Julia で学ぶ計算論的神経科学

山本 拓都

2023 年 5 月 6 日

目次

第 1 章

局所学習則

1.1　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)** [?], [?] 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 [?] [?] 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** [?] [?] [?]　(ここでの 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 [?], [?] などのモデルがある．総説としては [?], [?] , 数理モデル同士の関係については [?] 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 [?]　(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現象があり，これを**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 [?] [?] [?] などがある．発生の数理モデルに関する総説 [?], [?]

1.1.1　単純なデータセット

SOM における n 番目の入力を $\mathbf{v}(t) = \mathbf{v}_n \in \mathbb{R}^D (n = 1, \ldots, N)$, m 番目のニューロン $(m = 1, \ldots, M)$ 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 [?]．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．

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

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

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 [**?**]．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．

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

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．

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

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")
```

- ToDo: dims を v, w で修正

```
# inputs
dims = 2  # dims of inputs
Random.seed!(1234);
σv, σw = 0.1, 0.05
N = 300 # num of inputs
num_blobs = 5
map_width = 15
M = map_width^2
init_w = σw*randn(M, 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
    x = vcat([σ*randn(n, dims) .+ [cos(i/num_blobs*2π), ↵
        sin(i/num_blobs*2π)]' for i in 0:num_blobs-1]...)
    y = repeat(1:num_blobs, inner=n)
    return x, y
end
```

```
v, v_labels = make_blobs(N, num_blobs, dims, σv);
```

```
function plot_som(v, w; vcolor="tab:blue", wcolor="tab:orange")
    M, dims = size(w)
    map_width = Int(sqrt(M))
    rw = reshape(w, (map_width, map_width, dims))
    scatter(v[:, 1], v[:, 2], s=10, color=vcolor)
    plot(rw[:, :, 1], rw[:, :, 2], "k", alpha=0.8); plot(rw[:, :, 1]', ↵
        rw[:, :, 2]', "k", alpha=0.8)
    scatter(w[:, 1], w[:, 2], s=10, color=wcolor) # 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．- ToDo: 変数名を修正

```
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)
    M = size(init_w)[1]
    map_width = Int(sqrt(M))
    N = 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:map_width for j in 1:map_width]...)
        dist_mat = hcat([sum((pos .- pos[:, i]) .^2, dims=1)' for i in ↵
            1:M]...); #'
    end

    @showprogress for t in 1:T
        α = α0 * (1 - t/T); # update rate
        σ = max(σ0 * (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 N inputs
        for i in 1:N
            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
    end
    if return_history
        return w_history
    else
        return w
    end
end;
```

```
w_history = SOM(v, init_w, α0=2, σ0=6, T=100);
```

青点が $\mathbf{v}$，オレンジの点が $\mathbf{w}$ 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 $\mathbf{w}$ 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);
```

```
figure(figsize=(6, 4))
idx = [1, 50, 100]
for i in 1:length(idx)
```

```
        wh = w_history[idx[i]]
        subplot(2,length(idx),i)
        title("Epoch : "*string(idx[i]))
        plot_som(v, wh, vcolor=vcolors); axis("off")
        subplot(2,length(idx),i+length(idx))
        imshow(reshape(wh[:, 1], (map_width, map_width))); axis("off")
    end
    tight_layout()
```

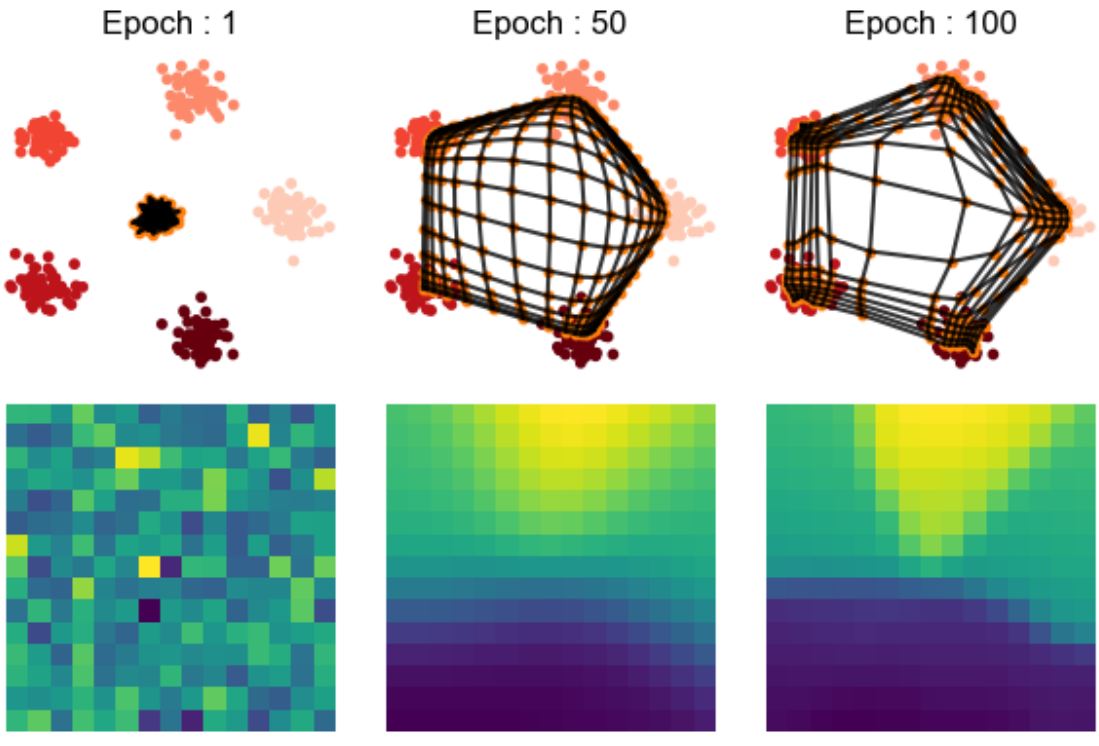

図 1.1 cell014.png

unified distance matrix を描画する．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．

