img[beginy:beginy+15, beginx:beginx+25]

    # Clop three patches
    inputs = stack([(gmask .* img_clopped[:, 1+i*5:i*5+16])[:] for i = 0:2])
    inputs = (inputs .- mean(inputs)) .* input_scale

    # Reset states
    model.r = inputs * model.U
    model.rh = model.Uh' * model.r[:]

    # Input an image patch until latent variables are converged
    for i in 1:nt_max
        # Update r and rh without update weights
        error, errorh, dr, drh = update!(model, model.param, inputs, false)

        # Compute norm of r and rh
        dr_norm = sqrt(sum(dr.^2))
        drh_norm = sqrt(sum(drh.^2))

        # Check convergence of r and rh, then update weights
        if dr_norm < eps && drh_norm < eps
            error, errorh, dr, drh = update!(model, model.param, inputs, true)
            errorarr[iter] = calculate_total_error(error, errorh, model, ~
                model.param) # Append errors
            break
        end

        # If failure to convergence, break and print error
        if i >= nt_max-2
            println("Error at patch:", iter)
            println(dr_norm, drh_norm)
            break
        end
    end

    # Decay learning rate
    if iter % 40 == 39
        model.k_z /= 1.015
    end

```

```

# Print moving average error
if iter % 1000 == 0
    moving_average_error = mean(errorarr[iter-999:iter])
    println("[", iter, "/", num_iter, "] Moving average error:", -
        moving_average_error)
end
end
return model, errorarr
end

```

シミュレーションの実行をする

```

# Simulation constants
num_iter = 5000 # number of iterations
nt_max = 1000 # Maximum number of simulation time
eps = 1e-3 # small value which determines convergence

model, errorarr = run_simulation(imgs, num_iter, nt_max, eps);

```

訓練中の損失の描画

訓練中の損失の変化を描画してみよう。損失が低下し、学習が進行したことが分かる。

```

function moving_average(x, n=100)
    ret = cumsum(x)
    ret[n:end] = ret[n:end] - ret[1:end-n+1]
    return ret[n - 1:end] / n
end

# Plot error
moving_average_error = moving_average(errorarr)
figure(figsize=(4, 2))
ylabel("Moving error")
xlabel("Iterations")
plot(1:size(moving_average_error)[1], moving_average_error)
tight_layout()

```

重み行列（受容野）の描画

学習後の重み行列 (U) を可視化してみよう。

```

# Plot Receptive fields
figure(figsize=(6, 3))
subplots_adjust(hspace=0.1, wspace=0.1)
for i in 1:32
    subplot(4, 8, i)
    imshow(reshape(model.U[:, i], (16, 16)), cmap="gray")
    axis("off")
end

```

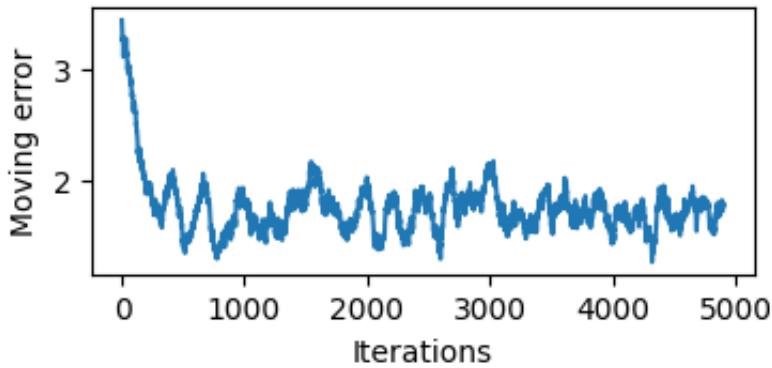


図 6.17 cell020.png

```
suptitle("Receptive fields of level 1", fontsize=14)
subplots_adjust(top=0.9)
```

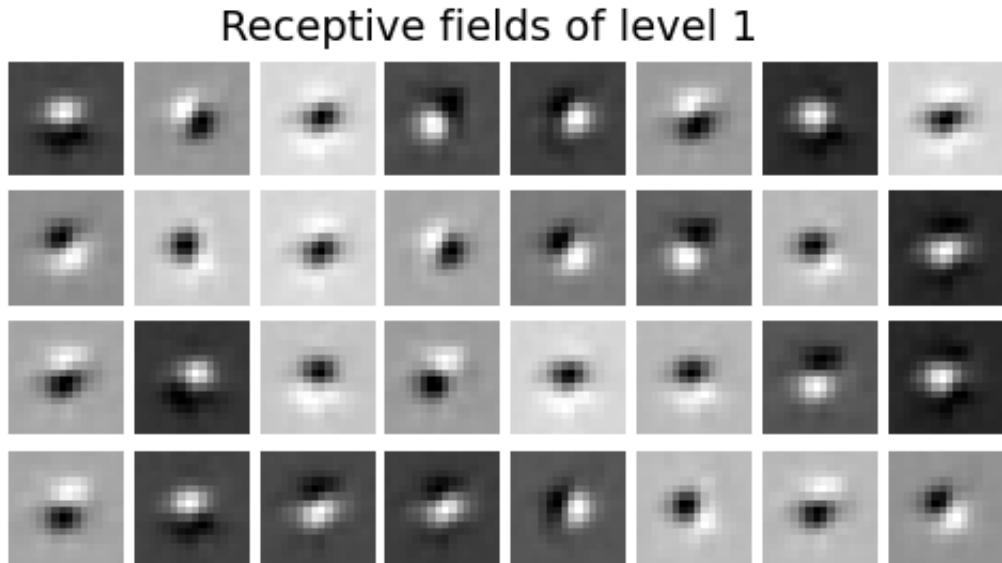


図 6.18 cell022.png

白色が **ON** 領域 (興奮), 黒色が **OFF** 領域 (抑制) を表す. Gabor フィルタ様の局所受容野が得られている. 次に, Level2 のニューロンの受容野は \mathbf{U} と \mathbf{U}^h の積を計算することで描画できる.

```
# Plot Receptive fields of level 2
```

```
zero_padding = zeros(80, 32)
U0 = [model.U; zero_padding; zero_padding]
U1 = [zero_padding; model.U; zero_padding]
U2 = [zero_padding; zero_padding; model.U]
U_ = [U0 U1 U2]
Uh_ = U_ * model.Uh

figure(figsize=(7, 3))
subplots_adjust(hspace=0.1, wspace=0.1)
for i in 1:24
    subplot(4, 6, i)
    imshow(reshape(Uh_[:, i], (16, 26)), cmap="gray")
    axis("off")
end

suptitle("Receptive fields of level 2", fontsize=14)
subplots_adjust(top=0.9)
```

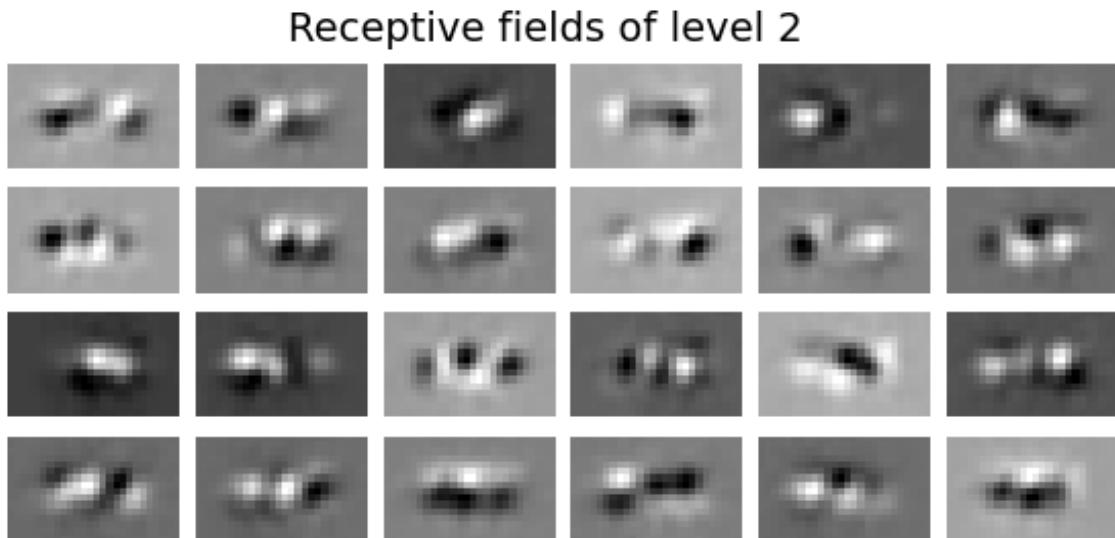


図 6.19 cell024.png

参考文献

- Amari, S.-I. (1972). “Learning Patterns and Pattern Sequences by Self-Organizing Nets of Threshold Elements”. *IEEE Trans. Comput.* C-21.11, pp. 1197–1206.
- Hopfield, J. J. (1982). “Neural networks and physical systems with emergent collective computational abilities”. *Proc. Natl. Acad. Sci. U. S. A.* 79.8, pp. 2554–2558.

- LeCun, Y. et al. (2006). “A tutorial on energy-based learning”. *Predicting structured.*
- Olshausen, B. A. and Field, D. J. (1996). “Emergence of simple-cell receptive field properties by learning a sparse code for natural images”. *Nature* 381.6583, pp. 607–609.
- (1997). “Sparse coding with an overcomplete basis set: a strategy employed by V1?” *Vision Res.* 37.23, pp. 3311–3325.
- Papyan, V. et al. (2018). “Theoretical Foundations of Deep Learning via Sparse Representations: A Multilayer Sparse Model and Its Connection to Convolutional Neural Networks”. *IEEE Signal Process. Mag.* 35.4, pp. 72–89.
- Rao, R. P. and Ballard, D. H. (1999). “Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects”. *Nat. Neurosci.* 2.1, pp. 79–87.
- Rozell, C. J. et al. (2008). “Sparse coding via thresholding and local competition in neural circuits”. *Neural Comput.* 20.10, pp. 2526–2563.

第 7 章

貢献度分配問題の解決策

7.1 勾配法と誤差逆伝播法

誤差逆伝播法 (back-propagation)

7.1.1 ニューラルネットワークモデル

この節では入力層、隠れ層、出力層からなる 3 層ニューラルネットワークを実装する。

```
using Base: @kwdef
using Parameters: @unpack # or using UnPack
using LinearAlgebra, Random, Statistics, PyPlot, ProgressMeter
```

```
abstract type NeuralNet end

@kwdef struct MLP <: NeuralNet
    L::Int # num. of layers
    f::Vector{Function}; ∇f::Vector{Function};
    params::Dict{String, Dict} # weights and bias
    grads::Dict{String, Dict} # gradient of params
    states::Dict{String, Dict} # state of forward/backward activity
end;
```

```
function MLP(num_units; activation="sigmoid")#, bias=true)
    L = length(num_units) - 1 # num of layers
    # initialization of parameters
    params, grads = Dict(), Dict()
    params["W"] = Dict{Int, Array}{l => 2 * (rand(num_units[l], num_units[l+1]) .-
        0.5) / sqrt(num_units[l]) for l in 1:L}
    params["b"] = Dict{Int, Array}{l => zeros(1, num_units[l+1]) for l in 1:L}
    for key in keys(params)
        grads["∇$key"] = Dict{Int, Array}{l => zero(params[key][l]) for l in 1:L}
    end
end
```

```

states = Dict(key => Dict{Int, Array}{}() for key in ["a", "z", "δ"])

# set activation functions
if activation isa Vector{String}
    @assert length(activation) == L "length of activation must be equal to L=$L, or use string"
    f = [eval(Symbol(activation[l])) for l in 1:L]
    ∇f = [eval(Symbol("∇$(activation[l])")) for l in 1:L]
elseif activation isa String
    f = [eval(Symbol(activation)) for l in 1:L]
    ∇f = [eval(Symbol("∇$(activation)")) for l in 1:L]
end
return MLP(L=L, f=f, ∇f=∇f, params=params, grads=grads, states=states)
end;

```

mutable struct **MLP** を用意し、**重みの初期化 (weight initialization)** を行う同名の関数 **MLP** を用意する。重みの初期化の手法は複数あるが、ここでは重みを W として、 $W_{ij} \sim U(-1/\sqrt{n}, 1/\sqrt{n})$ とする (Xavier initialization) (Glorot and Bengio, 2010)。ただし、 n は入力ユニット数である。

順伝播 (forward propagation)

$f(\cdot)$ を活性化関数とする。順伝播 (feedforward propagation) は以下のようになる。

$$\text{入力層 : } \mathbf{z}^{(0)} = \mathbf{x} \quad (7.1)$$

$$\text{隠れ層 : } \mathbf{z}^{(\ell)} = f\left(\mathbf{a}^{(\ell)}\right) \quad (7.2)$$

$$\mathbf{a}^{(\ell+1)} = W^{(\ell+1)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)} \quad (7.3)$$

$$\text{出力層 : } \hat{\mathbf{y}} = \mathbf{z}^{(L)} \quad (7.4)$$

```

sigmoid(x) = 1 ./ (1 .+ exp.(-x));
relu(x) = max.(x, 0);

∇sigmoid(z) = z .* (1 .- z)
∇relu(z) = z .> 0
∇tanh(z) = 1 .- z.^2

```

```

function softmax(x; dims=1)
    expx = exp.(x .- maximum(x))
    return expx ./ sum(expx, dims=dims)
end

```

```
#∇softmax
```

最後に活性化関数を付けたくない場合は恒等関数 **identity** を用いる。

```

function forward!(mlp::MLP, x::Array)
    @unpack L, params, states, f = mlp
    @unpack W, b = params # parameters
    @unpack a, z = states # state of forward/backward activity

    z[0] = x
    for l in 1:L
        a[l] = z[l-1] * W[l] .+ b[l]
        z[l] = f[l](a[l])
    end
    return z[L]
end

```

逆伝播 (backward propagation)

$$\text{目的関数} : \mathcal{L} = \frac{1}{2} \|\hat{\mathbf{y}} - \mathbf{y}\|^2 \quad (7.5)$$

$$\text{最急降下法} : \Delta W^{(\ell)} = -\eta \frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \quad (7.6)$$

$$\Delta \mathbf{b}^{(\ell)} = -\eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} \quad (7.7)$$

$$\text{誤差逆伝播法} : \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} = \hat{\mathbf{y}} - \mathbf{y} \quad (7.8)$$

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{a}^{(L)}} = (\hat{\mathbf{y}} - \mathbf{y}) \odot f' \left(\mathbf{a}^{(L)} \right) \quad (7.9)$$

$$\delta^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{a}^{(\ell)}} = \left(W^{(\ell+1)} \right)^T \delta^{(\ell+1)} \odot f' \left(\mathbf{a}^{(\ell)} \right) \quad (7.10)$$

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{a}^{(\ell)}} \frac{\partial \mathbf{a}^{(\ell)}}{\partial W^{(\ell)}} = \delta^{(\ell)} \left(\mathbf{z}^{(\ell-1)} \right)^T \quad (7.11)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{a}^{(\ell)}} \frac{\partial \mathbf{a}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}} = \delta^{(\ell)} \quad (7.12)$$

バッチ処理を考慮すると、行列を乗ずる順番が変わる。以下では $z = f(a), g(z) = f'(a)$ として膜電位を使わず、発火率情報のみを使うようにしている。このようにできない関数もあるが、今回はこのように書き下せる活性化関数のみを扱う。

$$\frac{d}{dx} \text{Sigmoid}(x) = \text{Sigmoid}(x) \cdot (1 - \text{Sigmoid}(x)) \quad (7.13)$$

であることに注意。

```

function backward!(mlp::MLP; losstype::String="binary_crossentropy")
    @unpack L, params, states, grads, ∇f = mlp
    @unpack W, b = params # parameters

```

```

@unpack ∇W, ∇b = grads # gradient of params
@unpack a, z, δ = states # state of forward/backward activity

n_batch = size(z[0])[1]
# backprop
for l in L:-1:1
    if l > 1
        δ[l-1] = δ[l] * W[l]' .* ∇f[l].(z[l-1])
    end
    ∇W[l] = z[l-1]' * δ[l] / n_batch
    ∇b[l] = sum(δ[l], dims=1) / n_batch
end
end

clog(x) = max(log(x), -1e2) # clamped log

function binary_crossentropy!(nn::NeuralNet, y^::Array, y::Array)
    @unpack L, states = nn
    @unpack δ = states
    error = y^ - y
    loss = sum(-y .* clog.(y^) + (1 .- y) .* clog.(1 .- y^))
    δ[L] = error
    return loss
end

function squared_error!(nn::NeuralNet, y^::Array, y::Array)
    @unpack L, states, ∇f = mlp
    @unpack δ = states
    error = y^ - y
    loss = sum(error .^ 2)
    δ[L] = error .* ∇f[L].(y^)
    return loss
end

```

Optimizer の作成

abstract type として Optimizer タイプを作成する。

```
abstract type Optimizer end
```

確率的勾配降下法 (stochastic gradient descent; SGD) を実装する。

```

# SGD optimizer
@kwdef struct SGD{FT} <: Optimizer
    η::FT = 1e-2
end

function optimizer_update!(param, grad, optimizer::SGD)
    @unpack η = optimizer
    param[:, :] -= η * grad

```

```
end
```

次に Adam (Kingma and Ba, 2014) を実装する.

```
# Adam optimizer
@kwdef mutable struct Adam{FT} <: Optimizer
    α::FT = 1e-4; β1::FT = 0.9; β2::FT = 0.999; ε::FT = 1e-8
    ms = Dict(); vs = Dict();
end

# Adam optimizer
function optimizer_update!(param, grad, optimizer::Adam)
    @unpack α, β1, β2, ε, ms, vs = optimizer
    key = objectid(param)
    if !haskey(ms, key)
        ms[key], vs[key] = zeros(size(param)), zeros(size(param))
    end
    m, v = ms[key], vs[key]
    m += (1 - β1) * (grad - m)
    v += (1 - β2) * (grad .* grad - v)
    param[:, :] -= α * m ./ (sqrt.(v) .+ ε)
end

function optim_step!(nn::NeuralNet, optimizer::Optimizer)
    @unpack L, params, grads = nn
    for key in keys(params)
        for l in 1:L
            optimizer_update!(params[key][l], grads[" $\nabla$ $key"][l], optimizer)
        end
    end
end

function train_step!(nn::NeuralNet, x::Array, y::Array, loss_fun::Function, ←
    optimizer::Optimizer=SGD())
    y^ = forward!(nn, x)
    loss = loss_fun(nn, y^, y)
    backward!(nn)
    optim_step!(nn, optimizer) # update params
    return loss
end
```

7.1.2 Zipser-Andersen モデル

Zipser-Andersen モデル (Zipser and Andersen, 1988) は頭頂葉の 7a 野のモデルであり、網膜座標系における物体の位置と眼球位置を入力として、頭部中心座標 (head centered coordinate) に変換する。隠れ層は PPC(Posterior parietal cortex) の細胞のモデルになっている。

データセットの生成

物体位置の表現には Gaussian 形式と monotonic 形式があるが、簡単のために、Gaussian 形式を用いる。なお、monotonic 形式については末尾の補足を参照してほしい。

```
# Gaussian 2d
function Gaussian2d(pos, sizex=8, sizey=8, σ=1)
    x, y = 0:sizex-1, 0:sizey-1
    X, Y = [i for i in x, j in 1:length(y)], [j for i in 1:length(x), j in y]
    x0, y0 = pos
    return exp.-((X .- x0) .^2 + (Y .- y0) .^2) / 2σ.^2
end
```

入力は 64(網膜座標系での位置)+2(眼球位置信号)=66 とする。眼球位置信号は原著では monotonic 形式による $32 (= 8 \text{ ユニット} \times 2(\text{x, y 方向}) \times 2 (\text{傾き正負}))$ ユニットで構成されるが、簡単のために眼球位置信号も x, y の 2 次元とする。視覚刺激は -40 度から 40 度までの範囲であり、10 度で離散化する。よって、網膜座標系での位置は 8×8 の行列で表現される。位置は 2 次元の Gaussian で表現する。ただし、 $1/e$ 幅 (ピークから $1/e$ に減弱する幅) は 15 度である。 $1/e$ の代わりに $1/2$ とすれば半値全幅 (FWHM) となる。スポットサイズを W 、Gaussian を $G(x)$ とすると、 $G(x + w/2) = G/e$ より、 $\sigma = \frac{\sqrt{2}w}{4}$ と求まる。

```
# dataset @eter
θmax = 40.0 # degree, θ ∈ [-θmax, θmax]
Δθ = 10.0 # degree
stimuli_size = Int(2θmax / Δθ)
w = 15.0 # degree; 1/e width
σ = √2w/(4Δθ);

# training @eter
n_data = 10000
n_traindata = Int(n_data*0.95)
n_batch = 100 # batch size
n_iter_per_epoch = Int(n_traindata/n_batch)
n_epoch = 2000; # number of epoch
```

```
# generate positions
Random.seed!(0)
retinal_pos = (rand(n_data, 2) .- 0.5) * 2θmax # ∈ [-40, 40]
head_centered_pos = (rand(n_data, 2) .- 0.5) * 2θmax # ∈ [-40, 40]
#retinal_pos = (rand(n_data, 2) .- 0.5) * 2θmax # ∈ [-40, 40]
#head_centered_pos = (rand(n_data, 2) .- 0.5) * 2θmax # ∈ [-40, 40]
eye_pos = head_centered_pos - retinal_pos; # ∈ [-80, 80]

# convert
input_retina = [hcat(Gaussian2d((retinal_pos[i, :] .+ θmax)/Δθ, stimuli_size, σ)... for i in 1:n_data)];
input_retina = vcat(input_retina...)
```

```
eye_pos /= 2θmax;

# concat
x_data = hcat(input_retina, eye_pos) #_encoded)
y_data = vcat([hcat(Gaussian2d((head_centered_pos[i, :] .+ θmax)/Δθ, stimuli_size, ~
    stimuli_size, σ)... ) for i in 1:n_data]...);

