Julia で学ぶ計算論的神経科学

山本 拓都

2023 年 7 月 10 日

目次

第 1 章

はじめに

第 2 章

神経細胞のモデル

第 3 章

シナプス伝達のモデル

第 4 章

神経回路網の演算処理

第 5 章

局所学習則

5.1 ロジスティック回帰とパーセプトロン

logistic regression, perceptron 1 層パーセプトロン https://www.cs.utexas.edu/~gdurrett/courses/fa2022/perc-lr-connections.pdf add SyntheticDatasets よりも正規分布 ×2 の方がいいか https://en.wikipedia.org/wiki/Perceptron https://arxiv.org/abs/2012.03642

perceptron は 0/1 or -1/1 のどちらか

UNDERSTANDING STRAIGHT-THROUGH ESTIMATOR IN TRAINING ACTIVATION QUANTIZED NEURAL NETS

Yoshua Bengio, Nicholas L´eonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv preprint arXiv:1308.3432, 2013.

Hinton (2012) in his lecture 15b

G. Hinton. Neural networks for machine learning, 2012. https://www.cs.toronto.edu/~hinton/coursera_lectures.html

delta rule

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

```
N = 400 # num of inputs
dims = 2  # dims of inputs
Random.seed!(1234);
```

```
X = [randn(Int(N/2), dims);  3.0 .+ randn(Int(N/2), dims)];
y = [zeros(Int(N/2)); ones(Int(N/2))];
```

```
figure(figsize=(4, 4))
scatter(X[y.==0, 1], X[y.==0, 2])
scatter(X[y.==1, 1], X[y.==1, 2])
xlabel(L"$x_1$"); ylabel(L"$x_2$");
tight_layout()
```

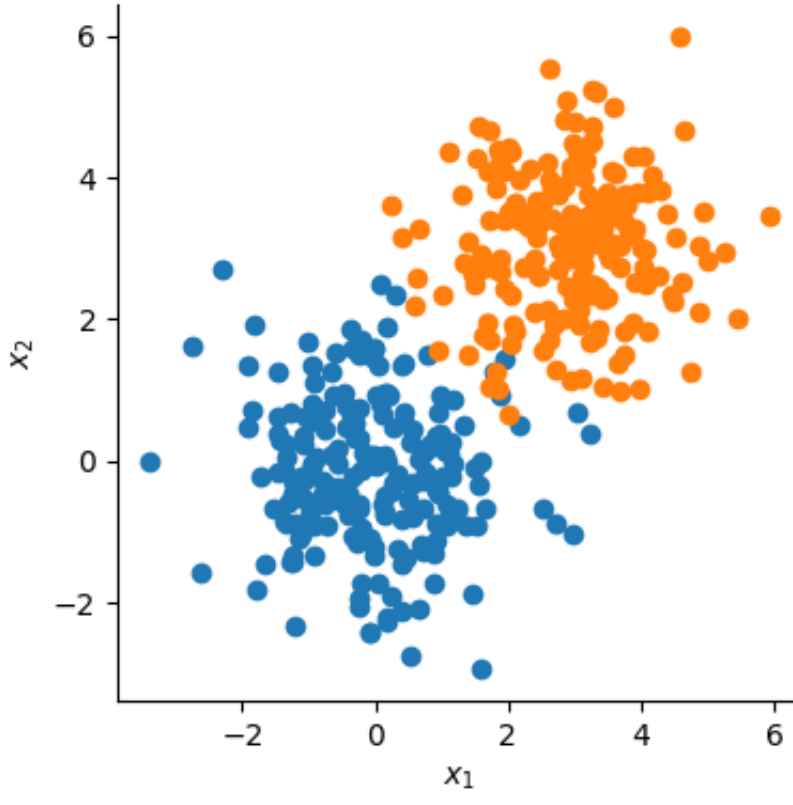

図 5.1 cell006.png

```
m, n = 2, 1
```

```
step(x) = 1.0(x > 0)
```

```
x = -1:0.01:1
figure(figsize=(3, 2))
plot(x, step.(x))
tight_layout()
```