```
    function Umatrix2d(w)
        M = size(w)[1]
        map_width = Int(sqrt(M))
        pos = hcat([[i, j] for i in 1:map_width for j in 1:map_width]...)
        abs_dist_mat = hcat([sum(abs.(pos .- pos[:, i]), dims=1)' for i in ↵
            1:M]...)
        adj_indices = [findall(x -> x == 1, abs_dist_mat[i, :]) for i in 1:M] # ↵
```

```
        adjacent indices
    U = [sqrt(sum((w[adj_indices[i], :] .- w[i, :]') .^2) / ↵
        size(adj_indices[i])[1]) for i in 1:M]
    U = reshape(U, (map_width, map_width));
    return U
end
```

```
# find best matching unit
function find_bmu(v, w)
    N = size(v)[1]
    dims = size(w)[2]
    pos = hcat([[i, j] for i in 1:map_width for j in 1:map_width]...)
    mapped_vpos = zeros(N, dims);
    for i in 1:N
        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 = Umatrix2d(w_history[end]);
```

```
mapped_vpos = find_bmu(v, w_history[end]);
```

複数の点が同じ位置に重なっていることに注意.

```
figure(figsize=(3,3))
title(L"$U$-matrix")
imshow(U, interpolation="bicubic")
scatter(mapped_vpos[:, 1], mapped_vpos[:, 2], color=vcolors, s=10)
tight_layout()
```

1.1.2 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.

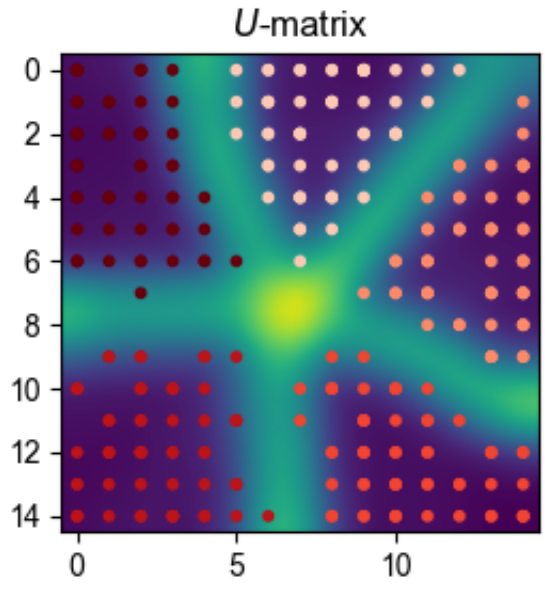

図 1.2 cell021.png

```
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 [?] を参考に作成．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, 0.2

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+1)[1:end-1]

# stimuli
v = product(rx, ry, rOD, rORθ, r)
rORxy = hcat(pol2cart.(2v[:, 4], v[:, 5])...)
v[:, 4], v[:, 5] = rORxy[1, :], rORxy[2, :];
v += (rand(size(v)...) .- 1) * 1e-5;
```

```
# initial neurons
map_width = 64
M = map_width^2
init_w = product(range(0, 1, length=map_width), range(0, 1, ↵
    length=map_width))
init_w += (rand(size(init_w)...) .- 1) * 0.05;
init_w = [init_w 2l*(rand(M) .- 0.5) hcat(pol2cart.(4π*(rand(M) .- 0.5), ↵
    r*rand(M))...)']
#w = reshape(w, (map_width, map_width, dims));
```

```
w = SOM(v, init_w, α0=1.5, σ0=5.0, T=50, return_history=false);
```

描画用関数を実装する．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が，これは読者への課題とする．

```
function plot_visual_maps(v, w)
    figure(figsize=(7, 6))
    subplot(2,2,1, adjustable="box", aspect=1); title("Retinotopic map")
    plot_som(v, w)

    M, dims = size(w)
    map_width = Int(sqrt(M))
    rw = reshape(w, (map_width, map_width, dims))

    ax1 = subplot(2,2,2, adjustable="box", aspect=1); title("Ocular ↵
        dominance (OD) map")
    imshow(rw[:, :, 3], cmap="gray", origin="lower")

    ins1 = ax1.inset_axes([1.05,0,0.05,1])
    colorbar(cax=ins1, aspect=40, pad=0.08, shrink=0.6)
    ins1.text(0, -0.15, "Left", ha="center", va="center")
    ins1.text(0, 0.15, "Right", ha="center", va="center")

    subplot(2,2,3, adjustable="box", aspect=1); title("Contours of OD and ↵
        OR")
    ORmap = atan.(rw[:, :, 5], rw[:, :, 4]); # get angle of polar
    sizex, sizey = map_width, map_width
    x, y = 0:sizex-1, 0:sizey-1
    X, Y = ones(sizey) * x', y * ones(sizex)';
    contour(X, Y, ORmap, cmap="hsv")
    contour(X, Y, rw[:, :, 3], colors="k", levels=1)

    ax2 = subplot(2,2,4, adjustable="box", aspect=1); title("Orientation ↵
```

```
        (OR) angle map")
    imshow(ORmap, cmap="hsv", origin="lower")

    cm = get_cmap(:hsv)
    lines, colors = [], []
    for i in 1:9
        θ = (i-1)/8*π
        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 = ax2.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()
end;
```