# split
x_traindata, y_traindata = x_data[1:n_traindata, :], y_data[1:n_traindata, :]
xtestdata, ytestdata = x_data[n_traindata+1:end, :], y_data[n_traindata+1:end, :];

product(sets...) = hcat([collect(x) for x in Iterators.product(sets...)])' # ←
    Array of Cartesian product of sets
```

モデルの定義を行う。

```
# model Θeter
n_in = stimuli_size^2 + 2 # number of inputs
n_hid = 16    # number of hidden units
n_out = stimuli_size^2   # number of outputs
η = 1e-2    # learning rate
losstype = "binary_crossentropy" # "squared_error"
```

```
nn = MLP([n_in, n_hid, n_out])#, bias=false)
optimizer = SGD(η=η);
loss_fun = binary_crossentropy!
#optimizer = Adam();
```

n_in, n_hid, n_out

学習を行う。

```
error_arr = zeros(n_epoch); # memory array of each epoch error

@showprogress "Training..." for e in 1:n_epoch
    for iter in 1:n_iter_per_epoch
        idx = (iter-1)*n_batch+1:iter*n_batch
        x, y = x_traindata[idx, :], y_traindata[idx, :]
        loss = train_step!(nn, x, y, loss_fun, optimizer)
        error_arr[e] += loss
    end
    error_arr[e] /= n_traindata
end
```

損失の変化を描画する。

```

figure(figsize=(4,2))
#semilogy(error_arr)
plot(error_arr)
ylabel("Error"); xlabel("Epoch"); xlim(0, n_epoch)
tight_layout()

```

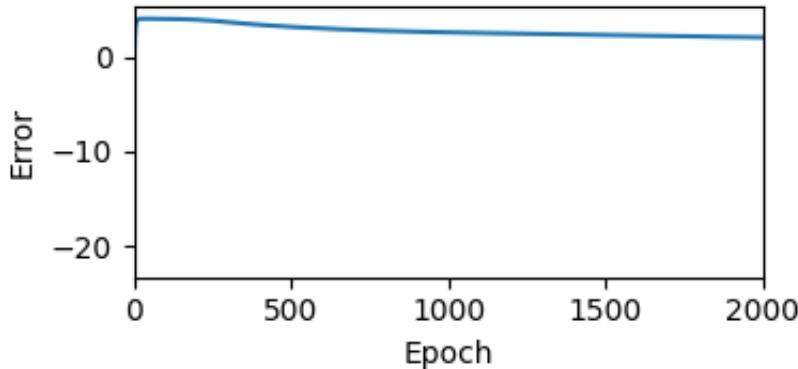


図 7.1 cell036.png

テストデータを用いて、出力を確認する。

```

x, y = xtestdata[1:2, :], ytestdata[1:2, :]
y^ = forward!(nn, x)

id = 1
figure(figsize=(6,2))
ax1 = subplot(1,3,1); title("input")
ax1.imshow(reshape(x[id, 1:64], (stimuli_size, stimuli_size))', ~
    interpolation="gaussian", extent=[-θmax, θmax, θmax, -θmax])
ax1.add_patch(plt.Circle((x[id, 65:66])*2θmax, radius=2, color="tab:red", fill=false))
xlabel("x"); ylabel("y");

ax2 = subplot(1,3,2); title("output")
ax2.imshow(reshape(y^[[id, :], (stimuli_size, stimuli_size))', ~
    interpolation="gaussian", extent=[-θmax, θmax, θmax, -θmax])
ax2.add_patch(plt.Circle((x[id, 65:66])*2θmax, radius=2, color="tab:red", fill=false))
xlabel("x");

ax3 = subplot(1,3,3); title("target")
ax3.imshow(reshape(y[[id, :], (stimuli_size, stimuli_size))', ~
    interpolation="gaussian", extent=[-θmax, θmax, θmax, -θmax])
ax3.add_patch(plt.Circle((x[id, 65:66])*2θmax, radius=2, color="tab:red", fill=false))
xlabel("x");

tight_layout()

```

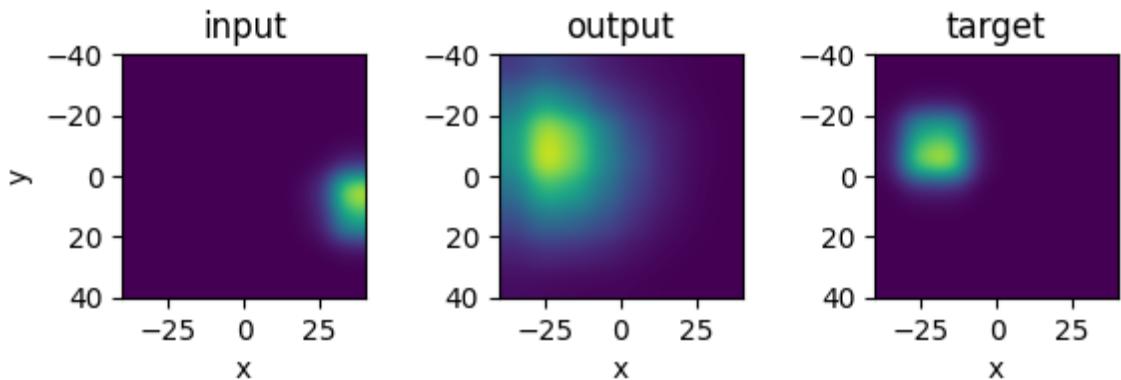


図 7.2 cell038.png

重み W_1 におけるゲインフィールドの描画を行う.

```
# Plot Gain fields
figure(figsize=(3.2, 3))
suptitle("Gain fields", fontsize=12)
subplots_adjust(hspace=0.1, wspace=0.1, top=0.925)
for i in 1:n_hid
    #subplot(3, 3, i)
    subplot(4, 4, i)
    imshow(reshape(nn.params["W"][1][1:stimuli_size^2, i], (stimuli_size, -
        stimuli_size)), cmap="hot")
    axis("off")
end
```

補足として Monotonic format による位置のエンコーディングに触れる. monotonic 形式を入力の眼球位置と出力の頭部中心座標で用いるという仮定には、視覚刺激を中心窓で捉えた際、得られる眼球位置信号を頭部中心座標での位置の教師信号として使用できるという利点がある. (Andersen and Mountcastle, 1983) では Parietal visual neurons (PVNs) の活動を調べ、傾き正あるいは負. 0 度をピークとして減少あるいは上昇の 4 種類あることを示した. 前者は一次関数 (と ReLU 関数) で記述可能である.

```
get_line(p1, p2) = [(p2[2]-p1[2])/(p2[1]-p1[1]), (p2[1]*p1[2] -
    p1[1]*p2[2])/((p2[1]-p1[1]))] # [slope, intercept]
eye_pos_encoding(x; linear_theta) = relu.(linear_theta[1, :] * x .+ linear_theta[2, :])

x = -20max:1:20max
slope_theta = hcat([get_line([80, 1], [-80, -2(i-1)/stimuli_size]) for i in 1:stimuli_size]...)
y = hcat(eye_pos_encoding.(x; linear_theta=slope_theta)...)

eye_pos_encoded = eye_pos_encoding(-10; linear_theta=slope_theta);
```

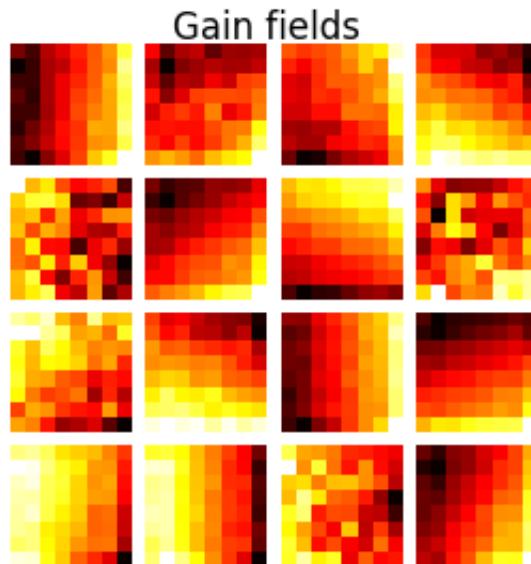


図 7.3 cell040.png

```
figure(figsize=(5,3))
subplot(2,1,1); plot(x, y'); xlabel("Eye position"); ylabel("Firing rate")
subplot(2,1,2); imshow(eye_pos_encoded[:, :']); title(L"Eye position $=-10^{\circ}\text{circ}$");
    xlabel("Units")
tight_layout()
```

7.2 BPTT (backpropagation through time)

通時的誤差逆伝播法 (backpropagation through time; BPTT). モデルの定義を行う.

```
using Base: @kwdef
using Parameters: @unpack # or using UnPack
using Random, ProgressMeter, PyPlot
```

```
f(x) = tanh(x)
df(x) = 1 - tanh(x)^2;
```

```
@kwdef struct RNNParameter{FT}
    dt::FT = 1 # time step (ms)
    τ::FT = 10 # time constant (ms)
    α::FT = dt / τ
```

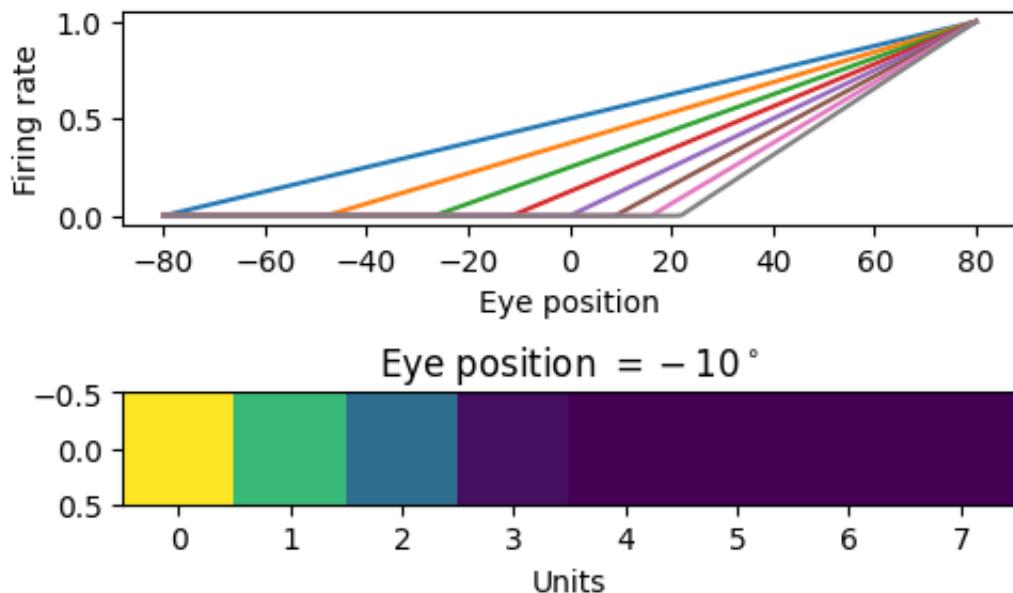


図 7.4 cell043.png

```

    η::FT = 1e-2 # learning rate
end

```

`w_in` は入力層から再帰層への重み, `w_rec` は再帰重み, `w_out` は出力重みである.

```

@kwdef mutable struct RNN{FT}
    param::RNNParameter = RNNParameter{FT}()

    n_batch::UInt32 # batch size
    n_in::UInt32 # number of input units
    n_rec::UInt32 # number of recurrent units
    n_out::UInt32 # number of output units

    h0::Array{FT} = zeros(n_batch, n_rec) # initial state of recurrent units

    # weights
    w_in::Array{FT} = 0.1*(rand(n_in, n_rec) .- 1)
    w_rec::Array{FT} = 1.5*randn(n_rec, n_rec)/sqrt(n_rec)
    w_out::Array{FT} = 0.1*(2*rand(n_rec, n_out) .- 1)/sqrt(n_rec)
    bias::Array{FT} = zeros(1, n_rec)

    # changes to weights
    dw_in::Array{FT} = zero(w_in)
    dw_rec::Array{FT} = zero(w_rec)
    dw_out::Array{FT} = zero(w_out)
    dbias::Array{FT} = zero(bias)

```

```
end
```

更新関数を定義する

```
function update!(variable::RNN, param::RNNParameter, x::Array, y::Array, ←
    training::Bool)
    @unpack n_batch, n_in, n_rec, n_out, h0, w_in, w_rec, w_out, bias, dw_in, ←
        dw_rec, dw_out, dbias = variable
    @unpack dt, τ, α, η = param

    t_max = size(x)[2] # number of timesteps
    u, h = zeros(n_batch, t_max, n_rec), zeros(n_batch, t_max, n_rec) # input ←
        (feedforward + recurrent), time-dependent RNN activity vector
    h[:, 1, :] = h0 # initial state

    y^ = zeros(n_batch, t_max, n_out) # RNN output
    error = zeros(n_batch, t_max, n_out) # readout error

    for t in 1:t_max-1
        u[:, t+1, :] = h[:, t, :] * w_rec + x[:, t+1, :] * w_in .+ bias
        h[:, t+1, :] = h[:, t, :] + α * (-h[:, t, :] + f.(u[:, t+1, :]))
        y^[:, t+1, :] = h[:, t+1, :] * w_out
        error[:, t+1, :] = y[:, t+1, :] - y^[:, t+1, :] # readout error
    end

    # backward
    if training
        z = zero(h)
        z[:, end, :] = error[:, end, :] * w_out'

        for t in t_max:-1:1
            zu = z[:, t, :] .* df.(u[:, t, :])
            if t ≥ 2
                z[:, t-1, :] = z[:, t, :] * (1 - α) + error[:, t, :] * w_out' + zu * ←
                    w_rec * α
                dw_rec[:, :] += h[:, t-1, :]' * zu
            end

            # Updates Δweights:
            dw_out[:, :] += h[:, t, :]' * error[:, t, :]
            dw_in[:, :] += x[:, t, :]' * zu
            dbias[:, :] .+= sum(zu)
        end

        # update weights
        w_out[:, :] += η / t_max * dw_out
        w_rec[:, :] += η / t_max * α * dw_rec
        w_in[:, :] += η / t_max * α * dw_in
        bias[:, :] += η / t_max * α * dbias

        # reset
        dw_in[:, :] = zero(w_in)
        dw_rec[:, :] = zero(w_rec)
    end
end
```

```

dw_out[:, :] = zero(w_out)
dbias[:, :] = zero(bias)
end

return error, y^, h
end

```

例として正弦波を出力する RNN を考える。入力 1, 中間 64, 出力 2 の RNN である。

```

nt = 100 # number of timesteps in one period
n_batch = 1 # batch size
n_in = 1 # number of inputs
n_out = 2 # number of outputs

begin_input = 0 # begin time steps of input
end_input = 30 # end time steps of input

tsteps = 0:nt-1 # array of time steps
x = ones(n_batch) * (begin_input .≤ tsteps .≤ end_input)' # input array

y = zeros(n_batch, nt, n_out) # target array
y[:, begin_input+1:end, 1] = sin.(tsteps[1:end-begin_input]*0.1)
y[:, begin_input+1:end, 2] = sin.(tsteps[1:end-begin_input]*0.2)

n_epoch = 25000 # number of epoch

# memory array of each epoch error
error_arr = zeros(Float32, n_epoch);

```

入力と訓練データの確認をする。

```

figure(figsize=(5, 4))
subplot(2,1,1); plot(x[1, :]); ylabel("x")
subplot(2,1,2); plot(tsteps, y[1, :, 1], label="Target1"); plot(tsteps, y[1, :, 2], -
    label="Target2")
xlabel("Time steps"); ylabel("y"); xlim(0, tsteps[end])
legend(loc="upper right", fontsize=8)
tight_layout()

```

モデルの定義をする。

```
rnn = RNN{Float32}(n_batch=n_batch, n_in=n_in, n_rec=32, n_out=n_out);
```

学習を実行する。

```

@showprogress "Training..." for e in 1:n_epoch
    error, y^, h = update!(rnn, rnn.param, x, y, true)
    error_arr[e] = sum(error.^ 2)
end

```

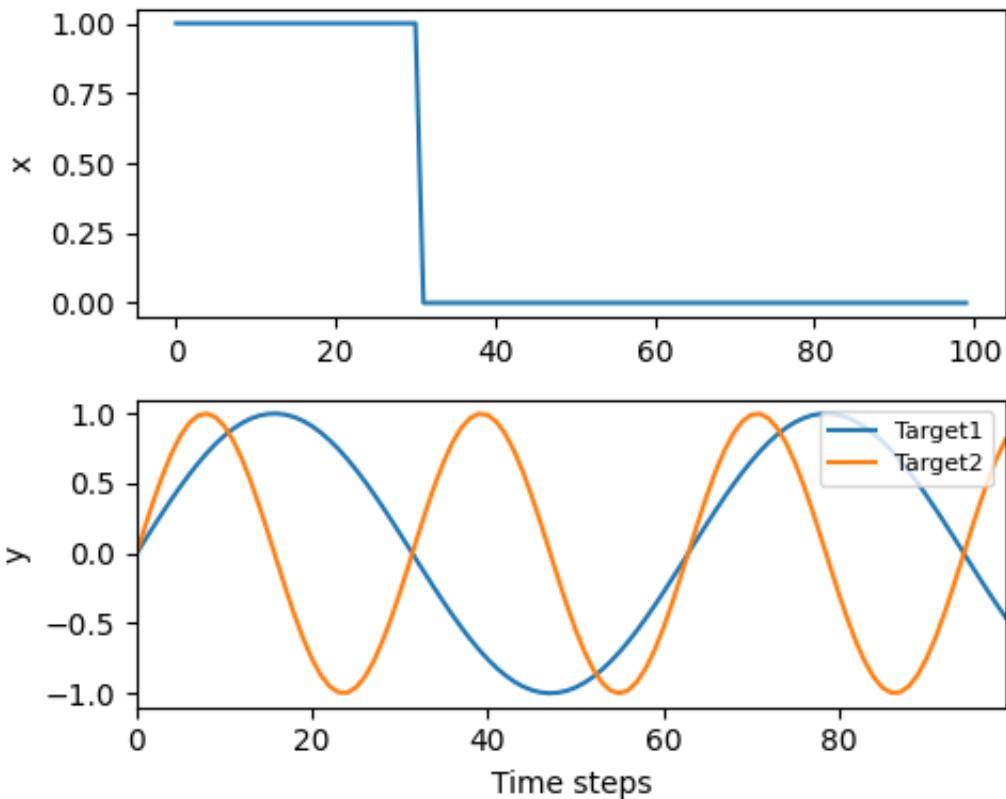


図 7.5 cell011.png

損失の推移を確認する。

```
figure(figsize=(4,3))
semilogy(error_arr); ylabel("Error"); xlabel("Epoch");
xlim(0, n_epoch)
tight_layout()
```

学習後の出力の確認を行う。

```
error, y^, h = update!(rnn, rnn.param, x, y, false)
println("Error : ", sum(error.^2))
```

見やすいように出力のピークに応じて中間層のユニットをソートする。

```
max_idx = Tuple.(argmax(h[1, :, :], dims=2))
h_ = h[1, :, sortperm(last.(max_idx)[:, 1])];
```

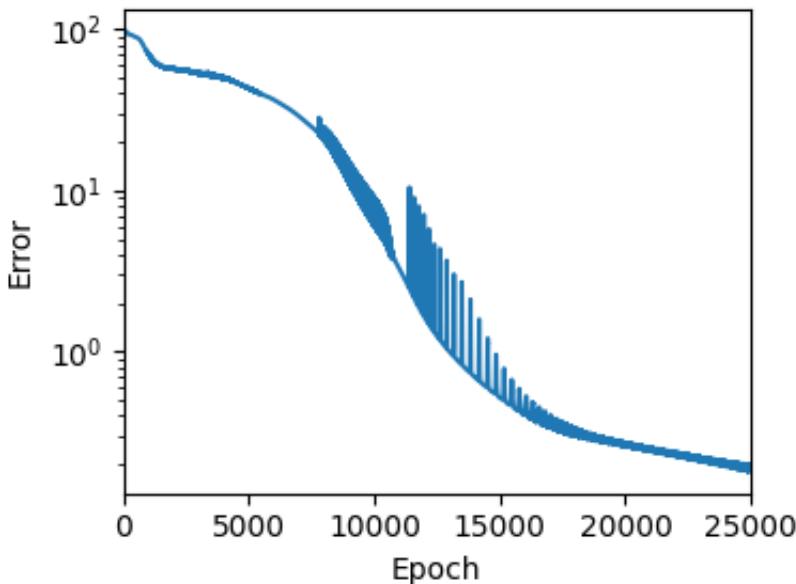


図 7.6 cell017.png

出力層、中間層の出力を描画する。