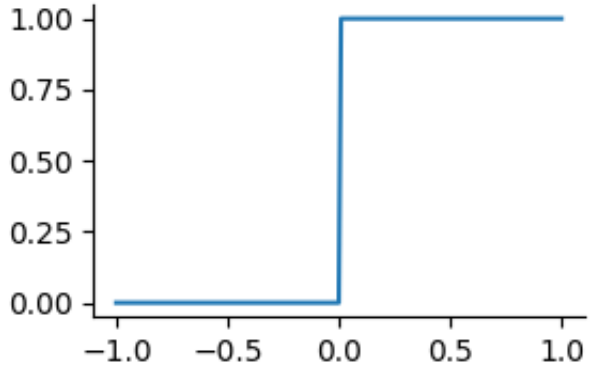

図 5.2 cell009.png

Here denotes the (point-wise) activation function, $W \in R^{m \times n}$ is the weight-matrix and $b \in R^n$ is the bias-vector. The vector $x \in R^m$ and the vector $y \in R^n$ denote the input, respectively the output

$$y = \sigma(W^\top x + b) \tag{5.1}$$

$$\text{Initialize } W^0, b^0; \tag{5.2}$$

$$\text{for } k = 1, 2, \dots \text{ do} \tag{5.3}$$

$$\left|\begin{array}{l} \text{for } i = 1, \dots, s \text{ do} \\ e_i = y_i - \sigma\left(\left(W^k\right)^\top x_i + b^k\right) \\ W^{k+1} = W^k + e_i x_i^\top \\ b^{k+1} = b^k + e_i \end{array}\right. \tag{5.4}$$

$$\text{end} \tag{5.5}$$

```
n_iter = 100
loss = zeros(n_iter);
W = randn(n, m)
b = randn(n)
for t in 1:n_iter
```

```
    y^ = step.(W * X' .+ b)'
    e = y - y^
    loss[t] = sum(e.^2)
    W[:, :] += 0.1*(1/200) * e' * X
    b[:] .+= 0.1*(1/200) * sum(e)
end
```

```
figure(figsize=(3,2))
plot(loss)
xlabel("Iteration"); ylabel("Loss")
tight_layout()
```

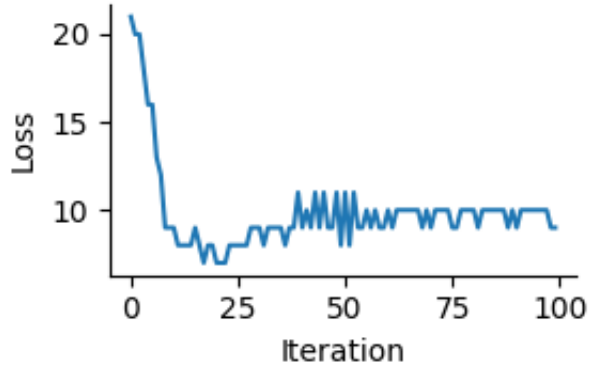

図 5.3 cell012.png

```
y^ = step.(W * X' .+ b)'; # prediction
```

```
p1 = X[y^[:, 1] .== 0, :]
p2 = X[y^[:, 1] .== 1, :];
```

ax + by + c = 0 y = -a/b x - c/b

```
xx = -3:0.01:6
yy = -W[1]/W[2]*xx .- b / W[2];
```

```
figure(figsize=(4, 4))
scatter(p1[:, 1], p1[:, 2])
```

```
scatter(p2[:, 1], p2[:, 2])
plot(xx, yy, color="k")
xlabel(L"$x_1$"); ylabel(L"$x_2$");
tight_layout()
```