```
plot_visual_maps(v, w)
```

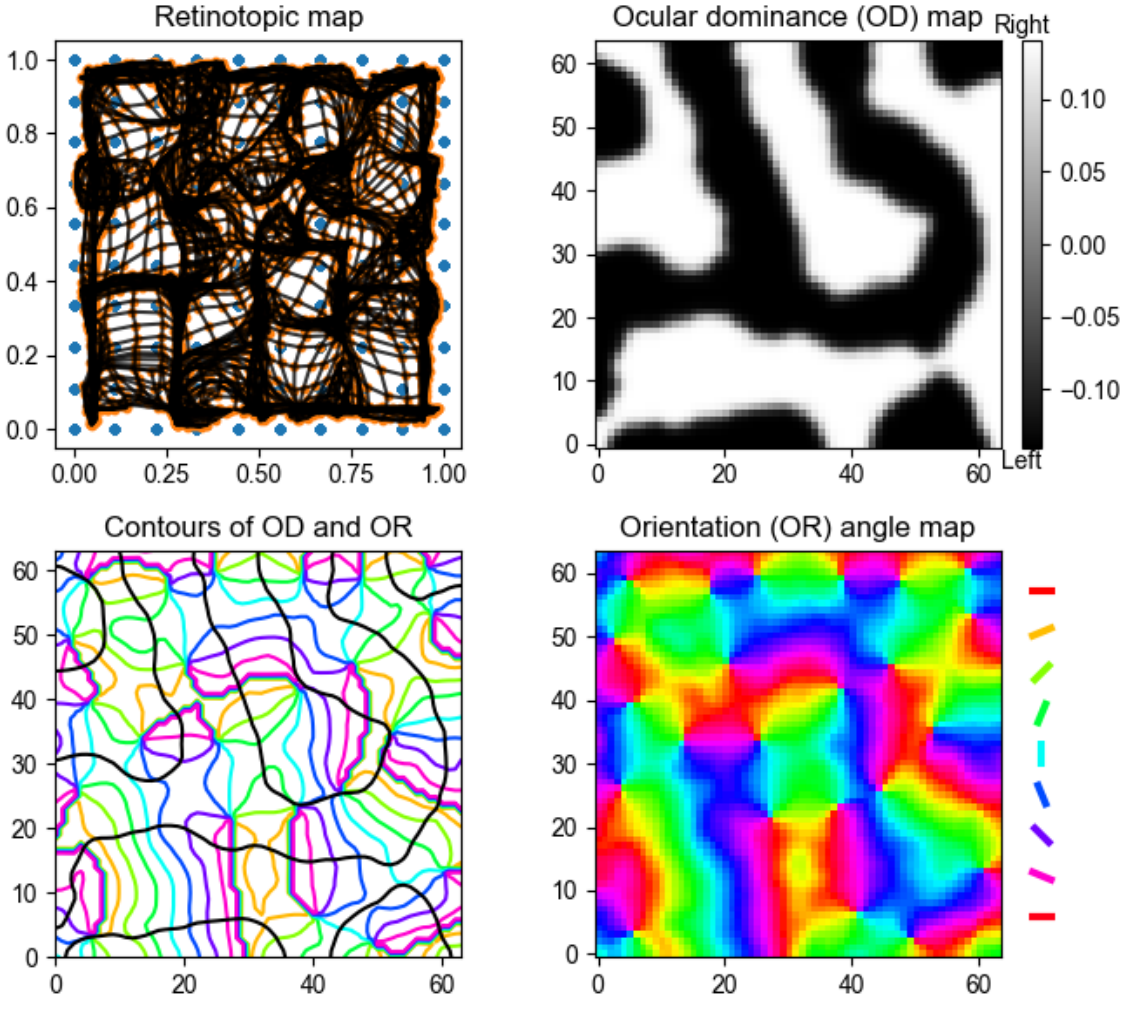

図 1.3 cell030.png

第 2 章

神経回路網によるベイズ推論

2.1 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

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

より,

$$p(y \mid \mathbf{x}) \propto \prod_i \frac{e^{-f_i(y)} f_i(y)^{x_i}}{x_i!} p(y)$$

2.2 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.[*1] データを X とし，パラメータを θ とする.

$$p(\theta \mid X) = \frac{p(X \mid \theta)p(\theta)}{\int p(X \mid \theta)p(\theta)d\theta}$$

分母の積分計算 $\int p(X \mid \theta)p(\theta)d\theta$ が求まればよい.

[*1] 変分推論は入れた方がいいと思うが，紙幅の都合上いれられるか微妙である.

モンテカルロ法

マルコフ連鎖

2.2.1 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 method; log_p: unnormalized log-posterior
function GaussianMH(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
        mH_ = log_p(θ_) + logpdf(MvNormal(θ_, σ*I), θ)    # final Hamiltonian
```

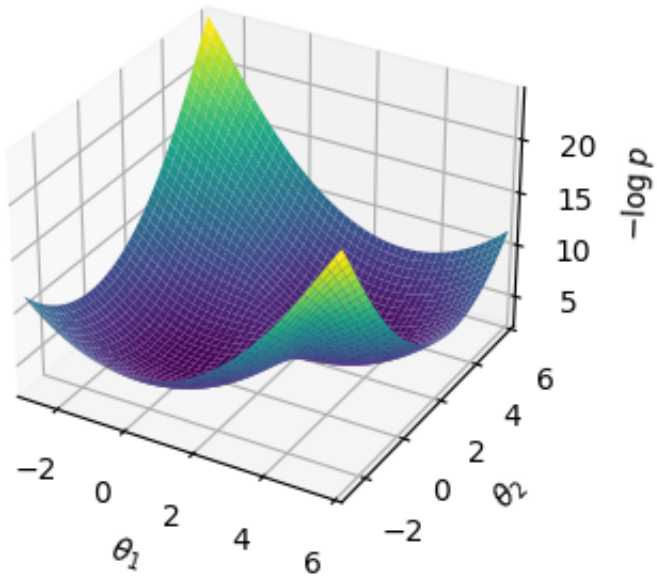

図 2.1 cell004.png

```
        if min(1, exp(mH_ - mH)) > rand()
            θ = θ_ # accept
            num_accepted += 1
        end
        samples[:, m] = θ
    end
    return samples, num_accepted
end;
```

```
log_p(θ) = logpdf(mixed_gauss, θ);
grad(θ)= ForwardDiff.gradient(log_p, θ)
```

```
θm, num_accepted = GaussianMH(log_p, [1.0,0.5], 1.0, 2000)
```

```
size(θm)
```

```
Um = kde((θm[1, :], θm[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()
```