```
figure(figsize=(5, 4))
subplot(2,1,1)
plot(tsteps, y[1, :, 1], "--k", alpha=.5, label="Target1")
plot(tsteps, y[1, :, 2], "-.k", alpha=.5, label="Target2")
plot(tsteps, y^*[1, :, 1], label="Output1")
plot(tsteps, y^*[1, :, 2], label="Output2")
ylabel("y"); xlim(0, tsteps[end])
legend(loc="upper right", ncol=2, fontsize=8)

subplot(2,1,2)
imshow(h_, cmap="turbo", aspect=0.85)
xlabel("Time steps"); ylabel("# hidden units")

tight_layout()
```

参考文献

Andersen, R. A. and Mountcastle, V. B. (1983). "The influence of the angle of gaze upon the excitability of the light-sensitive neurons of the posterior parietal cortex". *J. Neurosci.* 3.3, pp. 532–548.

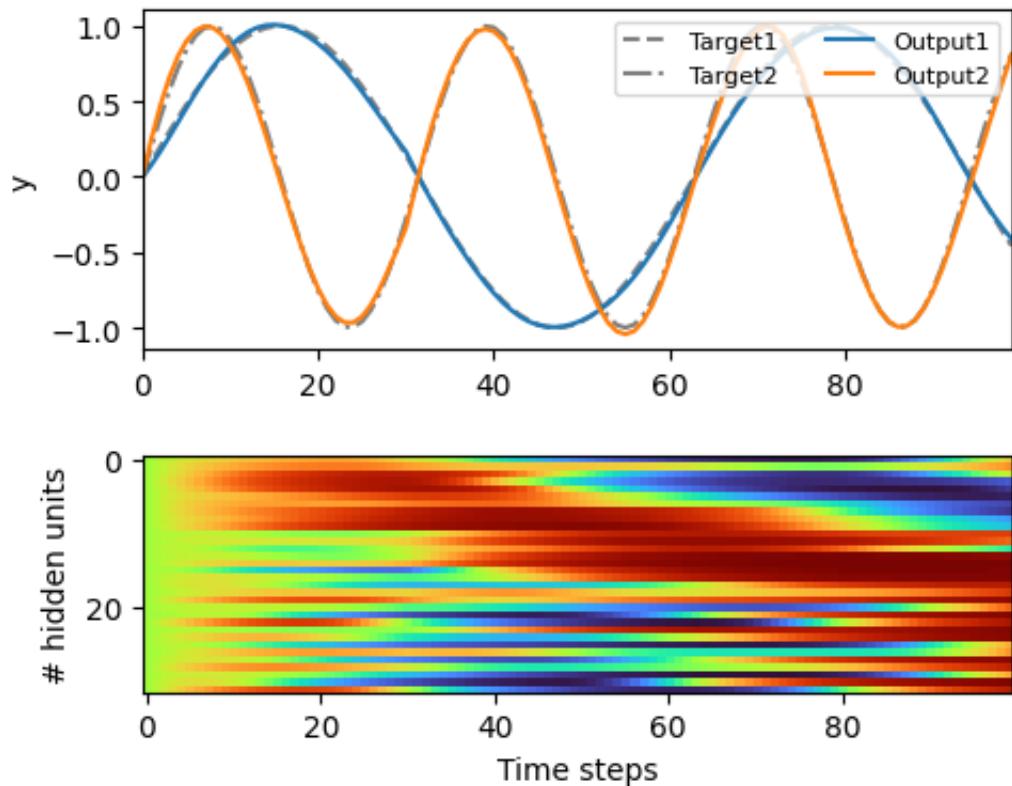


図 7.7 cell023.png

Glorot, X. and Bengio, Y. (2010). “Understanding the difficulty of training deep feedforward neural networks”. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Y. W. Teh and M. Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256.

Kingma, D. P. and Ba, J. (2014). “Adam: A method for stochastic optimization”. *arXiv preprint arXiv:1412.6980*.

Zipser, D. and Andersen, R. A. (1988). “A back-propagation programmed network that simulates response properties of a subset of posterior parietal neurons”. *Nature* 331.6158, pp. 679–684.

第8章

運動制御

8.1 躍度最小モデル

躍度最小モデル (minimum-jerk model; (Flash and Hogan, 1985)) を実装する。解析的に求まるが以下では二次計画法を用いて数値的に求める。

8.1.1 等式制約下の二次計画法 (Equality Constrained Quadratic Programming)

n 個の変数があり、 m 個の制約条件がある等式制約二次計画問題を考える。 $\mathbf{x} \in \mathbb{R}^n$, 対称行列 $\mathbf{P} \in \mathbb{R}^{n \times n}$, $\mathbf{q} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$. このとき、問題は次のようにになる。

$$\text{Minimize } \frac{1}{2} \mathbf{x}^\top \mathbf{P} \mathbf{x} + \mathbf{q}^\top \mathbf{x} \quad (8.1)$$

$$\text{subject to } \mathbf{A} \mathbf{x} = \mathbf{b} \quad (8.2)$$

Lagrange の未定乗数法を用いると解は

$$\begin{bmatrix} \mathbf{P} & \mathbf{A}^\top \\ \mathbf{A} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{q} \\ \mathbf{b} \end{bmatrix} \quad (8.3)$$

の解として与えられる。ここで $\lambda \in \mathbb{R}^m$ は Lagrange 乗数のベクトルである。

```
using LinearAlgebra, Random, ToeplitzMatrices, PyPlot
rc("axes.spines", top=false, right=false)
```

```
# Equality Constrained Quadratic Programming
function solve_quad_prog(P, q, A, b)
    """
    minimize : 1/2 * x'*P*x + q'*x
    subject to : A*x = b
    """
    K = [P A'; A zeros(size(A)[1], size(A)[1])] # KKT matrix
```

```

sol = K \ [-q; b]
return sol[1:size(A)[2]]
end

```

ちなみに Julia では $Ax = b$ の解を出すとき、 $x = A^{-1} * b$ よりも $x = A \backslash b$ とした方がよい。

```

P = diagm([1.0, 0.0])
q = [3.0, 4.0];
A = [1.0, 1.0];
b = [1.0]
x = solve_quad_prog(P, q, A, b);

```

8.1.2 躍度最小モデルの実装

1 次元における運動を考えよう。この仮定ではサッカードするときの眼球運動などが当てはまる。以下では (Yazdani et al., 2012) での問題設定を用いる。Toeplitz 行列を用いた実装は Yazdani らの Python で cvxopt を用いた実装を参考にして作成した。問題設定は以下のようとする。

$$\underset{u(t)}{\text{minimize}} \quad \|u(t)\|_2 \quad (8.4)$$

$$\text{subject to} \quad \dot{x}(t) = Ax(t) + Bu(t) \quad (8.5)$$

ただし、 $\|\cdot\|_2$ は L_2 ノルムを意味し、 $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$, $\mathbf{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix}$, $u(t) = \ddot{x}(t)$ とする。すなわち、制御信号 $u(t)$ は躍度 $\ddot{x}(t)$ と等しいとする。

```

T = 1.0 # simulation time (sec)
dt = 1e-2 # time step (sec)
nt = Int(T/dt) # number of samples
trange = range(0, 1, length=nt); # range of time

```

```

row_jerk = [[-1, 3, -3, 1]; zeros(nt-4)]
col_jerk = [-1; zeros(nt-4)];
D_jerk = Toeplitz(col_jerk, row_jerk);
# = diagm(0 => -ones(nt-3), 1 => 3*ones(nt-3), 2=>-3*ones(nt-3), «
# => ones(nt-3))[1:end-3, :]

```

実際には D_{jerk} には $(1/dt)^3$ を乗じるべきであるが、二次計画法の数値的な安定性のために結果の描画の際にのみ乗じる。

```

init_pos = [1; zeros(nt-1)]'
final_pos = [zeros(nt-1); 1]'

```

```

init_vel = [[-1, 1]; zeros(nt-2)]'
final_vel = [zeros(nt-2); [-1, 1]]'
init_accel = [[1, -2, 1]; zeros(nt-3)]'
final_accel = [zeros(nt-3); [1, -2, 1]]';

Aeq = [init_pos; final_pos; init_vel; final_vel; init_accel; final_accel];

beq = zeros(6) # (init or final) or (pos, vel, acc) = 2*3
beq[1] = 0      # initial position (m)
beq[2] = 2;     # final position (m)

```

二次計画法を解く.

```
sol_pos = solve_quad_prog(D_jerk' * D_jerk, zeros(nt), Aeq, beq);
```

位置解を速度, 加速度, 躍度に変換する.

```

# set D_vel and D_accel
row_vel = [[-1, 1]; zeros(nt-2)]
col_vel = [-1; zeros(nt-2)]
D_vel = (1/dt) * Toeplitz(col_vel, row_vel);

row_accel = [[1,-2,1]; zeros(nt-3)]
col_accel = [1; zeros(nt-3)]
D_accel = (1/dt)^2 * Toeplitz(col_accel, row_accel);

# compute solution of vel, accel and jerk
sol_vel = D_vel * sol_pos;
sol_accel = D_accel * sol_pos;
sol_jerk = (1/dt)^3 * D_jerk * sol_pos;

```

結果を描画する.

```

figure(figsize=(8, 4))
subplot(2,2,1)
plot(trange, sol_pos)
ylabel(L"Position ($m$)"); grid()

subplot(2,2,2)
plot(trange[1:nt-1], sol_vel)
ylabel(L"Velocity ($m/s$)"); grid()

subplot(2,2,3)
plot(trange[1:nt-2], sol_accel)
ylabel(L"Acceleration ($m/s^2$)"); xlabel("Time (s)"); grid()

subplot(2,2,4)
plot(trange[1:nt-3], sol_jerk)
ylabel(L"Jerk ($m/s^3$)"); xlabel("Time (s)"); grid()

tight_layout()

```

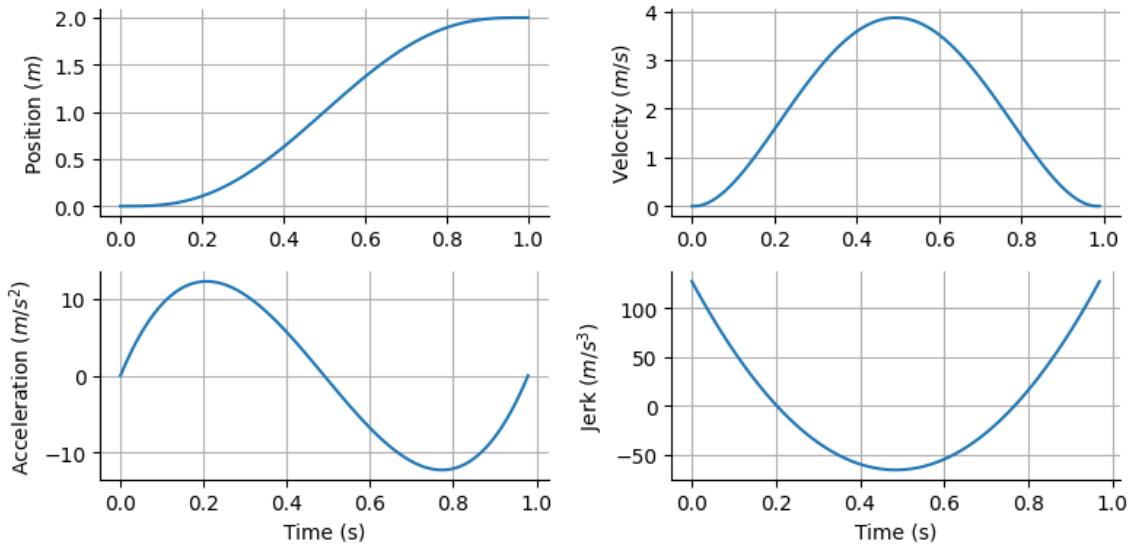


図 8.1 cell015.png

8.1.3 経由点を通る場合

経由点問題 (via-point problem) を考える。

```

via_point_pos = zeros(nt)'
via_point_pos[Int(nt/2)] = 1; # via point timing

Aeq2 = [init_pos; final_pos; via_point_pos; init_vel; final_vel; init_accel; -
        final_accel];

beq2 = zeros(7) # (init or final) or (pos, vel, acc) + via_point_pos = 2*3 + 1 = 7
beq2[1] = 2      # initial position (m)
beq2[2] = 4      # final position (m)
beq2[3] = 6;     # via point position (m)

```

```

sol2_pos = solve_quad_prog(D_jerk' * D_jerk, zeros(nt), Aeq2, beq2);
sol2_vel = D_vel * sol2_pos;
sol2_accel = D_accel * sol2_pos;
sol2_jerk = (1/dt)^3 * D_jerk * sol2_pos;

```

```

figure(figsize=(8, 4))
subplot(2,2,1)
plot(trange, sol2_pos)
ylabel(L"Position ($m$)");
grid()

```

```

subplot(2,2,2)
plot(trange[1:nt-1], sol2_vel)
ylabel(L"Velocity ($m/s$)"); grid()

subplot(2,2,3)
plot(trange[1:nt-2], sol2_accel)
ylabel(L"Acceleration ($m/s^2$)"); xlabel("Time (s)"); grid()

subplot(2,2,4)
plot(trange[1:nt-3], sol2_jerk)
ylabel(L"Jerk ($m/s^3$)"); xlabel("Time (s)"); grid()

tight_layout()

```

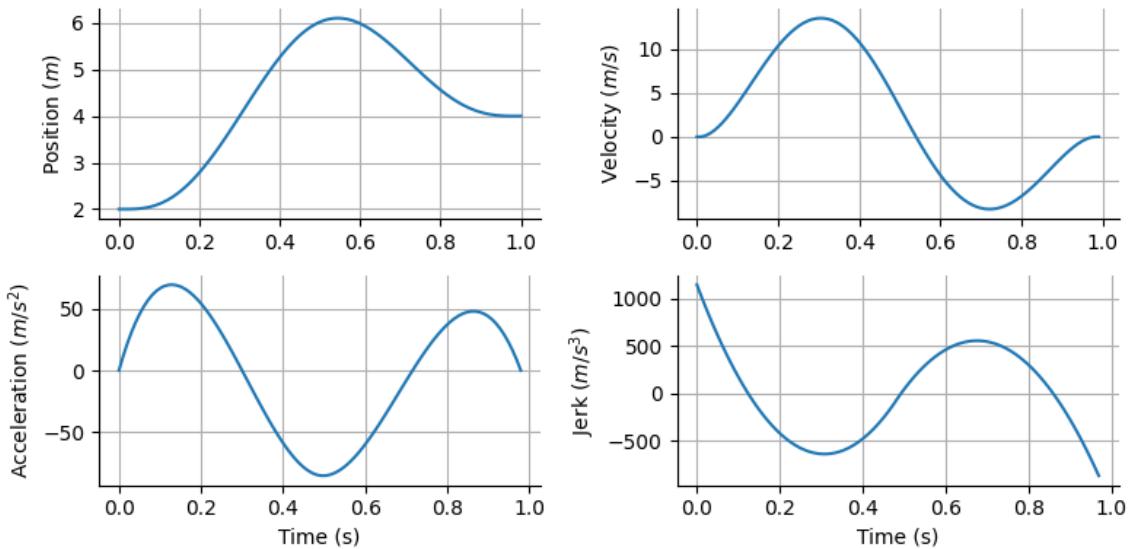


図 8.2 cell019.png

8.2 終点誤差分散最小モデル

終点誤差分散最小モデル (minimum-variance model; (Harris and Wolpert, 1998)) を実装する.
 $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{B} \in \mathbb{R}^n$ とする. $\dot{\mathbf{x}} = \mathbf{A}_c \mathbf{x} + \mathbf{B}_c(u + w)$ について, 差分化すると

$$\mathbf{x}(t + dt) = \mathbf{x}(t) + \dot{\mathbf{x}}dt \quad (8.6)$$

$$\mathbf{x}_{t+1} = \mathbf{I}\mathbf{x}_t + (\mathbf{A}_c dt)\mathbf{x}_t + (\mathbf{B}_c dt)(u + w) \quad (8.7)$$

となる。ここで \mathbf{I} は単位行列である。よって $\mathbf{A} = \mathbf{I} + \mathbf{A}_c dt$, $\mathbf{B} = \mathbf{B}_c dt$ として

$$\mathbf{x}_{t+1} = \mathbf{Ax}_t + \mathbf{B}(u_t + w_t) \quad (8.8)$$

と表せる。 \mathbf{x}_t の平均は

$$\mathbb{E}[\mathbf{x}_t] = \mathbf{A}^t \mathbf{x}_0 + \sum_{i=0}^{t-1} \mathbf{A}^{t-1-i} \mathbf{B} u_i \quad (8.9)$$

\mathbf{x}_t の分散は

$$\text{Cov}[\mathbf{x}_t] = k \sum_{i=0}^{t-1} (\mathbf{A}^{t-1-i} \mathbf{B}) (\mathbf{A}^{t-1-i} \mathbf{B})^\top u_i^2 \quad (8.10)$$

となる。

8.2.1 終点誤差分散最小モデルの実装

以下では田中先生の <https://www.motorcontrol.jp/archives/?MC13> のコードを参考に作成した。

```
using LinearAlgebra, Random, PyPlot
rc("axes.spines", top=false, right=false)

# Equality Constrained Quadratic Programming
function solve_quad_prog(P, q, A, b)
    """
    minimize : 1/2 * x'*P*x + q'*x
    subject to : A*x = b
    """
    K = [P A'; A zeros(size(A)[1], size(A)[1])] # KKT matrix
    sol = K \ [-q; b]
    return sol[1:size(A)[2]]
end;

function minimum_variance_model(Ac, Bc, x0, xf, tf, tp, dt)
    dims = size(x0)[1]
    ntf = round(Int, tf/dt)
    ntp = round(Int, tp/dt)
    nt = ntf + ntp # total time steps

    A = I(dims) + Ac * dt
    B = Bc*dt
    #A = exp(Ac*dt);
    #B = Ac^-1 * (I(dims) - A) * Bc;

    # calculation of V
    diagV = zeros(nt);
    for t=0:nt-1
```

```

if t < ntf
    diagV[t+1] = sum([(A^(k-t-1) * B * B' * A'^(k-t-1))[1,1] for k=ntf:nt-1])
else
    diagV[t+1] = diagV[t] + (A^(nt-t-2) * B * B' * A'^(nt-t-2))[1,1]
end
end
diagV /= maximum(diagV) # for numerical stability
V = Diagonal(diagV);

# 制約条件における行列Cとベクトルdの計算
#calculation of C
C = zeros(dims*(ntp+1), nt);
for p=1:ntp+1
    for q=1:nt
        if ntf-1+(p-1)-(q-1) >= 0
            idx = dims*(p-1)+1:dims*p
            C[idx, q] = A^(ntf-1-(q-1)+(p-1)) * B # if ntf-1-(q-1)+(p-1) == 0; ←
                A^(ntf-1-(q-1)+(p-1))*B equal to B
        end
    end
end

# calculation of d
d = vcat([xf - A^(ntf+t) * x0 for t=0:ntp]...);

# 制御信号を二次計画法で計算 (solution by quadratic programming)
u = solve_quad_prog(V, zeros(nt), C, d);

# 制御信号を二次計画法で計算 (forward solution)
x = zeros(dims, nt);
x[:,1] = x0;
Σ = zeros(dims, dims, nt);
Σ[:, :, 1] = B * u[1]^2 * B'
for t=1:nt-1
    x[:,t+1] = A*x[:, t] + B*u[t] # update
    Σ[:, :, t+1] = A * Σ[:, :, t] * A' + B * u[t]^2 * B' # variance
end
return x, u, Σ
end

```

```

t1 = 224*1e-3 # time const of eye dynamics (s)
t2 = 13*1e-3 # another time const of eye dynamics (s)
tm = 10*1e-3
dt = 1e-3 # simulation time step (s)
tf = 50*1e-3 # movement duration (s)
tp = 40*1e-3 # post-movement duration (s)
nt = round(Int, (tf+tp)/dt) # total time steps
trange = (1:nt) * dt * 1e3 # ms

# 2nd order
x0z = zeros(2) # initial state (pos=0, vel=0)
xfz = [10, 0] # final state (pos=10, vel=0)
Acz = [0 1; -1/(t1*t2) -1/t1-1/t2];
Bcz = [0, 1]

```

```
# 3rd order
x0_3 = zeros(3) # initial state (pos=0, vel=0, acc=0)
xf_3 = [10, 0, 0] # final state (pos=10, vel=0, acc=0)
Ac_3 = [0 1 0; 0 0 1; -1/(t1*t2*tm) -1/(t1*t2)-1/(t1*tm)-1/(t2*tm) -1/t1-1/t2-1/tm];
Bc_3 = [0, 0, 1/tm];

x2, u2, Σ2 = minimum_variance_model(Ac_2, Bc_2, x0_2, xf_2, tf, tp, dt);
x3, u3, Σ3 = minimum_variance_model(Ac_3, Bc_3, x0_3, xf_3, tf, tp, dt);
```

結果の描画.