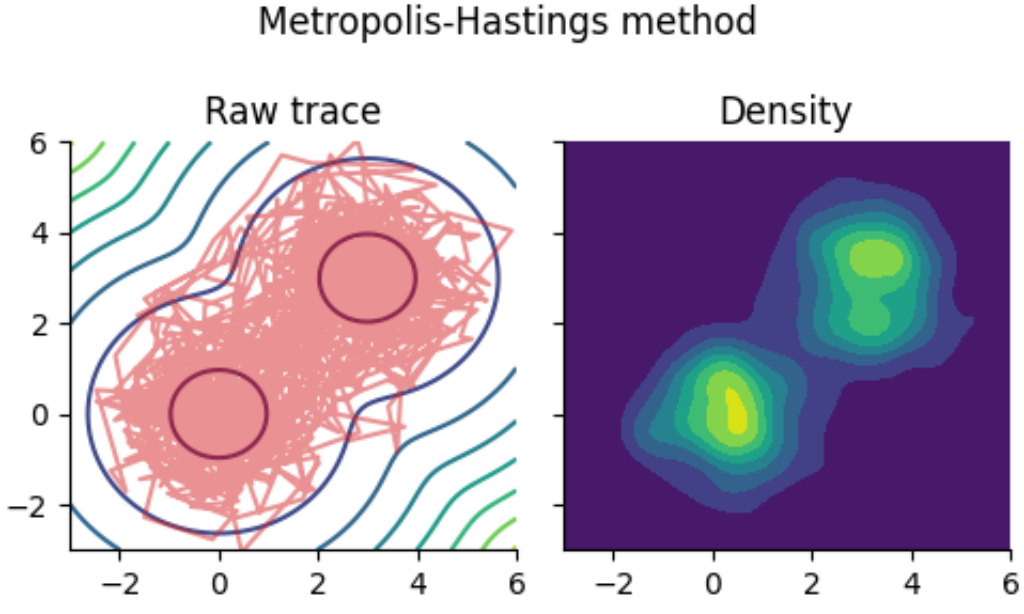

図 2.2 cell009.png

2.2.2 ランジュバン・モンテカルロ法 (LMC)

拡散過程

$$\frac{d\theta}{dt} = \nabla \log p(\theta) + \sqrt{2} dW$$

Euler – Maruyama 法により,

```
β = 1
ρ = sqrt(2*ϵ);
```

```
nt = 10000
ϵ = 0.1
```

```
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()
```

2.2.3 ハミルトニアン・モンテカルロ法 (HMC 法)

ハミルトニアン・モンテカルロ法 (Hamiltonian Monte Calro) あるいはハイブリッド・モンテカルロ法 (Hybrid Monte Calro) という一般化座標を $\mathbf{q}$, 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

$$\mathcal{H}(\mathbf{q},\mathbf{p}) := U(\mathbf{q}) + \frac{1}{2}\|\mathbf{p}\|^2$$

となる．このとき，次の 2 つの方程式が成り立つ．

$$\frac{d\mathbf{q}}{dt} = \frac{\partial \mathcal{H}}{\partial \mathbf{p}} = \mathbf{p}, \quad \frac{d\mathbf{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{q}} = -\frac{\partial U}{\partial \mathbf{q}}$$