```
figure(figsize=(6, 4))
subplot(2,2,1); plot(trange, x2[1, :], label="2nd order"); plot(trange, x3[1, :], '--',
label="3rd order");
ylabel("Eye position (deg)"); grid(); legend()
subplot(2,2,2); plot(trange, x2[2, :]); plot(trange, x3[2, :], '--');
ylabel("Eye velocity (deg/s)"); grid();
subplot(2,2,3); plot(trange, u2); plot(trange, u3, '--');
ylabel("Control signal"); xlabel("Time (ms)"); grid();
ax = gca(); ax[:ticklabel_format](style="sci", axis="y", scilimits=(0,0))
subplot(2,2,4); plot(trange, Σ2[1,1,:]); plot(trange, Σ3[1,1,:], '--');
ylabel("Positional Variance"); xlabel("Time (ms)"); grid()
tight_layout()
```

8.3 最適フィードバック制御モデル

ToDo: infiniteOFC と数式の統一を行う.

8.3.1 最適フィードバック制御モデルの構造

最適フィードバック制御モデル (optimal feedback control; OFC) の特徴として目標軌道を必要としないことが挙げられる. Kalman フィルタによる状態推定と線形 2 次レギュレーター (LQR: linear-quadratic regurator) により推定された状態に基づいて運動指令を生成という 2 つの流れが基本となる.

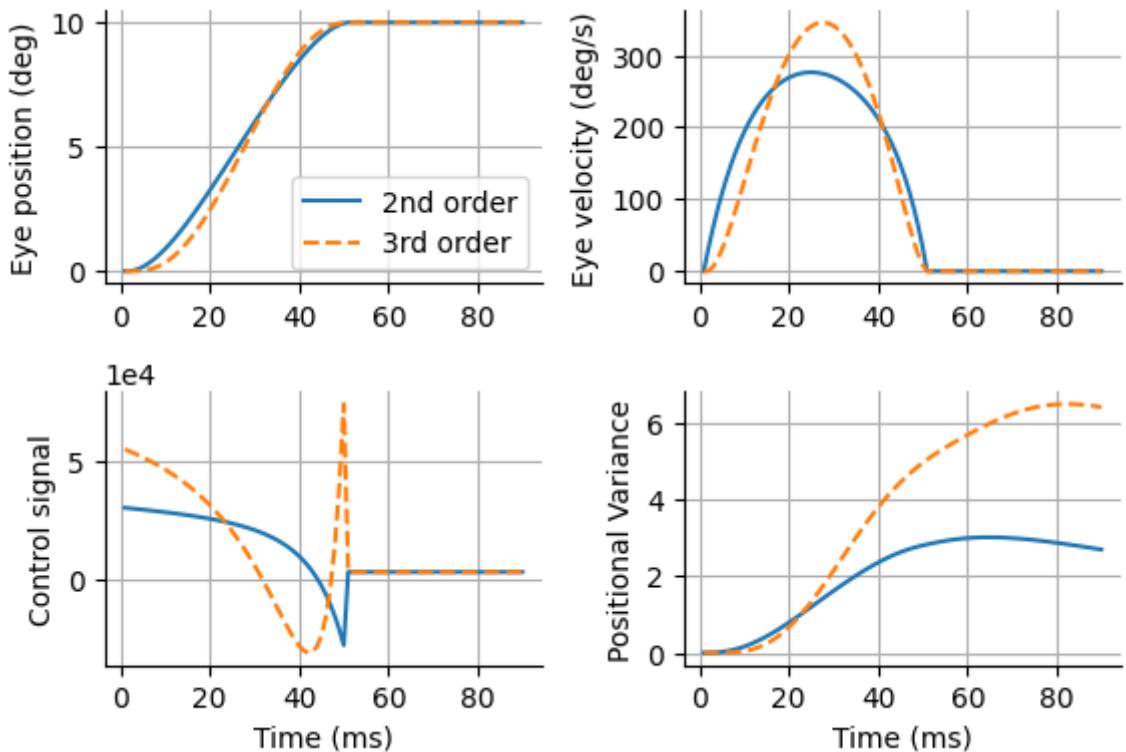


図 8.3 cell008.png

系の状態変化

$$\text{Dynamics} \quad \mathbf{x}_{t+1} = A\mathbf{x}_t + B\mathbf{u}_t + \boldsymbol{\xi}_t + \sum_{i=1}^c \varepsilon_t^i C_i \mathbf{u}_t \quad (8.11)$$

$$\text{Feedback} \quad \mathbf{y}_t = H\mathbf{x}_t + \omega_t + \sum_{i=1}^d \epsilon_t^i D_i \mathbf{x}_t \quad (8.12)$$

$$\text{Cost per step} \quad \mathbf{x}_t^\top Q_t \mathbf{x}_t + \mathbf{u}_t^\top R \mathbf{u}_t \quad (8.13)$$

LQG

加法ノイズしかない場合 ($C = D = 0$), 制御問題は線形 2 次ガウシアン (LQG: linear-quadratic-Gaussian) 制御と呼ばれる.

運動制御 (Linear-Quadratic Regulator)

$$\mathbf{u}_t = -L_t \hat{\mathbf{x}}_t \quad (8.14)$$

$$L_t = (R + B^\top S_{t+1} B)^{-1} B^\top S_{t+1} A \quad (8.15)$$

$$S_t = Q_t + A^\top S_{t+1} (A - BL_t) \quad (8.16)$$

$$s_t = \text{tr}(S_{t+1} \Omega^\xi) + s_{t+1}; s_T = 0 \quad (8.17)$$

$$S_T = Q$$

状態推定 (Kalman Filter)

$$\hat{\mathbf{x}}_{t+1} = A\hat{\mathbf{x}}_t + B\mathbf{u}_t + K_t (\mathbf{y}_t - H\hat{\mathbf{x}}_t) + \eta_t \quad (8.18)$$

$$K_t = A\Sigma_t H^\top (H\Sigma_t H^\top + \Omega^\omega)^{-1} \quad (8.19)$$

$$\Sigma_{t+1} = \Omega^\xi + (A - K_t H) \Sigma_t A^\top \quad (8.20)$$

この場合に限り、運動制御と状態推定を独立させることができる。

一般化 LQG

状態および制御依存ノイズがある場合、

8.3.2 実装

ライブラリの読み込みと関数の定義。

```
using Base: @kwdef
using Parameters: @unpack
using LinearAlgebra, Kronecker, Random, BlockDiagonals, PyPlot
rc("axes.spines", top=false, right=false)
rc("font", family="Arial")
```

ToDo: struct 修正 (n が両方に入っている)

```
@kwdef struct Reaching1DModelParameter
    n = 4 # number of dims
    p = 3 #
    i = 0.25 # kgm^2,
    b = 0.2 # kgm^2/s
    ta = 0.03 # s
    te = 0.04 # s
    L0 = 0.35 # m

    bu = 1 / (ta * te * i)
    α1 = bu * b
    α2 = 1/(ta * te) + (1/ta + 1/te) * b/i
```

```

 $\alpha_3 = b/i + 1/ta + 1/te$ 

A = [zeros(p) I(p); -[0, α1, α2, α3]']
B = [zeros(p); bu]
C = [I(p) zeros(p)]
D = Diagonal([1e-3, 1e-2, 5e-2])

Y = 0.02 * B
G = 0.03 * I(n)
end

@kwdef struct Reaching1DModelCostParameter
    n = 4
    dt = 1e-2 # sec
    T = 0.5 # sec
    nt = round(Int, T/dt) # num time steps
    Q = [zeros(nt-1, n, n); reshape(Diagonal([1.0, 0.1, 1e-3, 1e-4]), (1, n, n))]
    R = 1e-4 / nt

    init_pos = -0.5
    x1 = [init_pos; zeros(n-1)]#zeros(n)
    Σ1 = zeros(n, n)
end

```

Q の値は各時刻において一般座標（位置，速度，加速度，躍度）のそれぞれを 0 にするコストに対する重みづけである。例えば、速度も 0 にすることを重視すれば 2 番目の係数を上げる。 S と Σ は各時点での値を一時的にしか必要としないので更新する。

```

function LQG(param::Reaching1DModelParameter, -
            cost_param::Reaching1DModelCostParameter; discrete=true)
    @unpack n, p, A, B, C, D, G = param
    @unpack Q, R, x1, Σ1, dt, nt = cost_param

    if discrete
        A = I + A * dt
        B = B * dt
        C = C * dt
        D = sqrt(dt) * D
        G = sqrt(dt) * G
    end

    L = zeros(nt-1, n) # Feedback gains
    K = zeros(nt-1, n, p) # Kalman gains
    S = copy(Q[end, :, :]) # S_T = Q
    Σ = copy(Σ1);

    for t in 1:nt-1
        K[t, :, :] = A * Σ * C' / (C * Σ * C' + D) # update K
        Σ = G + (A - K[t, :, :] * C) * Σ * A' # update Σ
    end

    cost = 0
    for t in nt-1:-1:1

```

```

cost += tr(S * G)
L[t, :] = (R + B' * S * B) \ B' * S * A # update L
S = Q[t, :, :] + A' * S * (A - B * L[t, :]')      # update S
end

# adjust cost
cost += x_1' * S * x_1
return L, K, cost
end

```

シミュレーション

信号依存ノイズ Y が入っている場合は LQG とは異なってくる。

$$\mathbf{u}_t = -L_t \hat{\mathbf{x}}_t \quad (8.21)$$

$$L_t = \left(B^\top S_{t+1}^x B + R + \sum_n C_n^\top (S_{t+1}^x + S_{t+1}^e) C_n \right)^{-1} B^\top S_{t+1}^x A \quad (8.22)$$

$$S_t^x = Q_t + A^\top S_{t+1}^x (A - BL_t); \quad S_T^x = Q_T \quad (8.23)$$

$$S_t^e = A^\top S_{t+1}^x BL_t + (A - K_t H)^\top S_{t+1}^e (A - K_t H); \quad S_T^e = 0 \quad (8.24)$$

$$s_t = \text{tr} (S_{t+1}^x \Omega^\xi + S_{t+1}^e (\Omega^\xi + \Omega^\eta + K_t \Omega^\omega K_t^\top)) + s_{t+1}; \quad s_n = 0. \quad (8.25)$$

$$\hat{\mathbf{x}}_{t+1} = A\hat{\mathbf{x}}_t + B\mathbf{u}_t + K_t (y_t - H\hat{\mathbf{x}}_t) \quad (8.26)$$

$$K_t = A\Sigma_t^e H^\top (H\Sigma_t^e H^\top + \Omega^\omega)^{-1} \quad (8.27)$$

$$\Sigma_{t+1}^e = (A - K_t H) \Sigma_t^e A^\top + \sum_n C_n L_t \Sigma_t^x L_t^\top C_n^\top; \quad \Sigma_1^e = \Sigma_1 \quad (8.28)$$

$$\Sigma_{t+1}^x = K_t H \Sigma_t^e A^\top + (A - BL_t) \Sigma_t^x (A - BL_t)^\top; \quad \Sigma_1^x = \hat{\mathbf{x}}_1 \hat{\mathbf{x}}_1^\top \quad (8.29)$$

```

function gLQG(param::Reaching1DModelParameter, ~
    cost_param::Reaching1DModelCostParameter, maxiter=200, ε=1e-8)
    @unpack n, p, A, B, C, D, Y, G = param
    @unpack Q, R, x_1, Σ_1, dt, nt = cost_param

    A = I + A * dt
    B = B * dt
    C = C * dt
    D = sqrt(dt) * D
    G = sqrt(dt) * G
    Y = sqrt(dt) * Y

    L = zeros(nt-1, n) # Feedback gains
    K = zeros(nt-1, n, p) # Kalman gains

    cost = zeros(maxiter)
    for i in 1:maxiter
        S_x = copy(Q[end, :, :])
    end

```

```

Se = zeros(n, n)
Σx^ = x1 * x1' # \Sigma TAB \^x TAB \hat TAB
Σe = copy(Σ1)

for t in 1:nt-1
    K[t, :, :] = A * Σe * C' / (C * Σe * C' + D)

    AmBL = A - B * L[t, :]
    LΣx^L = L[t, :]' * Σx^ * L[t, :]

    Σx^ = K[t, :, :] * C * Σe * A' + AmBL * Σx^ * AmBL'
    Σe = G + (A - K[t, :, :] * C) * Σe * A' + Y * LΣx^L * Y'
end

for t in nt-1:-1:1
    cost[i] += tr(Sx * G + Se * (G + K[t, :, :] * D * K[t, :, :]'))
    L[t, :] = (R + B' * Sx * B + Y' * (Sx + Se) * Y) \ B' * Sx * A

    AmKC = A - K[t, :, :] * C
    Se = A' * Sx * B * L[t, :]' + AmKC' * Se * AmKC
    Sx = Q[t, :, :] + A' * Sx * (A - B * L[t, :]')
end

# adjust cost
cost[i] += x1' * Sx * x1 + tr((Sx + Se) * Σ1)
if i > 1 && abs(cost[i] - cost[i-1]) < ε
    cost = cost[1:i]
    break
end
end
return L, K, cost
end

```

状態ノイズがある場合に関しては Todorov の MATLAB コード <https://homes.cs.washington.edu/~todorov/software/gLQG.zip> を参照。位置は目標位置を基準とする座標で表現し、位置が 0 になるように運動を行う。状態の中に標的の位置を含めコストパラメータを修正することで初期位置を基準とする座標系での運動を記述できる。モデルに関しては Todorov2005 を参照。

```

function simulation(param::Reaching1DModelParameter, -
    cost_param::Reaching1DModelCostParameter,
    L, K; noisy=false)
@unpack n, p, A, B, C, D, Y, G = param
@unpack Q, R, x1, dt, nt = cost_param

X = zeros(n, nt)
u = zeros(nt)
X[:, 1] = x1 # m; initial position (target position is zero)

if noisy
    sqrt_dt = √dt
    X^ = zeros(n, nt)
    X^[1, 1] = X[1, 1]
end

```

```

for t in 1:nt-1
    u[t] = -L[t, :]' * X^[:, t]
    X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt + sqrt(dt) * (Y * u[t] * randn() + G * randn(n))
    dy = C * X[:, t] * dt + D * sqrt(dt) * randn(n-1)
    X^[:, t+1] = X^[:, t] + (A * X^[:, t] + B * u[t]) * dt + K[t, :, :] * (dy - C * X^[:, t] * dt)
end
else
    for t in 1:nt-1
        u[t] = -L[t, :]' * X[:, t]
        X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt
    end
end
return X, u
end

function simulation_all(param, cost_param, L, K)
    Xa, ua = simulation(param, cost_param, L, K, noisy=false);

    # noisy
    nsim = 10
    XSimAll = []
    uSimAll = []
    for i in 1:nsim
        XSim, u = simulation(param, cost_param, L, K, noisy=true);
        push!(XSimAll, XSim)
        push!(uSimAll, u)
    end

    # visualization
    @unpack dt, T = cost_param
    tarray = collect(dt:dt:T)
    label = [L"Position ($m$)", L"Velocity ($m/s$)", L"Acceleration ($m/s^2$)", "L"Jerk ($m/s^3$)"]

    fig, ax = subplots(1, 3, figsize=(10, 3))
    for i in 1:2
        for j in 1:nsim
            ax[i].plot(tarray, XSimAll[j][i,:], "tab:gray", alpha=0.5)
        end

        ax[i].plot(tarray, Xa[i,:], "tab:red")
        ax[i].set_ylabel(label[i]); ax[i].set_xlabel(L"Time ($s$)");
        ax[i].set_xlim(0, T); ax[i].grid()
    end

    for j in 1:nsim
        ax[3].plot(tarray, uSimAll[j], "tab:gray", alpha=0.5)
    end
    ax[3].plot(tarray, ua, "tab:red")
    ax[3].set_ylabel(L"Control signal ($N\cdot m$)"); ax[3].set_xlabel(L"Time ($s$)");
    ax[3].set_xlim(0, T); ax[3].grid()
end

```

```
tight_layout()  
end
```

```
param = Reaching1DModelParameter()  
cost_param = Reaching1DModelCostParameter();
```

```
L, K, cost = LQG(param, cost_param);  
simulation_all(param, cost_param, L, K)
```

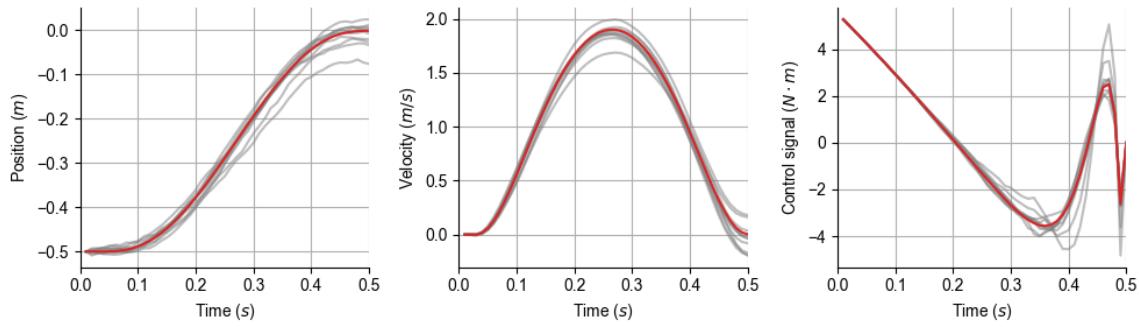


図 8.4 cell016.png

```
L, K, cost = glQG(param, cost_param);  
simulation_all(param, cost_param, L, K)
```

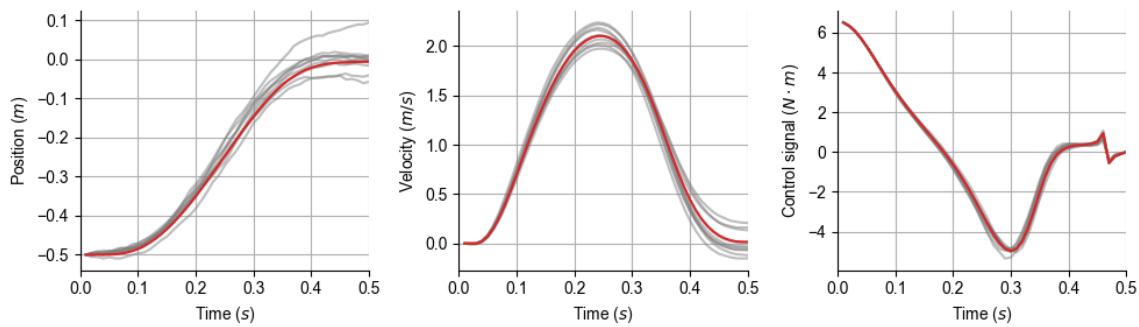


図 8.5 cell017.png

8.4 無限時間最適フィードバック制御モデル

8.4.1 モデルの構造

無限時間最適フィードバック制御モデル (infinite-horizon optimal feedback control model)
(Qian et al., 2013)

$$dx = (\mathbf{A}x + \mathbf{B}u)dt + \mathbf{Y}ud\gamma + \mathbf{G}d\omega \quad (8.30)$$

$$dy = \mathbf{C}xdt + \mathbf{D}d\xi \quad (8.31)$$

$$d\hat{x} = (\mathbf{A}\hat{x} + \mathbf{B}u)dt + \mathbf{K}(dy - \mathbf{C}\hat{x}dt) \quad (8.32)$$

8.4.2 実装

ライブラリの読み込みと関数の定義。

```
using Parameters: @unpack
using LinearAlgebra, Kronecker, Random, BlockDiagonals, PyPlot
rc("axes.spines", top=false, right=false)
rc("font", family="Arial")
```

定数の定義

$$\alpha_1 = \frac{b}{t_a t_e I}, \quad \alpha_2 = \frac{1}{t_a t_e} + \left(\frac{1}{t_a} + \frac{1}{t_e} \right) \frac{b}{I} \quad (8.33)$$

$$\alpha_3 = \frac{b}{I} + \frac{1}{t_a} + \frac{1}{t_e}, \quad b_u = \frac{1}{t_a t_e I} \quad (8.34)$$

```
@kwdef struct SaccadeModelParameter
    n = 4 # number of dims
    i = 0.25 # kgm^2,
    b = 0.2 # kgm^2/s
    ta = 0.03 # s
    te = 0.04 # s
    L0 = 0.35 # m

    bu = 1 / (ta * te * i)
    α1 = bu * b
    α2 = 1/(ta * te) + (1/ta + 1/te) * b/i
    α3 = b/i + 1/ta + 1/te

    A = [zeros(3) I(3); -[0, α1, α2, α3]']
    B = [zeros(3); bu]
    C = [I(3) zeros(3)]
    D = Diagonal([1e-3, 1e-2, 5e-2])
```

```

Y = 0.02 * B
G = 0.03 * I(n)

Q = Diagonal([1.0, 0.01, 0, 0])
R = 0.0001
U = Diagonal([1.0, 0.1, 0.01, 0])
end

```

$$\mathbf{X} := \begin{bmatrix} x \\ \tilde{x} \end{bmatrix}, d\bar{\omega} := \begin{bmatrix} d\omega \\ d\xi \end{bmatrix}, \bar{\mathbf{A}} := \begin{bmatrix} \mathbf{A} - \mathbf{BL} & \mathbf{BL} \\ \mathbf{0} & \mathbf{A} - \mathbf{KC} \end{bmatrix} \quad (8.35)$$

$$\bar{\mathbf{Y}} := \begin{bmatrix} -\mathbf{YL} & \mathbf{YL} \\ -\mathbf{YL} & \mathbf{YL} \end{bmatrix}, \bar{G} := \begin{bmatrix} \mathbf{G} & \mathbf{0} \\ \mathbf{G} & -\mathbf{KD} \end{bmatrix} \quad (8.36)$$

とする。元論文では F, \bar{F} が定義されていたが、 $F = 0$ とするため、以後の式から削除した。

$$\mathbf{P} := \begin{bmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{12} & \mathbf{P}_{22} \end{bmatrix} = \mathbb{E} [\mathbf{XX}^\top] \quad (8.37)$$

$$\mathbf{V} := \begin{bmatrix} \mathbf{Q} + \mathbf{L}^\top R \mathbf{L} & -\mathbf{L}^\top R \mathbf{L} \\ -\mathbf{L}^\top R \mathbf{L} & \mathbf{L}^\top R \mathbf{L} + \mathbf{U} \end{bmatrix} \quad (8.38)$$

aaa

$$K = \mathbf{P}_{22} \mathbf{C}^\top (\mathbf{D} \mathbf{D}^\top)^{-1} \quad (8.39)$$

$$\mathbf{L} = (R + \mathbf{Y}^\top (\mathbf{S}_{11} + \mathbf{S}_{22}) \mathbf{Y})^{-1} \mathbf{B}^\top \mathbf{S}_{11} \quad (8.40)$$

$$\bar{\mathbf{A}}^\top \mathbf{S} + \mathbf{S} \bar{\mathbf{A}} + \bar{\mathbf{Y}}^\top \mathbf{S} \bar{\mathbf{Y}} + \mathbf{V} = 0 \quad (8.41)$$

$$\bar{\mathbf{A}} \mathbf{P} + \mathbf{P} \bar{\mathbf{A}}^\top + \bar{\mathbf{Y}} \mathbf{P} \bar{\mathbf{Y}}^\top + \bar{\mathbf{G}} \bar{\mathbf{G}}^\top = 0 \quad (8.42)$$

$\mathbf{A} = (a_{ij})$ を $m \times n$ 行列、 $\mathbf{B} = (b_{kl})$ を $p \times q$ 行列とすると、それらのクロネッカー積 $\mathbf{A} \otimes \mathbf{B}$ は

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \quad (8.43)$$

で与えられる $mp \times nq$ 区分行列である。Roth's column lemma (vec-trick)

$$(\mathbf{B}^\top \otimes \mathbf{A}) \text{vec}(\mathbf{X}) = \text{vec}(\mathbf{AXB}) = \text{vec}(\mathbf{C}) \quad (8.44)$$

によりこれを解くと、

$$\mathbf{S} = -\text{vec}^{-1} \left((\mathbf{I} \otimes \bar{\mathbf{A}}^\top + \bar{\mathbf{A}}^\top \otimes \mathbf{I} + \bar{\mathbf{Y}}^\top \otimes \bar{\mathbf{Y}}^\top)^{-1} \text{vec}(\mathbf{V}) \right) \quad (8.45)$$

$$\mathbf{P} = -\text{vec}^{-1} \left((\mathbf{I} \otimes \bar{\mathbf{A}} + \bar{\mathbf{A}} \otimes \mathbf{I} + \bar{\mathbf{Y}} \otimes \bar{\mathbf{Y}})^{-1} \text{vec}(\bar{\mathbf{G}} \bar{\mathbf{G}}^\top) \right) \quad (8.46)$$

となる。ここで $\mathbf{I} = \mathbf{I}^\top$ を用いた。

K, L, S, P の計算

K, L, S, P の計算は次のようにする.

1. L と K をランダムに初期化
2. S と P を計算
3. L と K を更新
4. 収束するまで 2 と 3 を繰り返す.

収束スピードはかなり速い.