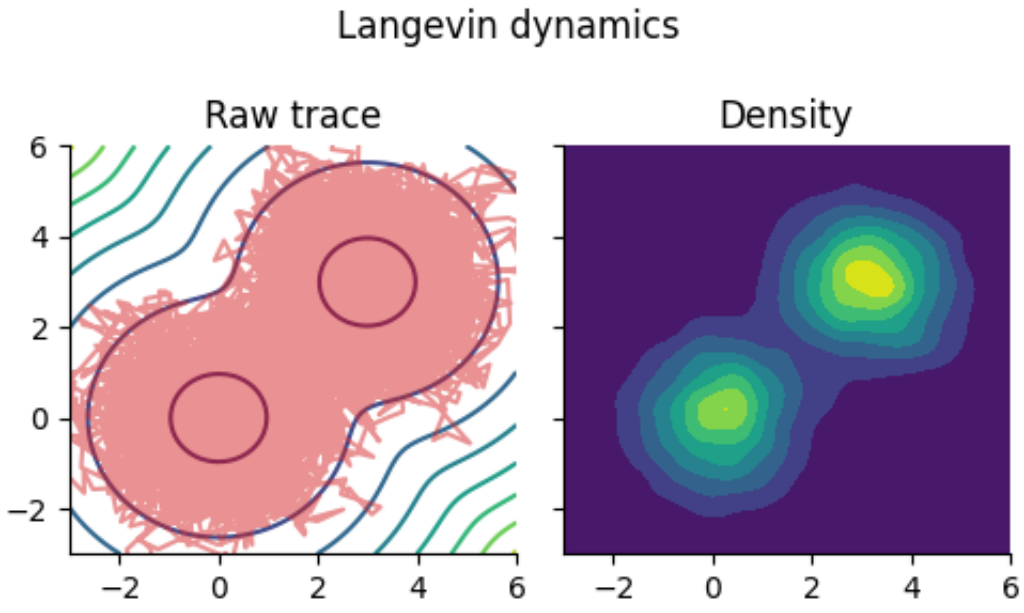

図 2.3　cell014.png

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．この処理を Metropolis-Hastings 法における採用・不採用アルゴリズムという．リープフロッグ (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
    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()
```

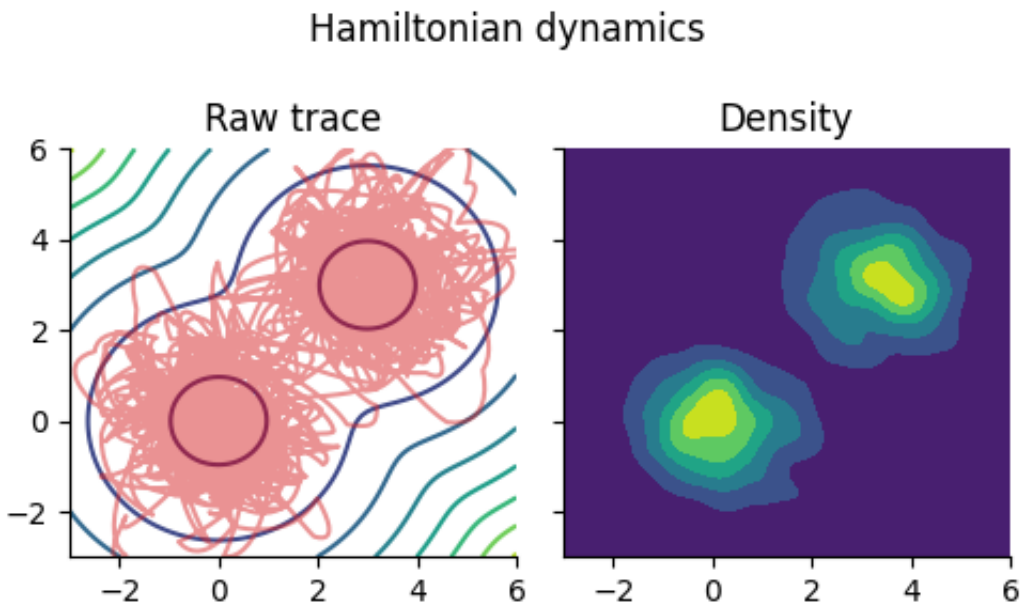

図 2.4 cell020.png

ToDo: 自己相関確認する

2.2.4 線形回帰への適応

```
# Generate Toy datas
num_train, num_test = 20, 100 # sample size
dims = 4 # dimensions
σy = 0.3

polynomial_expansion(x; degree=3) = hcat([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);
```

```
log_joint(w, ϕ, y, σy, μ₀, Σ₀) = sum(logpdf.(Normal.(ϕ * w, σy), y)) + ↵
    logpdf(MvNormal(μ₀, Σ₀), w);
```

```
α, β = 1e-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, :])
```

```

```

```
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()
```

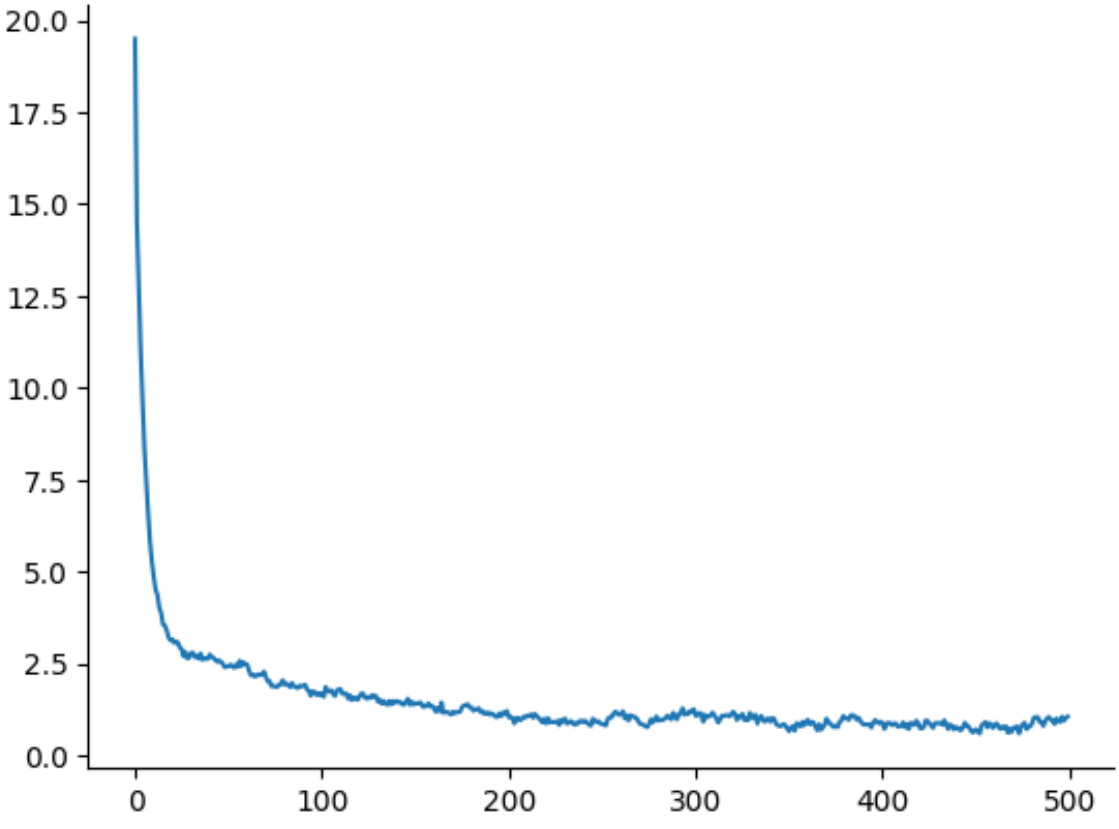

図 2.5 cell030.png

2.3 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.

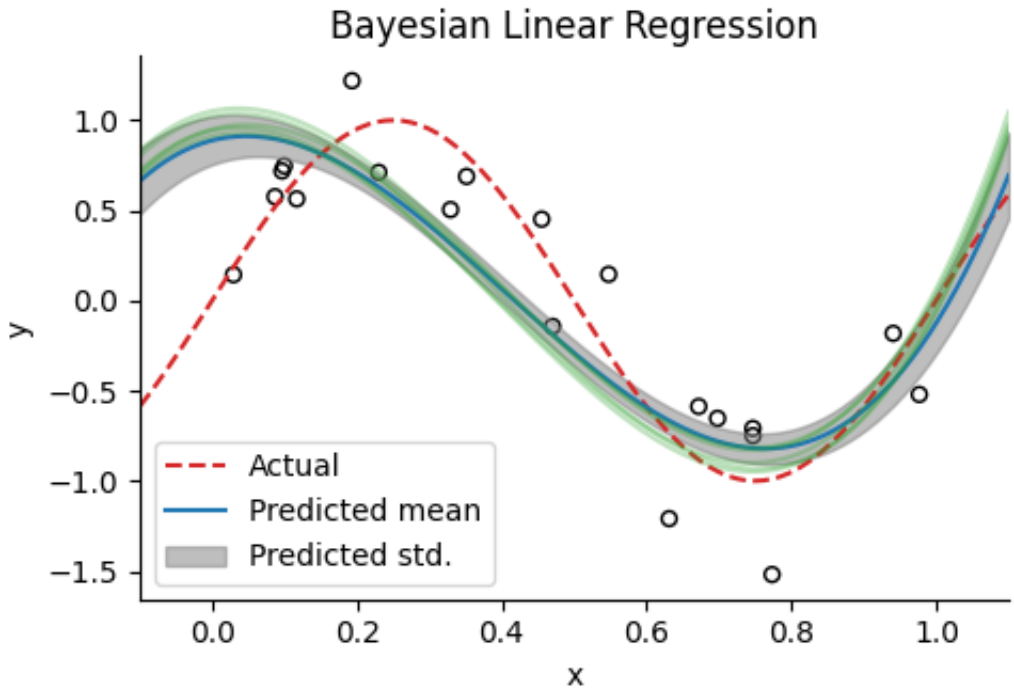

図 2.6 cell033.png

2.3.1 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 [?][?]. GSM モデルでは入力を次式で予測する：

$$\text{入力} = z\left(\sum \text{神経活動} \times \text{基底}\right) + \text{ノイズ}$$

前節までのスパ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 が基底の線形和に乗じられている点が異なる．[*2]

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．

$$p\left(\mathbf{x} \mid \mathbf{y}, z\right) = \mathcal{N}\left(z\mathbf{A}\mathbf{y}, \sigma_{\mathbf{x}}^2\mathbf{I}\right)$$

[*2] コードは [?] https://github.com/gergoorban/sampling_in_gsm, および [?] https://bitbucket.org/RSE_1987/ssn_inference_numerical_experiments/src/master/ を参考に作成した．

事前分布を

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{C})$$
$$p(z) = \Gamma(k, \vartheta)$$

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 A の作成

```
using PyPlot, LinearAlgebra, Random, Distributions, KernelDensity, StatsBase
using PyPlot: matplotlib
Random.seed!(2)
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;
```

```
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)
```

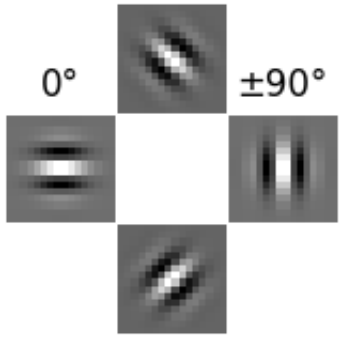

図 2.7 cell007.png

分散共分散行列 C の作成

C は y の事前分布の分散共分散行列である．[?] 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 **A** と同様に [?] に従って作成する．前項で作成した通り，**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 **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)

$$K(\theta, \theta') = \exp\left[\phi_1 \cos\left(\frac{|\theta - \theta'|}{\phi_2}\right)\right]$$

を用いる．ここでは $|\theta - \theta'| = m\pi$ $(m = 0, 1, \ldots)$ 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 $\epsilon\mathbf{I}$ 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()
```

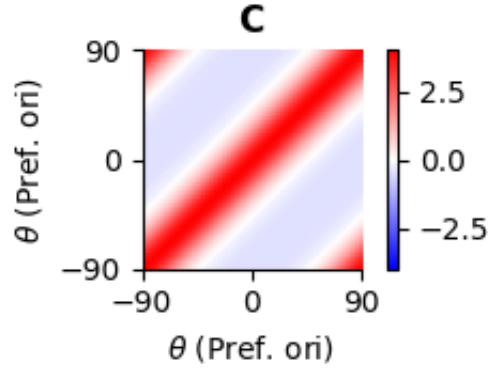

図 2.8　cell010.png

ここで Pref. ori は最適方位 (preferred orientation) を意味する．

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．

$$p(z \mid \mathbf{x}) \propto p(z)\mathcal{N}\left(0, z^2\mathbf{A}\mathbf{C}\mathbf{A}^\top + \sigma_x^2\mathbf{I}\right)$$
$$p(\mathbf{y} \mid z, \mathbf{x}) = \mathcal{N}\left(\mu(z, \mathbf{x}), \Sigma(z)\right)$$

ただし，

$$\Sigma(z) = \left(\mathbf{C}^{-1} + \frac{z^2}{\sigma_x^2}\mathbf{A}^\top\mathbf{A}\right)^{-1}$$
$$\mu(z, \mathbf{x}) = \frac{z}{\sigma_x^2}\Sigma(z)\mathbf{A}^\top\mathbf{x}$$