```

function infinite_horizon_ofc(param::SaccadeModelParameter, maxiter=1000, ε=1e-8)
    @unpack n, A, B, C, D, Y, G, Q, R, U = param

    # initialize
    L = rand(n)' # Feedback gains
    K = rand(n, 3) # Kalman gains
    I_2n = I(2n)

    for _ in 1:maxiter
        Ā = [A-B*L B*L; zeros(size(A)) (A-K*C)]
        Ȳ = [-ones(2) ones(2)] ⊗ (Y*L)
        Ḡ = [G zeros(size(K)); G (-K*D)]
        V = BlockDiagonal([Q, U]) + [1 -1; -1 1] ⊗ (L'* R * L)

        # update S, P
        S = -reshape((I_2n ⊗ (Ā)')' + (Ā)' ⊗ I_2n + (Ȳ)' ⊗ (Ȳ)') \ vec(V), (2n, 2n))
        P = -reshape((I_2n ⊗ Ā + Ā ⊗ I_2n + Ȳ ⊗ Ȳ) \ vec(Ḡ * (Ḡ)'), (2n, 2n))

        # update K, L
        P_22 = P[n+1:2n, n+1:2n]
        S_11 = S[1:n, 1:n]
        S_22 = S[n+1:2n, n+1:2n]

        K_t-1 = copy(K)
        L_t-1 = copy(L)

        K = P_22 * C' / (D * D')
        L = (R + Y' * (S_11 + S_22) * Y) \ B' * S_11
        if sum(abs.(K - K_t-1)) < ε && sum(abs.(L - L_t-1)) < ε
            break
        end
    end
    return L, K
end

param = SaccadeModelParameter()
L, K = infinite_horizon_ofc(param);

```

シミュレーション

関数を書く。

```

function simulation(param::SaccadeModelParameter, L, K, dt=0.001, T=2.0, ↵
    init_pos=-0.5; noisy=true)
    @unpack n, A, B, C, D, Y, G, Q, R, U = param
    nt = round(Int, T/dt)
    X = zeros(n, nt)
    u = zeros(nt)
    X[1, 1] = init_pos # m; initial position (target position is zero)

    if noisy
        sqrt_dt = √dt
        X^ = zeros(n, nt)
        X^[1, 1] = X[1, 1]
        for t in 1:nt-1
            u[t] = -L * X^[:, t]
            X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt + sqrt_dt * (Y * u[t] * ↵
                randn() + G * randn(n))
            dy = C * X[:, t] * dt + D * sqrt_dt * randn(n-1)
            X^[:, t+1] = X^[:, t] + (A * X^[:, t] + B * u[t]) * dt + K * (dy - C * ↵
                X^[:, t] * dt)
        end
    else
        for t in 1:nt-1
            u[t] = -L * X[:, t]
            X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt
        end
    end
    return X, u
end

```

理想状況でのシミュレーション

```

dt = 1e-3
T = 1.0

```

```
Xa, ua = simulation(param, L, K, dt, T, noisy=false);
```

ノイズを含むシミュレーション

ノイズを含む場合。

```

n = 4
nsim = 10
XSimAll = []
uSimAll = []
for i in 1:nsim

```

```
XSim, u = simulation(param, L, K, dt, T, noisy=true);
push!(XSimAll, XSim)
push!(uSimAll, u)
end
```

結果の描画

```
tarray = collect(dt:dt:T)
label = [L"Position ($m$)", L"Velocity ($m/s$)", L"Acceleration ($m/s^2$)", L"Jerk ←
($m/s^3$)"]
fig, ax = subplots(1, 3, figsize=(10, 3))
for i in 1:2
    for j in 1:n sim
        ax[i].plot(tarray, XSimAll[j][i,:], "tab:gray", alpha=0.5)
    end

    ax[i].plot(tarray, Xa[i,:], "tab:red")
    ax[i].set_ylabel(label[i]); ax[i].set_xlabel(L"Time ($s$)"); ax[i].set_xlim(0, T); ax[i].grid()
end

for j in 1:n sim
    ax[3].plot(tarray, uSimAll[j], "tab:gray", alpha=0.5)
end
ax[3].plot(tarray, ua, "tab:red")
ax[3].set_ylabel(L"Control signal ($N\cdot m$)"); ax[3].set_xlabel(L"Time ($s$)"); ←
ax[3].set_xlim(0, T); ax[3].grid()

tight_layout()
```

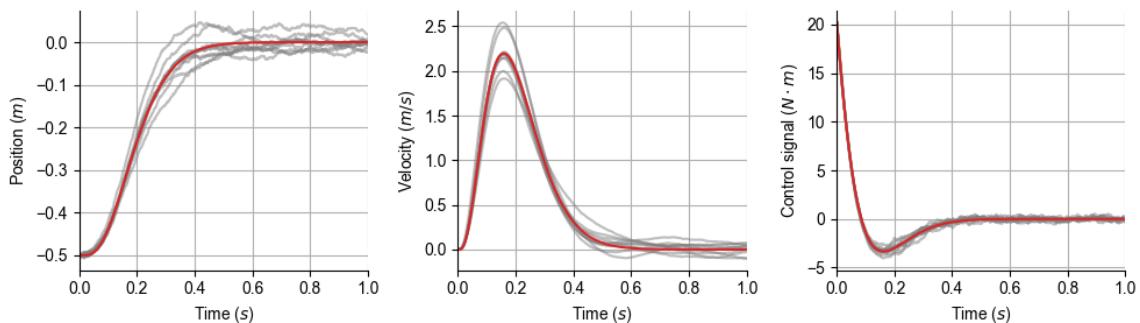


図 8.6 cell017.png

8.4.3 Target jump

target jump する場合の最適制御 (Li et al., 2018). 状態に target 位置も含むモデルであれば target 位置をずらせばよいが、ここでは自己位置をずらし target との相対位置を変化させることで target jump を実現する。

```

function target_jump_simulation(param::SaccadeModelParameter, L, K, dt=0.001, T=2.0,
    Ttj=0.4, tj_dist=0.1,
    init_pos=-0.5; noisy=true)
    # Ttj : target jumping timing (sec)
    # tj_dist : target jump distance
    @unpack n, A, B, C, D, Y, G, Q, R, U = param
    nt = round(Int, T/dt)
    ntj = round(Int, Ttj/dt)
    X = zeros(n, nt)
    u = zeros(nt)
    X[1, 1] = init_pos # m; initial position (target position is zero)

    if noisy
        sqrt_dt = sqrt(dt)
        X^ = zeros(n, nt)
        X^*[1, 1] = X[1, 1]
        for t in 1:nt-1
            if t == ntj
                X[1, t] -= tj_dist # When k == ntj, target ~
                jumpさせる (実際には現在の位置をずらす)
                X^*[1, t] -= tj_dist
            end
            u[t] = -L * X^[:, t]
            X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt + sqrt_dt * (Y * u[t] * ~
                randn() + G * randn(n))
            dy = C * X[:, t] * dt + D * sqrt_dt * randn(n-1)
            X^[:, t+1] = X^[:, t] + (A * X^[:, t] + B * u[t]) * dt + K * (dy - C * ~
                X^[:, t] * dt)
        end
    else
        for t in 1:nt-1
            if t == ntj
                X[1, t] -= tj_dist # When k == ntj, target ~
                jumpさせる (実際には現在の位置をずらす)
            end
            u[t] = -L * X[:, t]
            X[:, t+1] = X[:, t] + (A * X[:, t] + B * u[t]) * dt
        end
    end
    X[1, 1:ntj-1] .-= tj_dist;
    return X, u
end

```

Ttj = 0.4

```
tj_dist = 0.1
nt = round(Int, T/dt)
ntj = round(Int, Ttj/dt);
```

```
Xtj, utj = target_jump_simulation(param, L, K, dt, T, noisy=false);
```

```
XtjAll = []
utjAll = []
for i in 1:n sim
    XSim, u = target_jump_simulation(param, L, K, dt, T, noisy=true);
    push!(XtjAll, XSim)
    push!(utjAll, u)
end
```

```
target_pos = zeros(nt)
target_pos[1:ntj-1] .-= tj_dist;

fig, ax = subplots(1, 3, figsize=(10, 3))
for i in 1:2
    ax[i].plot(tarray, target_pos, "tab:green")
    for j in 1:n sim
        ax[i].plot(tarray, XtjAll[j][i,:], "tab:gray", alpha=0.5)
    end
    ax[i].axvline(x=Ttj, color="gray", linestyle="dashed")
    ax[i].plot(tarray, Xtj[i,:], "tab:red")
    ax[i].set_ylabel(label[i]); ax[i].set_xlabel(L"Time ($\$\$)"); ax[i].set_xlim(0, T); ax[i].grid()
end
for j in 1:n sim
    ax[3].plot(tarray, utjAll[j], "tab:gray", alpha=0.5)
end
ax[3].axvline(x=Ttj, color="gray", linestyle="dashed")
ax[3].plot(tarray, utj, "tab:red")
ax[3].set_ylabel(L"Control signal ($N \cdot \dot{m}$)"); ax[3].set_xlabel(L"Time ($\$\$)"); ax[3].set_xlim(0, T); ax[3].grid()

tight_layout()
```

8.5 ラット自由行動下の軌跡のシミュレーション

ラットが自由行動下において箱の中を探索する際の軌跡をシミュレーションする (Raudies and Hasselmo, 2012). これまでと異なり, 現象論的に運動を生成する. 場所細胞・格子細胞等自己位置と神経活動が相関する細胞のシミュレーションにおいて用いられる (George et al., 2024).

```
using PyPlot, LinearAlgebra, Random, Distributions
```

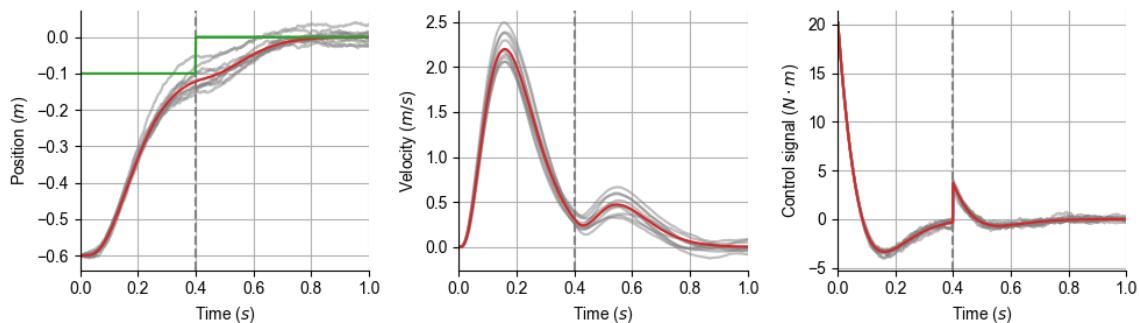


図 8.7 cell023.png

```
rc("axes.spines", top=false, right=false)
```

```
box_width, box_height = 0.8, 0.8; # Width and height of environment (meters)
perimeter_dist = 0.03 # Perimeter region distance to walls (meters)
σv = 0.13 # Forward velocity Rayleigh distribution scale (m/sec)
μω = 0.0 # Rotation velocity Gaussian distribution mean (rad/sec)
σω = (330 / 360) * 2π # Rotation velocity Gaussian distribution standard deviation ↴
(rad/sec)
dt = 0.02 # Simulation-step time increment (seconds)
decel_rate = 0.25; # velocity reduction factor when located in the perimeter
```

並進速度をレイリー分布、回転速度を正規分布に従うようにランダムサンプリングする。壁の接ベクトルとラットの距離を `dist_wall`、壁の法線ベクトルとラットの頭方向の角度の差を `angle_wall` とする。なお、壁とはラットの自己速度ベクトルと壁全体との交点である。

- ラットの自己速度ベクトルと壁全体との交点を求める。
- 交点の接ベクトルと法線ベクトルを求める。
- 接ベクトルとの距離を `dist_wall` とする。
- 法線ベクトルと成す角を `angle_wall` とする。

```
function min_dist_angle(pos, head_dir, wall_type="square")
    x, y = pos
    if wall_type == "square"
        dists = [box_width/2-x, box_height/2-y, box_width/2+x, box_height/2+y]
        dist_wall, nearest_wall = findmin(dists)
        angle_wall = mod(head_dir - (nearest_wall-1)*π/2 + π, 2π) - π
    elseif wall_type == "circle"
        dist_wall = box_width/2 - sqrt(x^2 + y^2)
        angle_wall = mod(head_dir - atan(y, x) + π, 2π) - π
    else
        @warn "'wall_type' must be 'square' or 'circle'"
```

```

    end
    return dist_wall, angle_wall
end;

```

```

function generate_trajectory(num_steps, wall_type)
    # store arrays
    position, velocity = zeros(num_steps, 2), zeros(num_steps, 2)
    head_dir = zeros(num_steps) # head direction
    speed = rand(Rayleigh(σv), num_steps) # Forward speed
    random_turn = rand(Normal(μω, σω), num_steps) * dt

    # initial values
    head_dir[1] = rand() * 2π
    position[1, :] = (rand(2) .- 0.5) .* ([box_width, box_height] .- perimeter_dist)

    # iteration of trajectory
    for t in 1:num_steps-1
        turn_angle = random_turn[t]
        dist_wall, angle_wall = min_dist_angle(position[t, :], head_dir[t], wall_type)
        if (dist_wall < perimeter_dist) && (abs(angle_wall) < π/2) # avoid wall
            speed[t] *= decel_rate # deceleration
            turn_angle += sign(angle_wall) * (π/2 - abs(angle_wall))
        end
        velocity[t, :] = speed[t] * [cos(head_dir[t]), sin(head_dir[t])]
        position[t+1, :] = position[t, :] + velocity[t, :] * dt
        head_dir[t+1] = mod(head_dir[t] + turn_angle, 2π) # turn,
    end
    return position, velocity, speed, head_dir
end;

```

5分間のシミュレーションを行う。

```

T = 300 # simulation time (sec)
num_steps = round(Int, T/dt)
wall_types = ["square", "circle"]
positions = zeros(2, num_steps, 2)
for i in 1:2
    positions[i, :, :, -] = generate_trajectory(num_steps, wall_types[i]);
end

```

黒点から始まり、赤点に終わる。

```

figure(figsize=(8, 4))
for i in 1:2
    subplot(1,2,i)
    title("Wall type: "*wall_types[i])
    xlabel("x (meters)"); ylabel("y (meters)")
    plot(positions[i, 1, 1], positions[i, 1, 2], "ko", label="Start")
    plot(positions[i, end, 1], positions[i, end, 2], "ro", label="Goal")
    plot(positions[i, :, 1], positions[i, :, 2], color="k", alpha=0.3)
end

```

```
tight_layout()
```

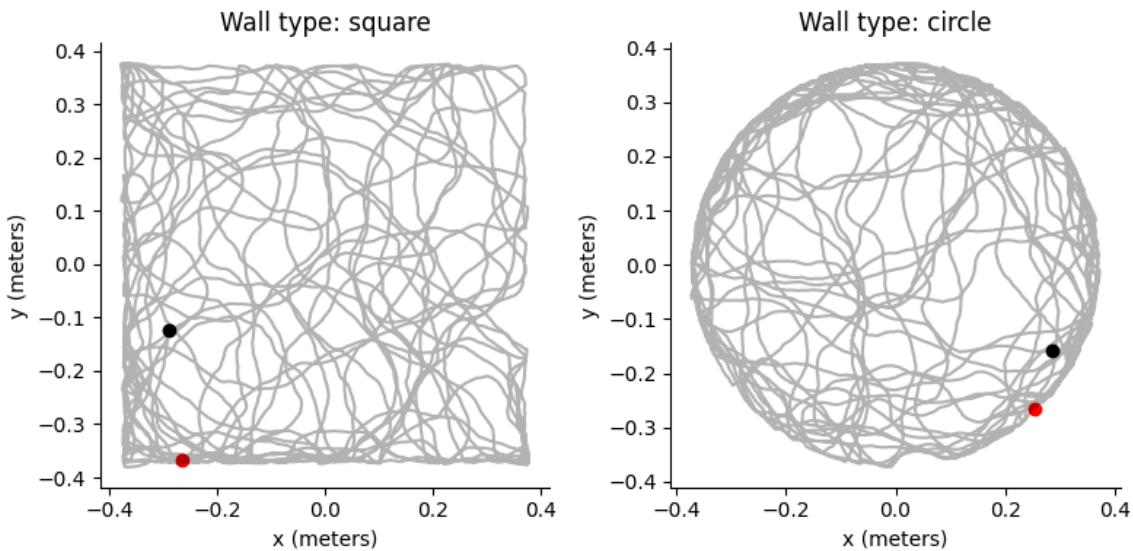


図 8.8 cell009.png

参考文献

- Flash, T. and Hogan, N. (1985). “The coordination of arm movements: an experimentally confirmed mathematical model”. *J. Neurosci.* 5.7, pp. 1688–1703.
- George, T. M. et al. (2024). “RatInABox, a toolkit for modelling locomotion and neuronal activity in continuous environments”. *Elife* 13.
- Harris, C. M. and Wolpert, D. M. (1998). “Signal-dependent noise determines motor planning”. *Nature* 394.6695, pp. 780–784.
- Li, Z. et al. (2018). “A Single, Continuously Applied Control Policy for Modeling Reaching Movements with and without Perturbation”. *Neural Comput.* 30.2, pp. 397–427.
- Qian, N. et al. (2013). “Movement duration, Fitts’s law, and an infinite-horizon optimal feedback control model for biological motor systems”. *Neural Comput.* 25.3, pp. 697–724.
- Raudies, F. and Hasselmo, M. E. (2012). “Modeling boundary vector cell firing given optic flow as a cue”. *PLoS Comput. Biol.* 8.6, e1002553.
- Yazdani, M. et al. (2012). “A simple control policy for achieving minimum jerk trajectories”. *Neural Netw.* 27, pp. 74–80.

第 9 章

神経回路網によるベイズ推論

9.1 ベイズ線形回帰 (Bayesian linear regression)

共役事前分布 (conjugate prior) を

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \quad (9.1)$$

と定義し、事後分布 (posterior) を

$$p(\mathbf{w} | \mathbf{Y}, \mathbf{X}) = \mathcal{N}(\mathbf{w} | \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}) \quad (9.2)$$

とする。ただし、

$$\hat{\boldsymbol{\Sigma}}^{-1} = \boldsymbol{\Sigma}_0^{-1} + \beta \Phi^\top \Phi \quad (9.3)$$

$$\hat{\boldsymbol{\mu}} = \hat{\boldsymbol{\Sigma}} (\beta \Phi^\top \mathbf{y} + \boldsymbol{\Sigma}_0^{-1} \boldsymbol{\mu}_0) \quad (9.4)$$

である。また、 $\Phi = \phi(\mathbf{x})$ であり、 $\phi(x) = [1, x, x^2, x^3]$ 、 $\boldsymbol{\mu}_0 = \mathbf{0}$ 、 $\boldsymbol{\Sigma}_0 = \alpha^{-1} \mathbf{I}$ とする。テストデータを \mathbf{x}^* とした際、予測分布は

$$p(y^* | \mathbf{x}^*, \mathbf{Y}, \mathbf{X}) = \mathcal{N}(y^* | \boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*) \quad (9.5)$$

となる。ただし、

$$\boldsymbol{\mu}^* = \hat{\boldsymbol{\mu}}^\top \phi(\mathbf{x}^*) \quad (9.6)$$

$$\boldsymbol{\Sigma}^* = \frac{1}{\beta} + \phi(\mathbf{x}^*)^\top \hat{\boldsymbol{\Sigma}} \phi(\mathbf{x}^*) \quad (9.7)$$

$$(9.8)$$

である。

```
using Base: @kwdef
using Parameters: @unpack
using PyPlot, LinearAlgebra, Random, Distributions
rc("axes.spines", top=false, right=false)
```

```
# Generate Toy datas
num_train, num_test = 20, 100 # sample size
dims = 4 # dimensions
σy = 0.3

polynomial_expansion(x; degree=3) = stack([x .^ p for p in 0:degree]);

Random.seed!(0);
x = rand(num_train)
y = sin.(2π*x) + σy * randn(num_train);
Φ = polynomial_expansion(x, degree=dims-1) # design matrix

xtest = range(-0.1, 1.1, length=num_test)
ytest = sin.(2π*xtest)
Φtest = polynomial_expansion(xtest, degree=dims-1);
```

```
@kwdef mutable struct BayesianLinearReg
    μ_hat::Array
    Σ_hat::Array
end

# Training params & definition of model
function BayesianLinearReg(Φ, y, α, β)
    Σ_hat = inv(α * I + β * Φ' * Φ)
    μ_hat = β * Σ_hat * Φ' * y;
    return BayesianLinearReg(μ_hat=μ_hat, Σ_hat=Σ_hat)
end;

function predict(Φ, blr::BayesianLinearReg, β)
    @unpack μ_hat, Σ_hat = blr
    μp = Φ * μ_hat
    σp = sqrt.(1/β .+ diag(Φ * Σ_hat * Φ'));
    return μp, σp
end;

function sampling_func(Φ, blr::BayesianLinearReg, num_sampling::Int)
    @unpack μ_hat, Σ_hat = blr
    dist = MvNormal(μ_hat, Matrix(Hermitian(Σ_hat)))
    sampled_params = rand(dist, num_sampling);
    return Φ * sampled_params
end;
```

```
α, β = 1e-3, 5.0;

blr = BayesianLinearReg(Φ, y, α, β);
μtest, σtest = predict(Φtest, blr, β);

num_sampling = 5
sampled_func = sampling_func(Φtest, blr, num_sampling);
```

```

figure(figsize=(5,3.5))
title("Bayesian Linear Regression")
scatter(x, y, facecolor="None", edgecolors="black", s=25) # samples
plot(xtest, ytest, "--", label="Actual", color="tab:red") # regression line
plot(xtest, ptest, label="Predicted mean", color="tab:blue") # regression line
fill_between(xtest, ptest+ptest, ptest-ptest, alpha=0.5, color="tab:gray", ~
    label="Predicted std.")
for i in 1:num_sampling
    plot(xtest, sampled_func[:, i], alpha=0.3, color="tab:green")
end
xlabel("x"); ylabel("y"); legend()
xlim(-0.1, 1.1); tight_layout()

```

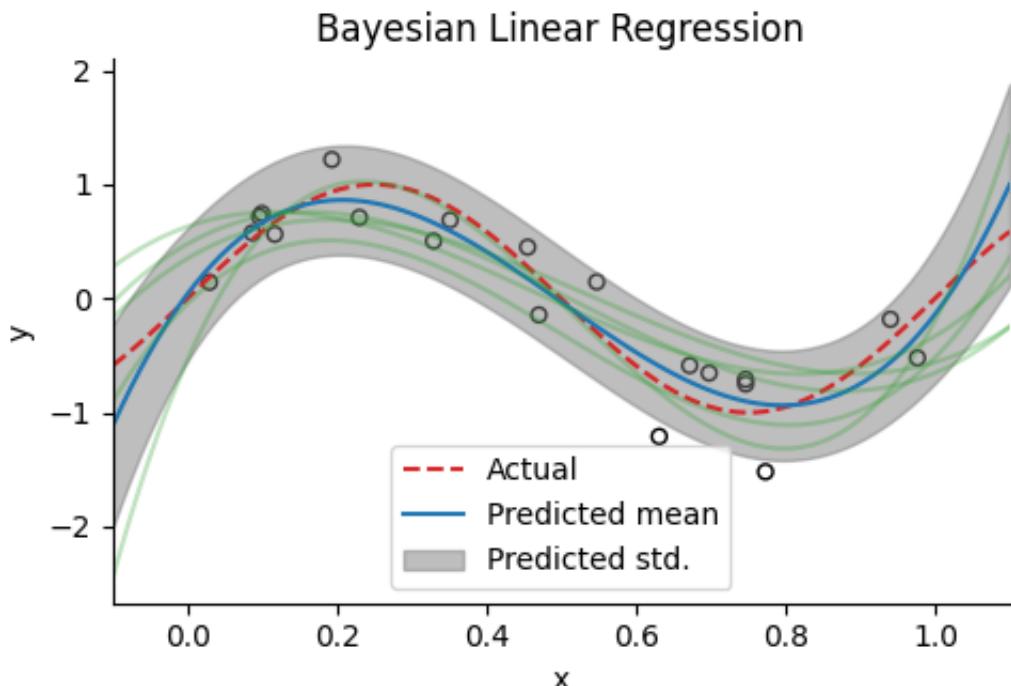


図 9.1 cell005.png

9.2 マルコフ連鎖モンテカルロ法 (MCMC)

前節では解析的に事後分布の計算をした。事後分布を近似的に推論する方法の1つにマルコフ連鎖モンテカルロ法 (Markov chain Monte Carlo methods; MCMC) がある。他の近似推論の手法としては Laplace 近似や変分推論 (variational inference) などがある。MCMC は他の手法に比して、事後

分布の推論だけでなく、確率分布を神経活動で表現する方法を提供するという利点がある。データを X とし、パラメータを θ とする。

$$p(\theta | X) = \frac{p(X | \theta)p(\theta)}{\int p(X | \theta)p(\theta)d\theta} \quad (9.9)$$

分母の積分計算 $\int p(X | \theta)p(\theta)d\theta$ が求まればよい。

モンテカルロ法

マルコフ連鎖

9.2.1 Metropolis-Hastings 法

```
using Base: @kwdef
using Parameters: @unpack
using PyPlot, LinearAlgebra, Random, Distributions, ForwardDiff, KernelDensity
rc("axes.spines", top=false, right=false)

mixed_gauss = MixtureModel([MvNormal(zeros(2), I), MvNormal(3*ones(2), I)], [0.5, ~
0.5]) # 分布を混ぜる

x = -3:0.1:6
pd(x1, x2) = logpdf(mixed_gauss, [x1, x2])

mixed_gauss_heat = pd.(x, x');

xpos = x * ones(size(x))'

fig = plt.figure(figsize=(4,3))
ax = fig.add_subplot(projection="3d")
surf = ax.plot_surface(xpos, xpos', -mixed_gauss_heat, cmap="viridis")
ax.set_xlim(-3, 6); ax.set_ylim(-3, 6);
ax.set_xlabel(L"\theta_1"); ax.set_ylabel(L"\theta_2"); ax.set_zlabel(L"-log ~
p");
tight_layout()
```

Metropolis-Hastings 法における採用・不採用アルゴリズム。

```
# Metropolis-Hastings method; log_p: unnormalized log-posterior
function gaussian_mh(log_p::Function, θ_init::Vector{Float64}, σ::Float64, ~
num_iter::Int)
    d = length(θ_init)
    samples = zeros(d, num_iter)
    num_accepted = 0
    θ = θ_init # init position
    for m in 1:num_iter
        θ_ = rand(MvNormal(θ, σ*I))
        mH = log_p(θ) + logpdf(MvNormal(θ, σ*I), θ_) # initial Hamiltonian
```

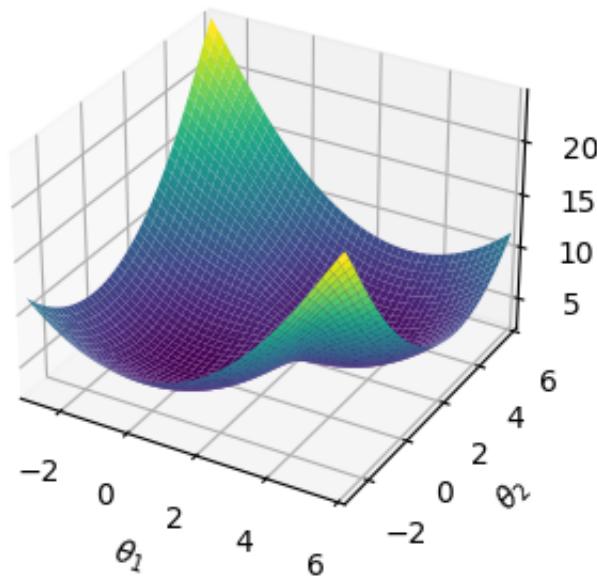


図 9.2 cell004.png

```
mH_ = log_p(theta_) + logpdf(MvNormal(theta_, sigma*I), theta) # final Hamiltonian

if min(1, exp(mH_ - mH)) > rand()
    theta = theta_ # accept
    num_accepted += 1
end
samples[:, m] = theta
end
return samples, num_accepted
end;
```

```
log_p(theta) = logpdf(mixed_gauss, theta);
grad(theta)= ForwardDiff.gradient(log_p, theta)
```

```
theta_m, num_accepted = gaussian_mh(log_p, [1.0,0.5], 1.0, 2000)
```

```
size(theta_m)
```

```
Um = kde((theta_m[1, :], theta_m[2, :]));
```

```

fig, ax = subplots(1, 2, figsize=(5, 3), sharex="all", sharey="all")
fig.suptitle("Metropolis-Hastings method")
ax[1].set_title("Raw trace")
ax[1].contour(x, x, -mixed_gauss_heat)
ax[1].plot(θm[1, :], θm[2, :], color="tab:red", alpha=0.5)
ax[1].set_xlim(-3,6); ax[1].set_ylim(-3,6)
ax[2].set_title("Density")
ax[2].contourf(Um.x, Um.x, Um.density)
fig.tight_layout()

```

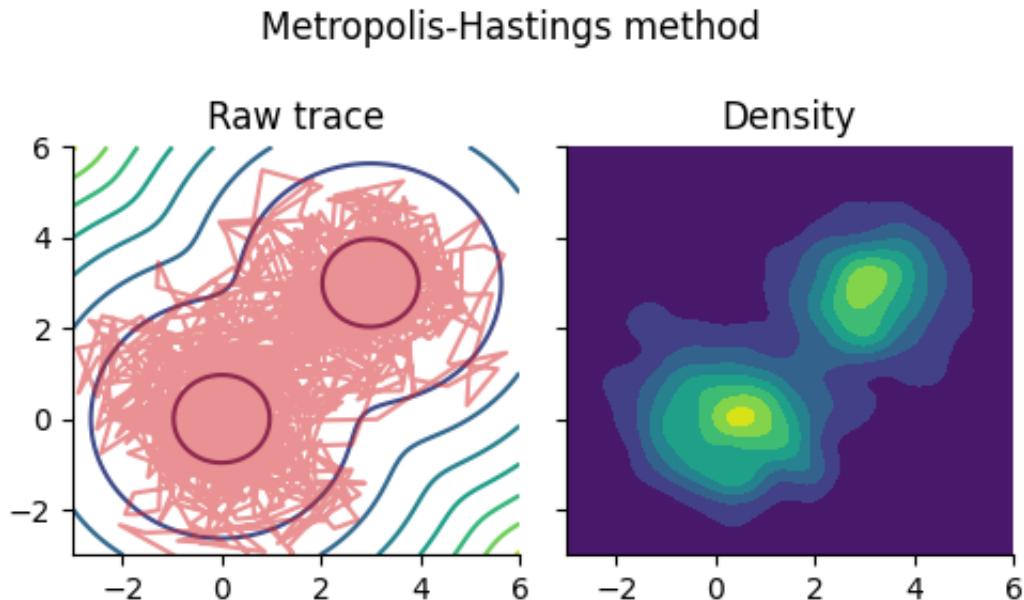


図 9.3 cell010.png

9.2.2 ランジュバン・モンテカルロ法 (LMC)

拡散過程

$$\frac{d\theta}{dt} = \nabla \log p(\theta) + \sqrt{2}dW \quad (9.10)$$

Euler – Maruyama 法により,

```

nt = 10000
ε = 0.1

```

```
β = 1
ρ = sqrt(2*ε);

θl = zeros(nt, 2)
θ = [1.0, 0.5]
for t in 1:nt
    θ += ε * β * grad(θ) + ρ * randn(2)
    θl[t, :] = θ
end

Ul = kde((θl[:, 1], θl[:, 2]));

fig, ax = subplots(1, 2, figsize=(5, 3), sharex="all", sharey="all")
fig.suptitle("Langevin dynamics")
ax[1].set_title("Raw trace")
ax[1].contour(x, x, -mixed_gauss_heat)
ax[1].plot(θl[:, 1], θl[:, 2], color="tab:red", alpha=0.5)
ax[1].set_xlim(-3,6); ax[1].set_ylim(-3,6)
ax[2].set_title("Density")
ax[2].contourf(Ul.x, Ul.x, Ul.density)
fig.tight_layout()
```

Langevin dynamics

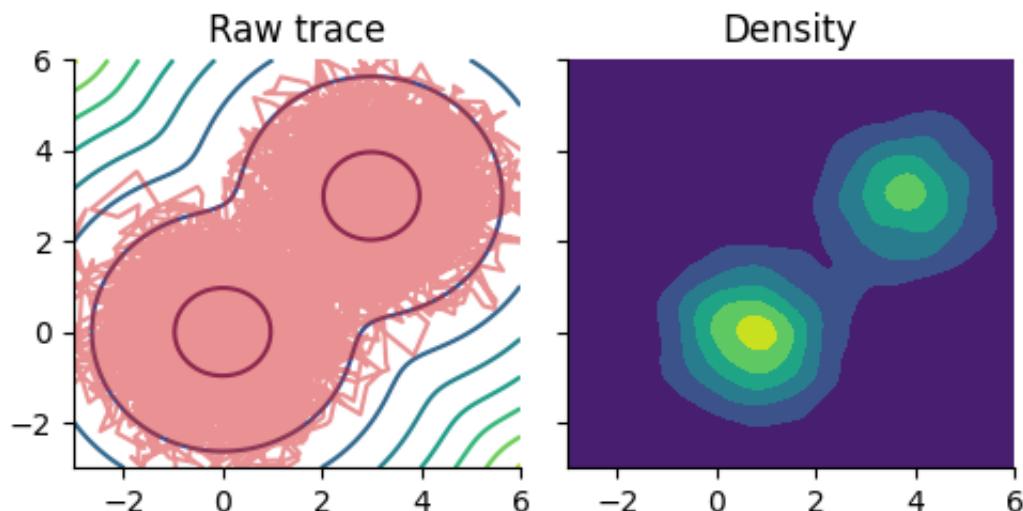


図 9.4 cell015.png

9.2.3 ハミルトニアン・モンテカルロ法 (HMC 法)

LMC よりも一般的な MCMC の手法として Hamiltonian モンテカルロ法 (Hamiltonian Monte Carlo; HMC) あるいはハイブリッド・モンテカルロ法 (Hybrid Monte Carlo) がある。エネルギーポテンシャルの局面上を Hamilton 力学に従ってパラメータを運動させることにより高速にサンプリングする手法である。一般化座標を \mathbf{q} 、一般化運動量を \mathbf{p} とする。ポテンシャルエネルギーを $U(\mathbf{q})$ としたとき、古典力学（解析力学）において保存力のみが作用する場合のハミルトニアン（Hamiltonian） $\mathcal{H}(\mathbf{q}, \mathbf{p})$ は

$$\mathcal{H}(\mathbf{q}, \mathbf{p}) := U(\mathbf{q}) + \frac{1}{2} \|\mathbf{p}\|^2 \quad (9.11)$$

となる。このとき、次の 2 つの方程式が成り立つ。

$$\frac{d\mathbf{q}}{dt} = \frac{\partial \mathcal{H}}{\partial \mathbf{p}} = \mathbf{p}, \quad \frac{d\mathbf{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{q}} = -\frac{\partial U}{\partial \mathbf{q}} \quad (9.12)$$

これを **ハミルトンの運動方程式** (hamilton's equations of motion) あるいは**正準方程式** (canonical equations) という。リープfrog（leap frog）法により離散化する。

```
function leapfrog(grad::Function, θ::Vector{Float64}, p::Vector{Float64}, ε::Float64, L::Int)
    for l in 1:L
        p += 0.5 * ε * grad(θ)
        θ += ε * p
        p += 0.5 * ε * grad(θ)
    end
    return θ, p
end;
```

```
# Hamiltonian Monte Carlo method; log_p: unnormalized log-posterior
function hmc(log_p::Function, θ_init::Vector{Float64}, ε::Float64, L::Int, num_iter::Int)
    grad(θ)= ForwardDiff.gradient(log_p, θ)
    d = length(θ_init)
    samples = zeros(d, num_iter)
    num_accepted = 0
    θ = θ_init # init position
    for m in 1:num_iter
        p = randn(d) # get momentum
        H = -log_p(θ) + 0.5 * p' * p # initial Hamiltonian
        θ_, p_ = leapfrog(grad, θ, p, ε, L) # update
        H_ = -log_p(θ_) + 0.5 * p_'' * p_ # final Hamiltonian

        if min(1, exp(H - H_)) > rand()
            θ = θ_ # accept
            num_accepted += 1
        end
        samples[:, m] = θ
    end
end
```

```

    end
    return samples, num_accepted
end;

```

```

ps = zeros(nt, 2)
θs = zeros(nt, 2)
p = randn(2)
θ = randn(2)
for t in 1:nt
    if t in 20:10:nt
        p = randn(2)
    end
    p += 0.5 * ε * grad(θ)
    θ += ε * p
    p += 0.5 * ε * grad(θ)
    ps[t, :] = p
    θs[t, :] = θ
end

```

```
Us = kde((θs[:, 1], θs[:, 2]));
```

```

fig, ax = subplots(1, 2, figsize=(5, 3), sharex="all", sharey="all")
fig.suptitle("Hamiltonian dynamics")
ax[1].set_title("Raw trace")
ax[1].contour(x, x, -mixed_gauss_heat)
ax[1].plot(θs[:, 1], θs[:, 2], color="tab:red", alpha=0.5)
ax[1].set_xlim(-3,6); ax[1].set_ylim(-3,6)
ax[2].set_title("Density")
ax[2].contourf(Us.x, Us.x, Us.density)
fig.tight_layout()

```

ToDo: 自己相関確認する

9.2.4 線形回帰への適応

```

# Generate Toy data
num_train, num_test = 20, 100 # sample size
dims = 4 # dimensions
σy = 0.3

polynomial_expansion(x; degree=3) = stack([x .^ p for p in 0:degree]);

Random.seed!(0);
x = rand(num_train)
y = sin.(2π*x) + σy * randn(num_train);
Φ = polynomial_expansion(x, degree=dims-1) # design matrix

```

Hamiltonian dynamics

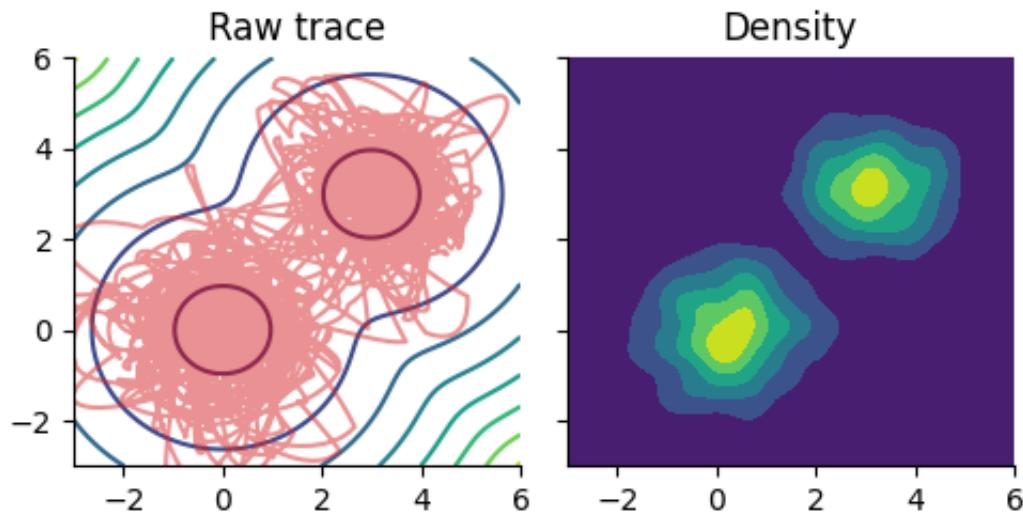


図 9.5 cell021.png

```
xtest = range(-0.1, 1.1, length=num_test)
ytest = sin.(2π*xtest)
ϕtest = polynomial_expansion(xtest, degree=dims-1);
```

```
log_joint(w, ϕ, y, σy, μ₀, Σ₀) = sum(logpdf.(Normal.(ϕ * w, σy), y)) + ←
    logpdf(MvNormal(μ₀, Σ₀), w);
```

```
α, β = 5e-3, 5.0
```

```
w = randn(dims)
μ₀ = zeros(dims)
Σ₀ = 1/α * I;
```

```
ulp(w) = log_joint(w, ϕ, y, σy, μ₀, Σ₀)
```

```
w_init = rand(MvNormal(μ₀, Σ₀), 1)[:, 1]
```

```
@time samples, num_accepted = hmc(ulp, w_init, 1e-2, 10, 500)

plot(samples[1, :])
```

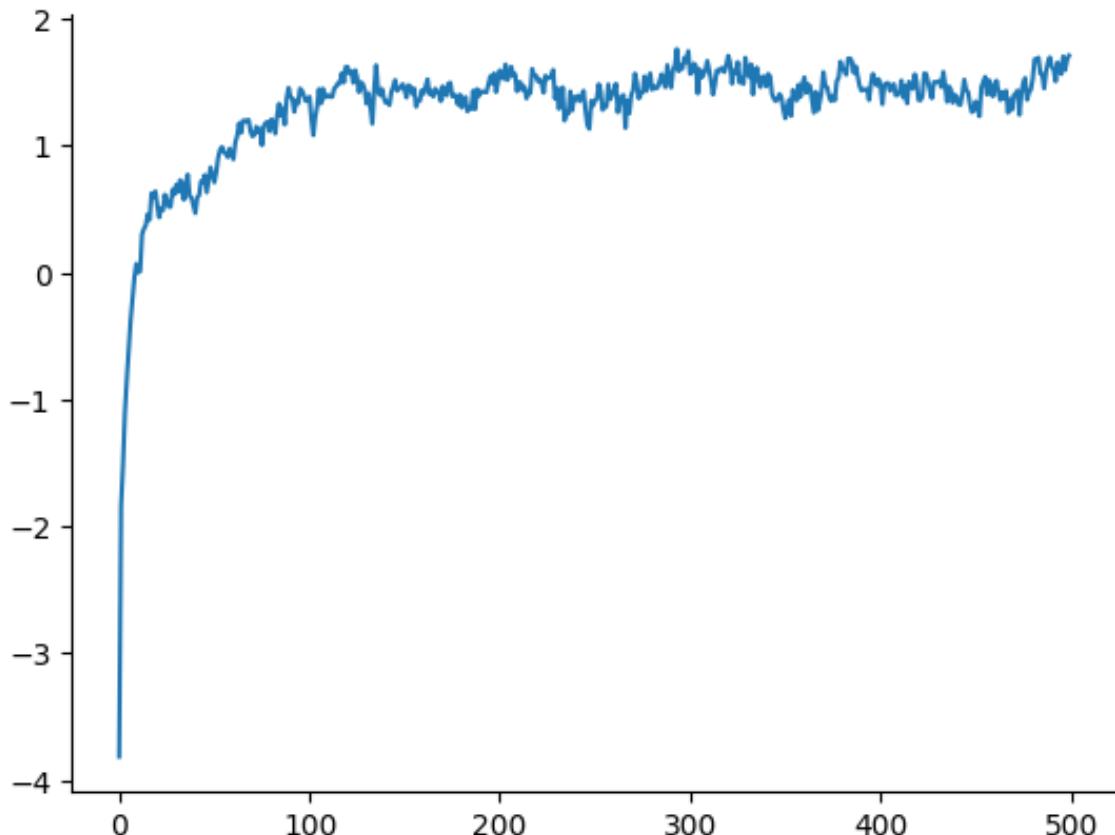


図 9.6 cell031.png