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，$p(\mathbf{y} \mid z, \mathbf{x})$ 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．

$$p(\mathbf{y} \mid \mathbf{x}) = \int dz\ p(z \mid \mathbf{x}) \cdot p(\mathbf{y} \mid z, \mathbf{x})$$

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．

$$z_{\mathrm{MAP}} = \underset{z}{\mathrm{argmax}} p(z \mid \mathbf{x})$$

次に z_{MAP} の周辺で $p(z \mid \mathbf{x})$ 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 [?] 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，$p(z \mid \mathbf{x}) \simeq \delta(z - z_{\mathrm{MAP}})$ とすることを提案している．この場合，z は定数とみなせ，$p(\mathbf{y} \mid \mathbf{x}) \simeq p(\mathbf{y} \mid \mathbf{x}, z = z_{\mathrm{MAP}})$ 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 $\mathbf{x}$ とすると，$\mathbf{x} = z^*\mathbf{x}$ 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，$p(\mathbf{y} \mid \mathbf{x}) \simeq p(\mathbf{y} \mid \mathbf{x}, z = z^*)$ 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，$z = z_{\mathrm{MAP}}$ とする手法を用いることにする．

```
# log pdf of p(z)
log_Pz(z, k, θ) = logpdf.(Gamma(k, θ), z)

# pdf of p(z|x)
function Pz_x(z_range, x, ACAᵀ, σₓ², k, θ)
    n_contrasts = length(z_range)
    log_p = zeros(n_contrasts)
    μxz = zeros(size(x))
    dz = z_range[2] - z_range[1]
    for i in 1:n_contrasts
```

```
        Cxz = z_range[i]^2 * ACAᵀ + σₓ² * I
        log_p[i] = log_Pz(z_range[i], k, θ) + logpdf(MvNormal(μxz, ↵
            Symmetric(Cxz)), 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, σₓ², A, AᵀA, C⁻¹)
    Σz = inv(C⁻¹ + (z^2 / σₓ²) * AᵀA)
    μzx = (z/σₓ²) * Σz * A' * x
    return μzx, Σz
end;
```

シミュレーション

```
AᵀA = A' * A
ACAᵀ = A * C * A'

σₓ = 1.0 # Noise of the x process
σₓ² = σₓ^2
k, θ = 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 = hcat([rand(MvNormal(vec(z*A*y), σₓ*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()
```

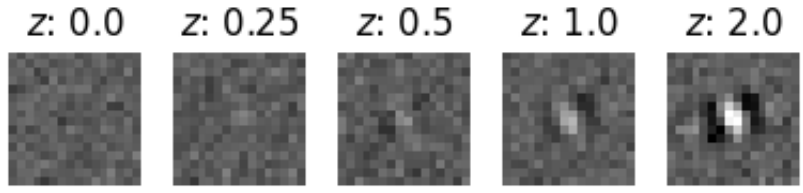

図 2.9 cell019.png

事後分布の計算をする.

```
μ_post = zeros(n_samples, Ny)
σ_post = zeros(n_samples, Ny)
Σ_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', σₓ², k, θ)
    Z_MAP[s] = z_range[argmax(p_z)] # MAP estimated z
    μ_post[s, :], Σ_post[s, :, :] = post_moments(X[s, :], Z_MAP[s], σₓ², A, ↵
        A'A, C⁻¹)
    σ_post[s, :] = sqrt.(diag(Σ_post[s, :, :]))
end
```

結果

```
θs = 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()
```

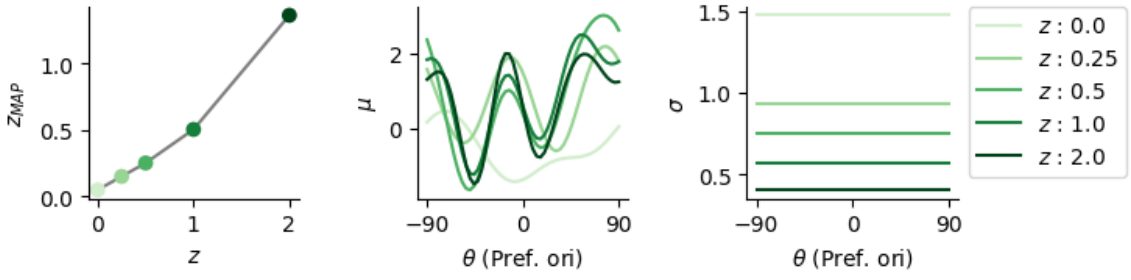

図 2.10 cell023.png

```
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

```
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.

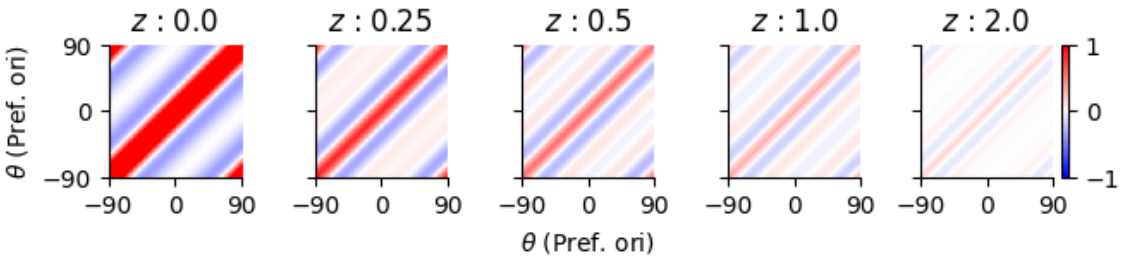

図 2.11 cell024.png

```
nt = 1000
h_gsm = zeros(n_samples, Ny, nt)
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(mea
    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()
```