```
yhmc = φtest * samples[:, 300:end];
yhmc_mean = mean(yhmc, dims=2)[:,];
yhmc_std = std(yhmc, dims=2)[:,];
```

```
figure(figsize=(5,3.5))
title("Bayesian Linear Regression")
scatter(x, y, facecolor="None", edgecolors="black", s=25) # samples
```

```

plot(xtest, ytest, "--", label="Actual", color="tab:red") # regression line
plot(xtest, yhmc_mean, label="Predicted mean", color="tab:blue") # regression line
fill_between(xtest, yhmc_mean+yhmc_std, yhmc_mean-yhmc_std, alpha=0.5, ~
    color="tab:gray", label="Predicted std.")
for i in 1:5
    plot(xtest, yhmc[:, end-i], alpha=0.3, color="tab:green")
end
xlabel("x"); ylabel("y"); legend()
xlim(-0.1, 1.1); tight_layout()

```

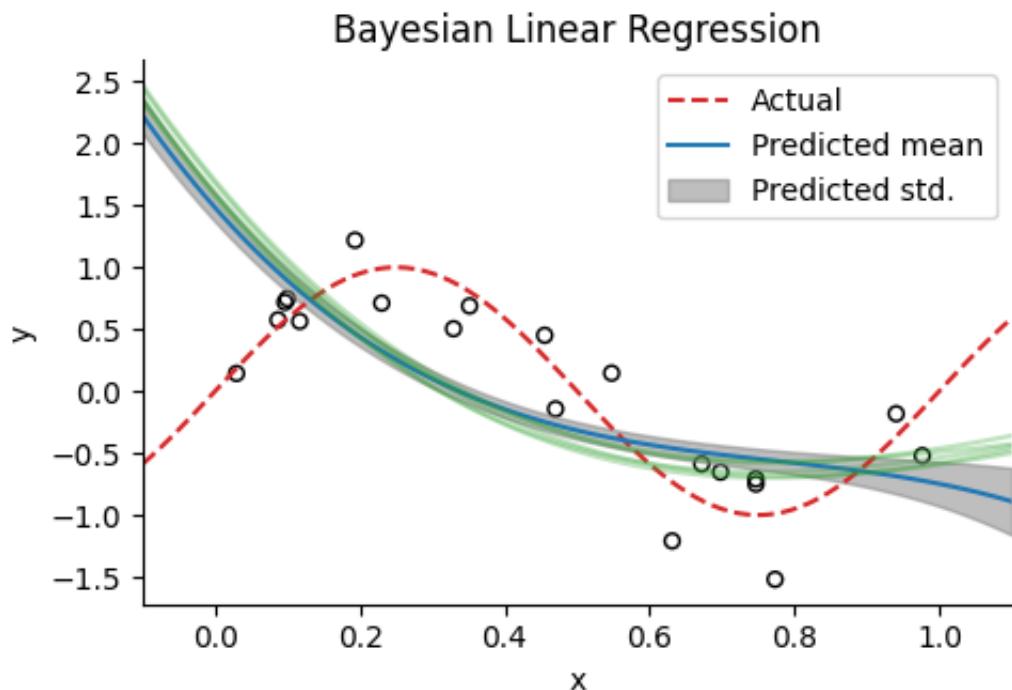


図 9.7 cell033.png

9.3 神経サンプリング

サンプリングに基づく符号化 (sampling-based coding; SBC or neural sampling model) をガウス尺度混合モデルを例にとり実装する。

9.3.1 ガウス尺度混合モデル

ガウス尺度混合 (Gaussian scale mixture; GSM) モデルは確率的生成モデルの一種である (Wainwright and Simoncelli, 1999)(Orbán et al., 2016). GSM モデルでは入力を次式で予測する：

$$\text{入力} = z \left(\sum \text{神経活動} \times \text{基底} \right) + \text{ノイズ} \quad (9.13)$$

前節までのスペース符号化モデル等と同様に、入力が基底の線形和で表されるとしている。ただし、尺度 (scale) パラメータ z が基底の線形和に乘じられている点が異なる。^{*1}

事前分布

$\mathbf{x} \in \mathbb{R}^{N_x}$, $\mathbf{A} \in \mathbb{R}^{N_x \times N_y}$, $\mathbf{y} \in \mathbb{R}^{N_y}$, $\mathbf{z} \in \mathbb{R}$ とする。

$$p(\mathbf{x} | \mathbf{y}, z) = \mathcal{N}(z\mathbf{Ay}, \sigma_x^2 \mathbf{I}) \quad (9.14)$$

事前分布を

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{C}) \quad (9.15)$$

$$p(z) = \Gamma(k, \vartheta) \quad (9.16)$$

とする。 $\Gamma(k, \vartheta)$ はガンマ分布であり、 k は形状 (shape) パラメータ、 ϑ は尺度 (scale) パラメータである。 $p(\mathbf{y})$ は \mathbf{y} の事前分布であり、刺激がない場合の自発活動の分布を表していると仮定する。

重み行列 \mathbf{A} の作成

```
using PyPlot, LinearAlgebra, Random, Distributions, KernelDensity, StatsBase
using PyPlot: matplotlib
Random.seed!(0)
rc("axes.spines", top=false, right=false)

function gabor(x, y, θ, σ=1, λ=2, ψ=0)
    xθ = x * cos(θ) + y * sin(θ)
    yθ = -x * sin(θ) + y * cos(θ)
    return exp(-.5(xθ^2 + yθ^2)/σ^2) * cos(2π/λ * xθ + ψ)
end;
```

^{*1} コードは (Orbán et al., 2016) https://github.com/gergoorban/sampling_in_gsm, および (Echeveste et al., 2020) https://bitbucket.org/RSE_1987/ssn_inference_numerical_experiments/src/master/ を参考に作成した。

```

function get_A(WH, Ny)
    Nx = WH^2
    A = zeros(Nx, Ny) # weight matrix
    p = range(-3, 3, length=WH) # position
    θ = (1:Ny) / Ny * π # theta for gabor
    for i in 1:Ny
        gb = gabor.(p', p, θ[i])
        gb /= norm(gb) + 1e-8 # normalization
        A[:, i] = gb[:] # flatten and save
    end
    return A
end;

WH = 16    # width/height of input image
Nx = WH^2 # dimension of the observed variable x
Ny = 50   # dimension of the hidden variable y

A = get_A(WH, Ny);

```

重み行列 \mathbf{A} の一部を描画してみよう.

```

figure(figsize=(2,2))
plot_idx = [2,4,6,8]
weight_idx = [37,25,50,13]
titles = ["", "0°", "±90°", ""]
for i in 1:4
    subplot(3,3,plot_idx[i])
    title(titles[i])
    imshow(reshape(A[:, weight_idx[i]], WH, WH), cmap="gray")
    axis("off")
end
subplots_adjust(wspace=0.01, hspace=0.01)

```

分散共分散行列 \mathbf{C} の作成

\mathbf{C} は y の事前分布の分散共分散行列である. (Orbán et al., 2016) では自然画像を用いて作成しているが, ここでは簡単のため \mathbf{A} と同様に (Echeveste et al., 2020) に従って作成する. 前項で作成した通り, \mathbf{A} の各基底には周期性があるため, 類似した基底を持つニューロン同士は類似した出力をすると考えられる. Echeveste らは $\theta \in [-\pi/2, \pi/2]$ の範囲において Fourier 基底を複数作成し, そのグラム行列 (Gram matrix) を係数倍したものを \mathbf{C} と設定している. ここではガウス過程 (Gaussian process) モデルとの類似性から, 周期カーネル (periodic kernel)

$$K(\theta, \theta') = \exp \left[\phi_1 \cos \left(\frac{|\theta - \theta'|}{\phi_2} \right) \right] \quad (9.17)$$

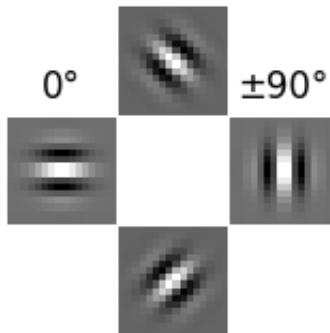


図 9.8 cell007.png

を用いる。ここでは $|\theta - \theta'| = m\pi$ ($m = 0, 1, \dots$) の際に類似度が最大になればよいので, $\phi_2 = 0.5$ とする。これが正定値行列になるように単位行列の係数倍 eI を加算し, スケーリングした上で, `Symmetric(C)` や `Matrix(Hermitian(C))` により実対象行列としたものを `C` とする。`C` を正定値行列にする理由は Julia の `MvNormal` が Cholesky 分解を用いて多変量正規分布の乱数を生成するためである。事前に `cholesky(C)` が実行できるか確認するのもよい。

```
function get_C(Ny, C_range=[-0.5, 4.0], eps=0.1, ψ₁=2.0, ψ₂=0.5)
    K(x₁, x₂, ψ₁, ψ₂) = exp(ψ₁ * cos(abs(x₁-x₂) / ψ₂)) # periodic kernel
    θ = (1:Ny) / Ny * π # theta for gabor
    C = K.(θ', θ, ψ₁, ψ₂) # create covariance matrix
    C += eps * I # regularization to make C positive definite
    C_min, C_max = minimum(C), maximum(C)
    C = C_range[1] .+ (C_range[2]-C_range[1]) * (C .- C_min) / (C_max - C_min)
    return Symmetric(C); # make symmetric matrix using upper triangular matrix
end;
```

```
C = get_C(Ny)

figure(figsize=(3,2))
title(L"\$\\mathbf{C}\$")
ims = imshow(C, origin="lower", cmap="bwr", vmin=-4, vmax=4, extent=(-90, 90, -90, 90))
xticks([-90,0,90]); yticks([-90,0,90]);
xlabel(L"\$\\theta\$ (Pref. ori)"); ylabel(L"\$\\theta\$ (Pref. ori)")
colorbar(ims);
tight_layout()
```

ここで Pref. ori は最適方位 (preferred orientation) を意味する。

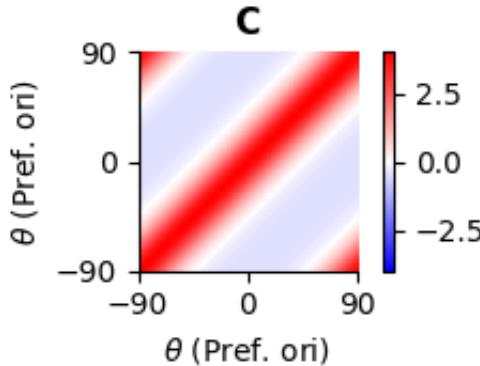


図 9.9 cell010.png

事後分布の計算

事後分布は z と \mathbf{y} のそれぞれについて次のように求められる.

$$p(z | \mathbf{x}) \propto p(z) \mathcal{N} \left(0, z^2 \mathbf{A} \mathbf{C} \mathbf{A}^\top + \sigma_x^2 \mathbf{I} \right) \quad (9.18)$$

$$p(\mathbf{y} | z, \mathbf{x}) = \mathcal{N} (\mu(z, \mathbf{x}), \Sigma(z)) \quad (9.19)$$

ただし,

$$\Sigma(z) = \left(\mathbf{C}^{-1} + \frac{z^2}{\sigma_x^2} \mathbf{A}^\top \mathbf{A} \right)^{-1} \quad (9.20)$$

$$\mu(z, \mathbf{x}) = \frac{z}{\sigma_x^2} \Sigma(z) \mathbf{A}^\top \mathbf{x} \quad (9.21)$$

である. 最終的な予測において z の事後分布は必要でないため, $p(\mathbf{y} | z, \mathbf{x})$ から z を消去することを考えよう. 厳密に行う場合, 次式のように周辺化 (marginalization) により, z を (積分) 消去する必要がある.

$$p(\mathbf{y} | \mathbf{x}) = \int dz p(z | \mathbf{x}) \cdot p(\mathbf{y} | z, \mathbf{x}) \quad (9.22)$$

周辺化においては, まず z の MAP 推定 (最大事後確率推定) 値 z_{MAP} を求める.

$$z_{\text{MAP}} = \underset{z}{\operatorname{argmax}} p(z | \mathbf{x}) \quad (9.23)$$

次に z_{MAP} の周辺で $p(z | \mathbf{x})$ を積分し, 積分値が一定の閾値を超える z の範囲を求め, この範囲で z を積分消去してやればよい. しかし, z は単一のスカラー値であり, この手法で推定するのは煩雑であるために近似手法が (Echeveste et al., 2017) において提案されている. Echeveste らは第一の近似として, z の分布を z_{MAP} でのデルタ関数に置き換える, すなわち, $p(z | \mathbf{x}) \simeq \delta(z - z_{\text{MAP}})$ とすることを提案して

いる。この場合、 z は定数とみなせ、 $p(\mathbf{y} | \mathbf{x}) \simeq p(\mathbf{y} | \mathbf{x}, z = z_{\text{MAP}})$ となる。第二の近似として、 z_{MAP} を真のコントラスト z^* で置き換えることが提案されている。GSMへの入力 \mathbf{x} は元の画像を $*$ とすると、 $\mathbf{x} = z^* *$ としてスケーリングされる。この入力の前処理の際に用いる z^* を用いてしまおうということである。この場合、 $p(\mathbf{y} | \mathbf{x}) \simeq p(\mathbf{y} | \mathbf{x}, z = z^*)$ となる。しかし、入力を任意の画像とする場合、 z^* は未知である。簡便さと精度のバランスを取り、ここでは第一の近似、 $z = z_{\text{MAP}}$ とする手法を用いることにする。

```
# log pdf of p(z)
log_Pz(z, k, theta) = logpdf(Gamma(k, theta), z)

# pdf of p(z|x)
function Pz_x(z_range, x, A * A', sigma_x^2, k, theta)
    n_contrasts = length(z_range)
    log_p = zeros(n_contrasts)
    mu_xz = zeros(size(x))
    dz = z_range[2] - z_range[1]
    for i in 1:n_contrasts
        Cx_z = z_range[i]^2 * A * A' + sigma_x^2 * I
        log_p[i] = log_Pz(z_range[i], k, theta) + logpdf(MvNormal(mu_xz, Symmetric(Cx_z)), x)
    end
    p = exp.(log_p .- maximum(log_p)) # for numerical stability
    p /= sum(p) * dz
    return p
end;
```

```
# mean and covariance matrix of p(y|x, z)
function post_moments(x, z, sigma_x^2, A, A' * A, C^-1)
    Sigma_z = inv(C^-1 + (z^2 / sigma_x^2) * A' * A)
    mu_zx = (z / sigma_x^2) * Sigma_z * A' * x
    return mu_zx, Sigma_z
end;
```

シミュレーション

```
A' * A = A' * A
A * A' = A * C * A'

sigma_x = 1.0 # Noise of the x process
sigma_x^2 = sigma_x^2
k, theta = 2.0, 2.0 # Parameter of the gamma dist. for z (Shape, Scale)

C^-1 = inv(C); # inverse of C
```

入力データの作成

```
Z = [0.0, 0.25, 0.5, 1.0, 2.0] # true contrasts z^*
n_samples = size(Z)[1]
y = rand(MvNormal(zeros(Ny), C), 1) # sampling from P(y)=N(0, C)
```

```
X = stack([rand(MvNormal(vec(z*A*y), sigma_x*I)) for z in Z])';
```

```
x_min, x_max = minimum(X), maximum(X)

figure(figsize=(4,2))
for s in 1:n_samples
    subplot(1, n_samples, s)
    title(L"$z$: " * string(Z[s]))
    imshow(reshape(X[s, :], WH, WH), vmin=x_min, vmax=x_max, cmap="gray")
    axis("off")
end
tight_layout()
```

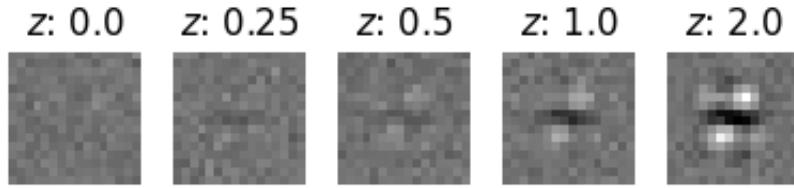


図 9.10 cell019.png

事後分布の計算をする。

```
mu_post = zeros(n_samples, Ny)
sigma_post = zeros(n_samples, Ny)
Sigma_post = zeros(n_samples, Ny, Ny)

z_range = range(0, 5.0, length=100) # range of z for MAP estimation
Z_MAP = zeros(n_samples)

for s in 1:n_samples
    p_z = Pz_x(z_range, X[s, :], ACA^T, sigma_x^2, k, theta)
    Z_MAP[s] = z_range[argmax(p_z)] # MAP estimated z
    mu_post[s, :, :, :] = post_moments(X[s, :], Z_MAP[s], sigma_x^2, A, A^TA, C^-1)
    sigma_post[s, :, :] = sqrt.(diag(Sigma_post[s, :, :, :]))
end
```

結果

```
theta_s = range(-90, 90, length=Ny)
cm = get_cmap(:Greens) # get color map
cms = cm.((1:n_samples)/n_samples) # color list

fig, ax = subplots(1, 3, figsize=(7.5, 2))
ax[1].scatter(Z, Z_MAP, c=cms)
ax[1].plot(Z, Z_MAP, color="tab:gray", zorder=0)
```

```

ax[1].set_xlabel(L"$z$"); ax[1].set_ylabel(L"$z_{MAP}$");
for s in 1:n_samples
    ax[2].plot(θs, μ_post[s, :], color=cms[s])
    ax[3].plot(θs, σ_post[s, :], color=cms[s], label=L"$z$ : "*string(Z[s]))
end
ax[2].set_ylabel(L"$\mu$"); ax[3].set_ylabel(L"$\sigma$")
for i in 2:3
    ax[i].set_xticks([-90,0,90])
    ax[i].set_xlabel(L"$\theta$ (Pref. ori)")
end
ax[3].legend(bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0)
tight_layout()

```

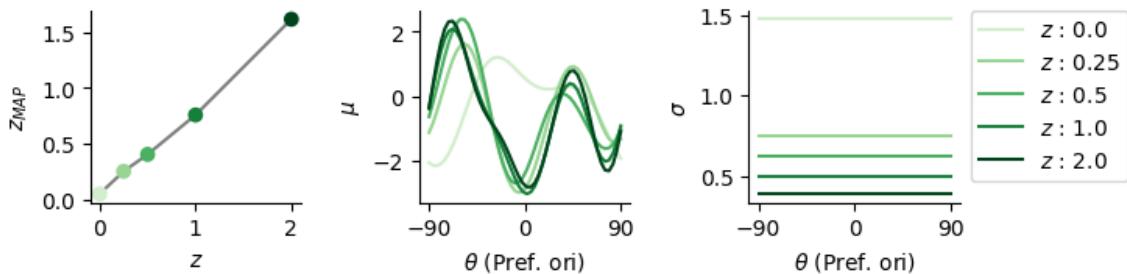


図 9.11 cell023.png

```

fig, ax = subplots(1, n_samples, figsize=(7.5, 1), sharex="all", sharey="all")
for s in 1:n_samples
    ax[s].set_title(L"$z$ : "*string(Z[s]))
    ims = ax[s].imshow(Σ_post[s, :, :], origin="lower", cmap="bwr", extent=(-90, 90, -90, 90), vmin=-1, vmax=1)
    ax[s].set_xticks([-90,0,90]); ax[s].set_yticks([-90,0,90]);
    if s == 1
        ax[s].set_ylabel(L"$\theta$ (Pref. ori)")
    elseif s == ceil(Int, n_samples/2)
        ax[s].set_xlabel(L"$\theta$ (Pref. ori)");
    end
end
fig.colorbar(ims, ax=ax[n_samples]);

```

出力のサンプリングを行う。

```
membrane_potential(y, α=2.4, β=1.9, γ=0.6) = α * max(0, y+β)^γ
```

事後分布から応答をサンプリングする。

```

nt = 1000
h_gsm = zeros(n_samples, Ny, nt)

```

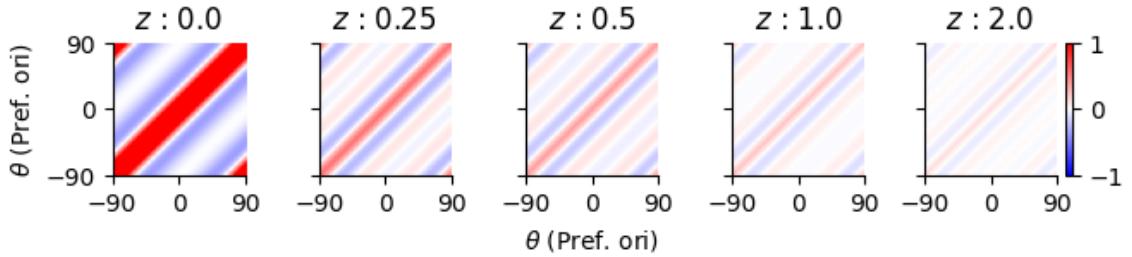


図 9.12 cell024.png

```

for s in 1:n_samples
    μ = μ_post[s, :]
    Σ = Σ_post[s, :, :]
    sample = rand(MvNormal(μ, Symmetric(Σ)), nt)
    h_gsm[s, :, :] = membrane_potential.(sample)
end

```