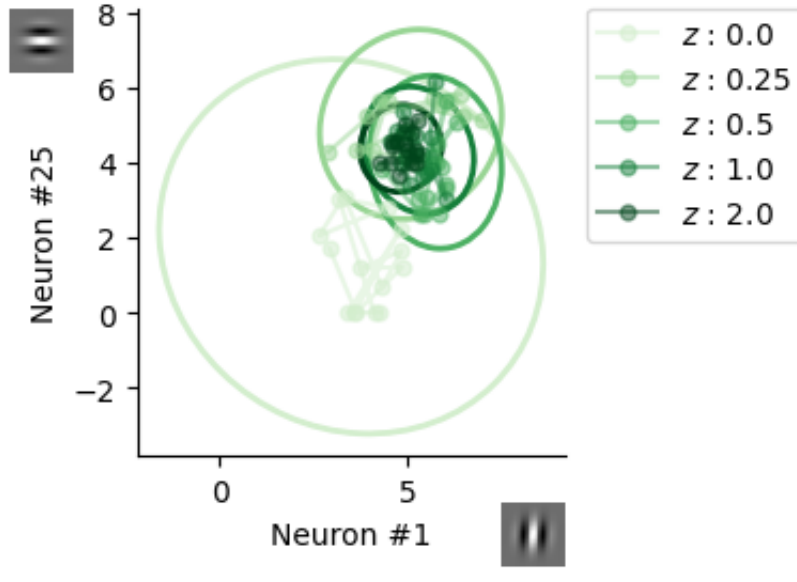

図 2.12 cell030.png

2.3.2 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．Hamiltonian を用いる場合，一般化座標 **q** を興奮性神経細胞の活動 **u**, 一般化運動量 **p** を抑制性神経細胞の活動 **v** に対応させる．*ToDo: 詳しい説明* 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 ∇ᵤlogP(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) = ∇ᵤlogP(u, x, σₓ², A, C⁻¹);
```

```
function NeuralLMC(∇log_p::Function, u_init::Vector{Float64}, α::Float64, ↵
    ρ::Float64, dt::Float64, nt::Int)
    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 NeuralHMC(∇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)
        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 = NeuralLMC(∇log_p, u_init, α_ext[1], ρ, dt, nt);
```

```
@time u_nhmc, v_nhmc = NeuralHMC(∇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(L"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()
```

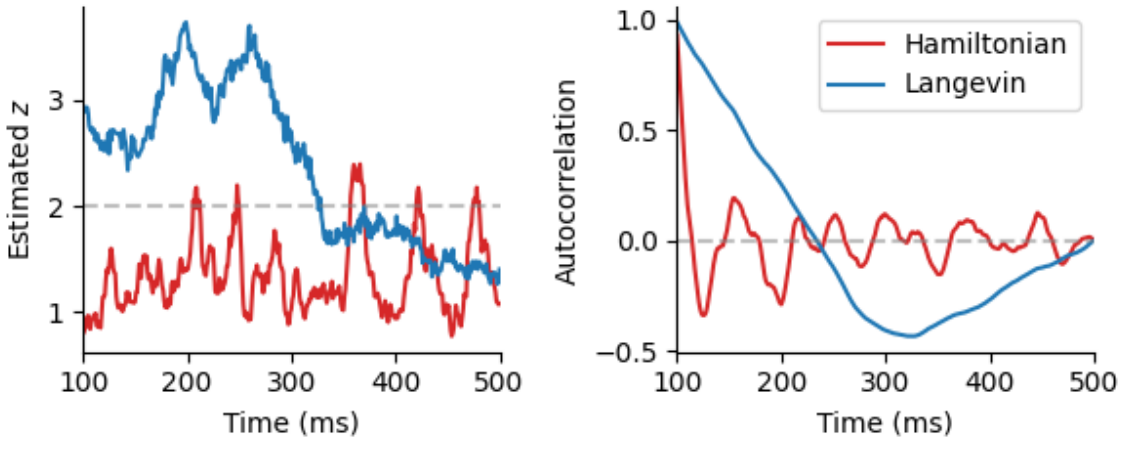

図 2.13 cell042.png

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, :])

kde_bound = ((3,7),(3,7)) # ((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()
```

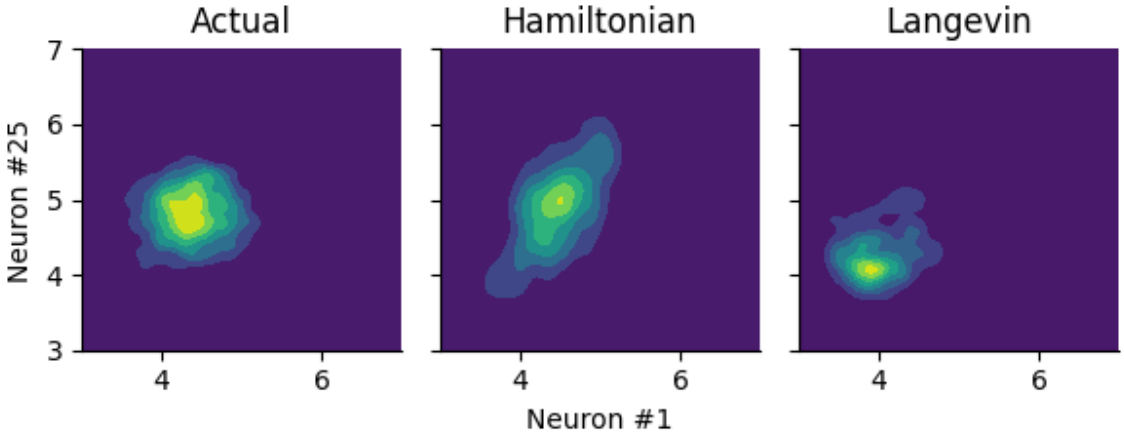

図 2.14 cell045.png

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，[?] 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 [?]．burn-in がなくなり効率良くサンプリングできる．

2.3.3 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* - [?] - [?] - [?]

2.3.4 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* - [?] - [?]