```

# modified from ~
# https://matplotlib.org/stable/gallery/statistics/confidence_ellipse.html
function confidence_ellipse(x, y, ax, n_std=3, alpha=1, facecolor="none", ~
    edgecolor="tab:gray")
    pearson = cor(x,y)
    rx, ry = sqrt(1 + pearson), sqrt(1 - pearson)
    ellipse = matplotlib.patches.Ellipse((0, 0), width=2*rx, height=2*ry, alpha=alpha,
        fc=facecolor, ec=edgecolor, lw=2, zorder=0)
    scales = [std(x), std(y)] * n_std
    means = [mean(x), mean(y)]
    transf =
        matplotlib.transforms.Affine2D().rotate_deg(45).scale(scales...).translate(means...)
    ellipse.set_transform(transf + ax.transData)
    return ax.add_patch(ellipse)
end;

```

```

fig, ax = subplots(figsize=(4, 3))
unit_idx = [1, 25]
for s in 1:n_samples
    h₁, h₂ = h_gsm[s, unit_idx[1], :], h_gsm[s, unit_idx[2], :]
    ax.plot(h₁[1:15], h₂[1:15], marker="o", markersize=5, alpha=0.5, color=cms[s], ~
        label=L"$z=$"+string(Z[s]))
    confidence_ellipse(h₁, h₂, ax, 3, 1, "none", cms[s])
end
ax.set_xlabel("Neuron #"+string(unit_idx[1])); ax.set_ylabel("Neuron ~
    #"+string(unit_idx[2]))
axins = [ax.inset_axes([0.85, -0.25, 0.15, 0.15]), ax.inset_axes([-0.3, 0.85, 0.15, 0.15])]
for i in 1:2
    axins[i].imshow(reshape(A[:,unit_idx[i]], WH, WH), cmap="gray")
    axins[i].axis("off")

```

```

end
ax.set_aspect("equal", "box")
ax.legend(bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0)
tight_layout()

```

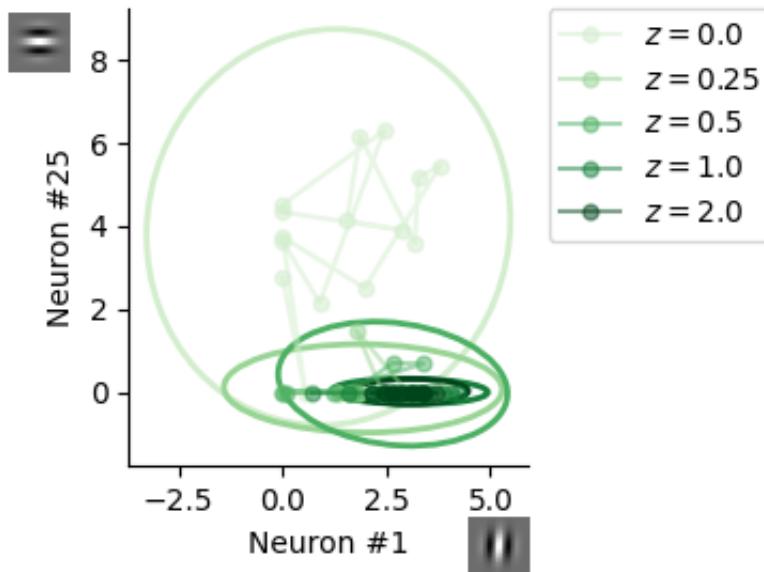


図 9.13 cell030.png

9.3.2 興奮性・抑制性神経回路によるサンプリング

前節で実装した MCMC を興奮性・抑制性神経回路 (excitatory-inhibitory (E-I) network) で実装する。HMC と LMC の両方を神経回路で実装する。ハミルトニアンを用いる場合、一般化座標 \mathbf{q} を興奮性神経細胞の活動 \mathbf{u} 、一般化運動量 \mathbf{p} を抑制性神経細胞の活動 \mathbf{v} に対応させる。 \mathbf{u}, \mathbf{v} は同じ次元のベクトルとする。 \mathbf{u}, \mathbf{v} の時間発展はハミルトニアン \mathcal{H} を導入して

$$\tau \frac{d\mathbf{u}}{dt} = \frac{\partial \mathcal{H}}{\partial \mathbf{v}}, \quad \tau \frac{d\mathbf{v}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{u}} \quad (9.24)$$

と書ける。一般的には $\mathcal{H}(\mathbf{u}, \mathbf{v}) = E(\mathbf{u}) + \frac{1}{2}\mathbf{v}^\top \mathbf{v}$ であり、 $p(\mathbf{u}, \mathbf{v}) \propto \exp(-\mathcal{H}(\mathbf{u}, \mathbf{v}))$ である。力学的エネルギーを保つ運動は、対数同時分布における等値線上の運動と同じである。(Aitchison and Lengyel, 2016) では

$$\mathcal{H}(\mathbf{u}, \mathbf{v}) = \log p(\mathbf{u}, \mathbf{v}) + \text{Const.} = \log p(\mathbf{v}|\mathbf{u}) + \log p(\mathbf{u}) + \text{Const.} \quad (9.25)$$

とし, $p(\mathbf{v}|\mathbf{u}) = \mathcal{N}(\mathbf{v}; \mathbf{B}\mathbf{u}, \mathbf{M}^{-1})$, $p(\mathbf{u}) = \mathcal{N}(\mathbf{0}, \mathbf{C}^{-1})$ としている. この場合,

$$\frac{d\mathbf{u}}{dt} = \frac{1}{\tau} \frac{\partial \mathcal{H}}{\partial \mathbf{v}} = \frac{1}{\tau} \frac{\partial \log p(\mathbf{u}, \mathbf{v})}{\partial \mathbf{v}} = \frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{v}} \quad (9.26)$$

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\tau} \frac{\partial \mathcal{H}}{\partial \mathbf{u}} = -\frac{1}{\tau} \frac{\partial \log p(\mathbf{u}, \mathbf{v})}{\partial \mathbf{u}} = -\frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{u}} - \frac{1}{\tau} \frac{\partial \log p(\mathbf{u})}{\partial \mathbf{u}} \quad (9.27)$$

となる. このままでは等値線上を運動することになるので, Langevin ダイナミクスを付け加える.

$$\frac{d\mathbf{u}}{dt} = \frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{v}} + \frac{1}{\tau_L} \frac{\partial \log p(\mathbf{u}, \mathbf{v})}{\partial \mathbf{u}} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.28)$$

$$= \frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{v}} + \frac{1}{\tau_L} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{u}} + \frac{1}{\tau_L} \frac{\partial \log p(\mathbf{u})}{\partial \mathbf{u}} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.29)$$

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{u}} - \frac{1}{\tau} \frac{\partial \log p(\mathbf{u})}{\partial \mathbf{u}} + \frac{1}{\tau_L} \frac{\partial \log p(\mathbf{u}, \mathbf{v})}{\partial \mathbf{v}} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.30)$$

$$= -\frac{1}{\tau} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{u}} + \frac{1}{\tau_L} \frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{v}} - \frac{1}{\tau} \frac{\partial \log p(\mathbf{u})}{\partial \mathbf{u}} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.31)$$

となる. それぞれの項は

$$\frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{v}} = \mathbf{B}^\top \mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) \quad (9.32)$$

$$\frac{\partial \log p(\mathbf{v}|\mathbf{u})}{\partial \mathbf{u}} = -\mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) \quad (9.33)$$

$$\frac{\partial \log p(\mathbf{u})}{\partial \mathbf{u}} = -\mathbf{C}\mathbf{u} \quad (9.34)$$

であるので,

$$\frac{d\mathbf{u}}{dt} = \frac{1}{\tau} \mathbf{B}^\top \mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) - \frac{1}{\tau_L} \mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) - \frac{1}{\tau_L} \mathbf{C}\mathbf{u} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.35)$$

$$\frac{d\mathbf{v}}{dt} = \frac{1}{\tau} \mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) + \frac{1}{\tau_L} \mathbf{B}^\top \mathbf{M} (\mathbf{B}\mathbf{u} - \mathbf{v}) + \frac{1}{\tau} \mathbf{C}\mathbf{u} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.36)$$

となる. $\mathbf{B} = \mathbf{I}$ とすると,

$$\frac{d\mathbf{u}}{dt} = \frac{1}{\tau} \mathbf{M} (\mathbf{u} - \mathbf{v}) - \frac{1}{\tau_L} \mathbf{M} (\mathbf{u} - \mathbf{v}) - \frac{1}{\tau_L} \mathbf{C}\mathbf{u} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.37)$$

$$= \left[\left(\frac{1}{\tau} - \frac{1}{\tau_L} \right) \mathbf{M} - \frac{1}{\tau_L} \mathbf{C} \right] \mathbf{u} - \left(\frac{1}{\tau} - \frac{1}{\tau_L} \right) \mathbf{M}\mathbf{v} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.38)$$

$$\frac{d\mathbf{v}}{dt} = \frac{1}{\tau} \mathbf{M} (\mathbf{u} - \mathbf{v}) + \frac{1}{\tau_L} \mathbf{M} (\mathbf{u} - \mathbf{v}) + \frac{1}{\tau} \mathbf{C}\mathbf{u} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.39)$$

$$= \left[\left(\frac{1}{\tau} + \frac{1}{\tau_L} \right) \mathbf{M} + \frac{1}{\tau_L} \mathbf{C} \right] \mathbf{u} - \left(\frac{1}{\tau} + \frac{1}{\tau_L} \right) \mathbf{M}\mathbf{v} + \sqrt{\frac{2}{\tau_L}} d\eta \quad (9.40)$$

となり, \mathbf{u} , \mathbf{v} と定行列およびノイズに依存してサンプリングダイナミクスを記述できる. 長々と式変形を書いたが, 重要なのは興奮性・抑制性という 2 種類の細胞群の相互作用により生み出された振動を用

いてサンプリングにおける自己相関を下げることができるという点である。簡単のため、前項で用いた入力刺激のうち、最も z が大きいサンプルのみを使用する。

```

dt = 1e-2 # ms
τ, τl = 10.0, 150.0 # ms
α_in = [1/τ - 1/τl, 1/τ + 1/τl]
α_ext = [1/τl, -1/τ]
ρ = sqrt(2*dt/τl);

nt = 50000
M = cat(ones(1,1), C; dims=(1,2));
x_idx = n_samples # get last x
x = X[x_idx, :]
u_init = [1; zeros(Ny)];

```



```

function ∇u logP(u, x, σx², A, C⁻¹)
    z, y = abs(u[1]), u[2:end]
    pred_error = A' * (x - z*A*y) / σx² # prediction error signal
    du = zeros(size(u))
    du[1] = sign(u[1]) * (y' * pred_error - z)
    du[2:end] = z * pred_error - C⁻¹*y
    return du
end

```



```

∇log_p(u) = ∇u logP(u, x, σx², A, C⁻¹);

```



```

function neural_llmc(∇log_p::Function, u_init::Vector{Float64}, α::Float64, ρ::Float64, dt::Float64, nt::Int)
    p = length(u_init)
    d = length(u_init)
    u = zeros(nt, d)
    u[1, :] = u_init

    for t in 1:nt-1
        I_ext = ∇log_p(u[t, :]) # external input
        u[t+1, :] = u[t, :] + dt * (α * I_ext) + ρ * randn(d)
    end
    return u
end

```



```

function neural_hmc(∇log_p::Function, u_init::Vector{Float64}, M::Matrix{Float64},
    α_in::Vector{Float64}, α_ext::Vector{Float64}, ρ::Float64, dt::Float64, nt::Int)
    d = length(u_init)
    u, v = zeros(nt, d), zeros(nt, d)
    u[1, :] = u_init

    for t in 1:nt-1
        I_ext = ∇log_p(u[t, :]) # external input
        I_in = M * (u[t, :] - v[t, :]) # internal input
        u[t+1, :] = u[t, :] + dt * (α_in[1] * I_in + α_ext[1] * I_ext) + ρ * randn(d)
    end
    return u
end

```

```

    v[t+1, :] = v[t, :] + dt * (α_in[2] * I_in + α_ext[2] * I_ext) + ρ * randn(d)
end
return u, v
end;

```

```
@time u_nlmc = neural_lmc(∇log_p, u_init, α_ext[1], ρ, dt, nt);
```

```
@time u_nhmc, v_nhmc = neural_hmc(∇log_p, u_init, M, α_in, α_ext, ρ, dt, nt);
```

初めの 100ms は burn-in 期間として除く。またダウンサンプリングする。

```
L = 100
burn_in = 10000
mcmc_time = (burn_in*dt):(L*dt):(nt*dt); # time for plot
```

```
mean_nlmc = mean(u_nlmc[burn_in:L:end, 2:end], dims=2); # pseudo-LFP
mean_nhmc = mean(u_nhmc[burn_in:L:end, 2:end], dims=2);
```

```
autocorr_nlmc = autocor(mean_nlmc, 1:length(mean_nlmc)-1);
autocorr_nhmc = autocor(mean_nhmc, 1:length(mean_nhmc)-1);
```

z の推定過程を描画する。また、 z を除いた \mathbf{u} を平均化し、自己相関の度合いを確認する。

```

fig, ax = subplots(1,2, figsize=(6,2.5), sharex="all")
ax[1].plot(mcmc_time, u_nhmc[burn_in:L:end, 1], color="tab:red")
ax[1].plot(mcmc_time, u_nlmc[burn_in:L:end, 1], color="tab:blue")
ax[1].axhline(Z[x_idx], linestyle="dashed", color="tab:gray", alpha=0.5)
ax[1].set_xlabel("Time (ms)");
ax[1].set_ylabel("Estimated $z$")
ax[1].set_xlim(mcmc_time[1], mcmc_time[end])

ax[2].plot(mcmc_time[1:end-1], autocorr_nhmc, color="tab:red", label="Hamiltonian")
ax[2].plot(mcmc_time[1:end-1], autocorr_nlmc, color="tab:blue", label="Langevin")
ax[2].set_xlabel("Time (ms)");
ax[2].set_ylabel("Autocorrelation")
ax[2].set_xlim(mcmc_time[1], mcmc_time[end])
ax[2].axhline(0, linestyle="dashed", color="tab:gray", alpha=0.5)
ax[2].legend()
fig.tight_layout()

```

Hamiltonian ネットワークは自己相関を振動により低下させることで、効率の良いサンプリングを実現している。ToDo: 普通に MCMC やる場合も自己相関は確認したほうがいいという話をどこかに書く。推定された事後分布を特定の神経細胞のペアについて確認する。

```

h_nhmc = membrane_potential.(u_nhmc[burn_in:L:end, :])
h_nlmc = membrane_potential.(u_nlmc[burn_in:L:end, :])

```

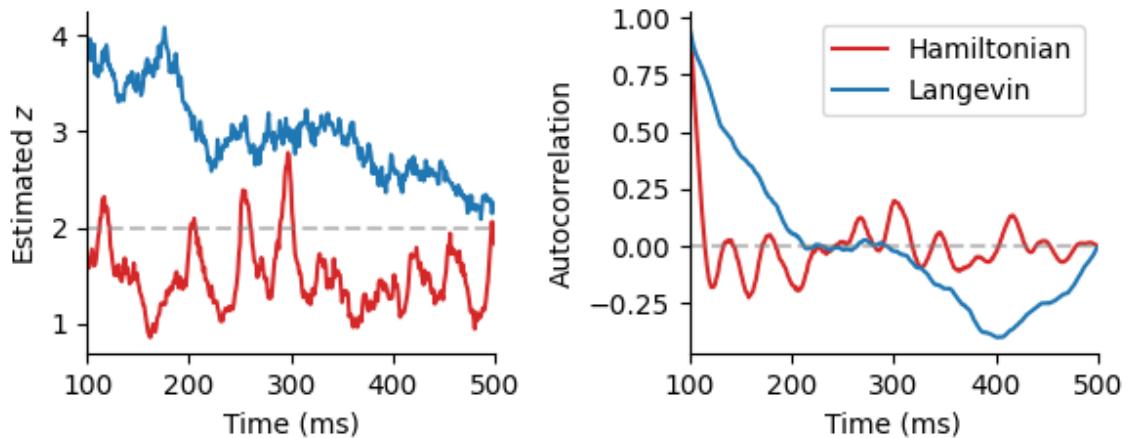


図 9.14 cell045.png

```

kde_bound = ((-3,5),(0,8)) # ((xlo,xhi),(ylo,yhi))
U_gsm = kde((h_gsm[x_idx, unit_idx[1], :], h_gsm[x_idx, unit_idx[2], :]), ~
    boundary=kde_bound)
U_nhmc = kde((h_nhmc[:, unit_idx[1]+1], h_nhmc[:, unit_idx[2]+1]), boundary=kde_bound)
U_nlmc = kde((h_nlmc[:, unit_idx[1]+1], h_nlmc[:, unit_idx[2]+1]), ~
    boundary=kde_bound);

```

```

fig, ax = plt.subplots(1,3, figsize=(6, 2.5), sharey="all", sharex="all")
ax[1].contourf(U_gsm.x, U_gsm.x, U_gsm.density)
ax[1].set_title("Actual")
ax[2].contourf(U_nhmc.x, U_nhmc.x, U_nhmc.density)
ax[2].set_title("Hamiltonian")
ax[3].contourf(U_nlmc.x, U_nlmc.x, U_nlmc.density)
ax[3].set_title("Langevin")
ax[1].set_ylabel("Neuron #" + string(unit_idx[2]))
ax[2].set_xlabel("Neuron #" + string(unit_idx[1]))
fig.tight_layout()

```

Hamiltonian ネットワークの方が安定して事後分布を推定することができている。ToDo: 以下の記述。ここでは重みを設定したが、(Echeveste et al., 2020) では RNN に BPTT で重みを学習させている。動的な入力に対するサンプリング (Berkes et al., 2011)。burn-in がなくなり効率良くサンプリングできる。

9.3.3 Spiking ニューラルネットワークにおけるサンプリング

前項で挙げた例は発火率モデルであったが、SNNにおいてサンプリングを実行する機構自体は考案されている。ToDo: 以下の記述。(Buesing et al., 2011)(Masset et al., 2022)(Zhang et al., 2022)

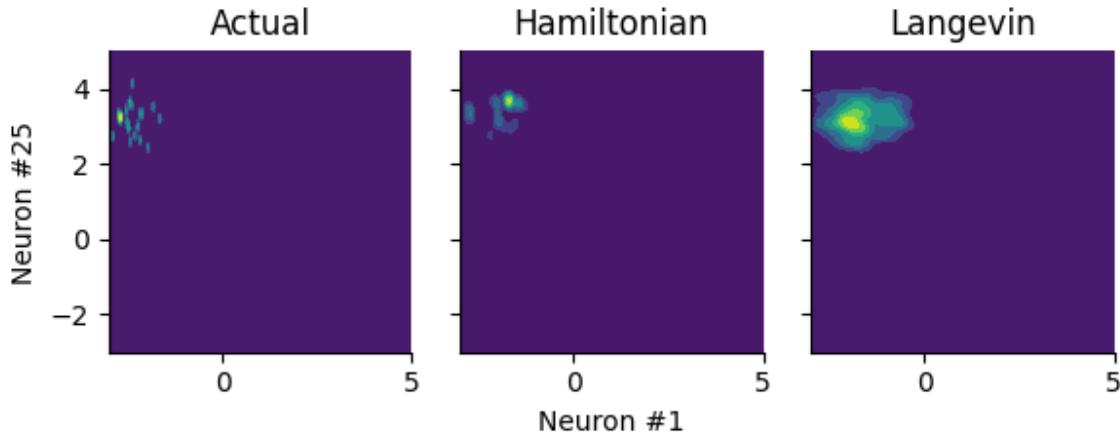


図 9.15 cell048.png

9.3.4 シナプスサンプリング

ここまでシナプス結合強度は変化せず、神経活動の変動によりサンプリングを行うというモデルについて考えてきた。一方で、シナプス結合強度自体が短時間で変動することによりベイズ推論を実行するというモデルがあり、**シナプスサンプリング (synaptic sampling)** と呼ばれる。ToDo: 以下の記述。(Kappel et al., 2015)(Aitchison et al., 2021)

9.4 確率的集団符号化 (probabilistic population coding)

Distributional Population Coding or distributed distributional codes (DDCs) ポアソン分布

$$P(X = k) = \frac{e^{-\lambda} \lambda^k}{k!} \quad (9.41)$$

より、

$$p(y | \mathbf{x}) \propto \prod_i \frac{e^{-f_i(y)} f_i(y)^{x_i}}{x_i!} p(y) \quad (9.42)$$

参考文献

Aitchison, L. and Lengyel, M. (2016). “The Hamiltonian Brain: Efficient Probabilistic Inference with Excitatory-Inhibitory Neural Circuit Dynamics”. *PLoS Comput. Biol.* 12.12, e1005186.

- Aitchison, L. et al. (2021). "Synaptic plasticity as Bayesian inference". *Nat. Neurosci.* 24.4, pp. 565–571.
- Berkes, P., Turner, R., and Fiser, J. (2011). "The Army of One (Sample): the Characteristics of Sampling-based Probabilistic Neural Representations". *Nature Precedings*, pp. 1–1.
- Buesing, L. et al. (2011). "Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons". *PLoS Comput. Biol.* 7.11, e1002211.
- Echeveste, R., Hennequin, G., and Lengyel, M. (2017). "Asymptotic scaling properties of the posterior mean and variance in the Gaussian scale mixture model". *arXiv preprint arXiv:1706.00925*.
- Echeveste, R. et al. (2020). "Cortical-like dynamics in recurrent circuits optimized for sampling-based probabilistic inference". *Nat. Neurosci.* 23.9, pp. 1138–1149.
- Kappel, D. et al. (2015). "Network Plasticity as Bayesian Inference". *PLoS Comput. Biol.* 11.11, e1004485.
- Masset, P. et al. (2022). "Population geometry enables fast sampling in spiking neural networks". *bioRxiv*, p. 2022.06.03.494680.
- Orbán, G. et al. (2016). "Neural Variability and Sampling-Based Probabilistic Representations in the Visual Cortex". *Neuron* 92.2, pp. 530–543.
- Wainwright, M. J. and Simoncelli, E. (1999). "Scale mixtures of Gaussians and the statistics of natural images". *Advances in Neural Information Processing Systems* 12.
- Zhang, W.-H. et al. (2022). "Sampling-based Bayesian inference in recurrent circuits of stochastic spiking neurons". *bioRxiv*, p. 2022.01.26.477877.