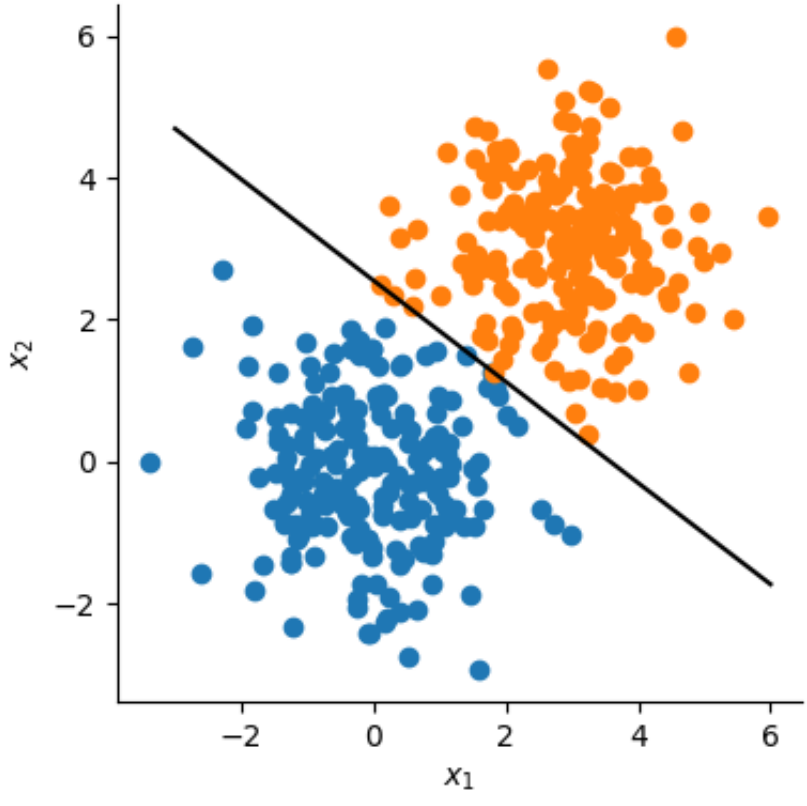

図 5.4 cell017.png

5.2 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 [?], [?]

5.2.1 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．

$$[\text{ユークリッド距離}] : c = \underset{m}{\operatorname{argmin}} \left[\|\mathbf{v}(t) - \mathbf{w}_m(t)\|^2 \right] \tag{5.6}$$

$$[\text{コサイン類似度}] : c = \underset{m}{\operatorname{argmax}} \left[\frac{\mathbf{w}_m(t)^\top \mathbf{v}(t)}{\|\mathbf{w}_m(t)\| \|\mathbf{v}(t)\|} \right] \tag{5.7}$$

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．

$$\mathbf{w}_m(t+1) = \mathbf{w}_m(t) + h_{cm}(t)[\mathbf{v}(t) - \mathbf{w}_m(t)] \tag{5.8}$$

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．

$$h_{cm}(t) = \alpha(t) \exp\left(-\frac{\|\mathbf{x}_c - \mathbf{x}_m\|^2}{2\sigma^2(t)}\right) \tag{5.9}$$

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
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);
```

```
# 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; vcolor="tab:blue")
    num_w, dims = size(w)
```

[*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．

```
    num_w_sqrt = Int(sqrt(num_w))
    rw = reshape(w, (num_w_sqrt, num_w_sqrt, dims))
    scatter(v[:, 1], v[:, 2], s=10, color=vcolor)
    plot(rw[:, :, 1], rw[:, :, 2], "k", alpha=0.5);
    plot(rw[:, :, 1]', rw[:, :, 2]', "k", alpha=0.5)
    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(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 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
    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(σ0 * (1 - t/T), 1); # decay from large to small (linearly ↵
            decreased, avoid zero)
        wm = ceil(Int, σ)
        h = gaussian_mask(2wm+1, 2wm+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] - wm):min(num_w_sqrt, win_idx[j] + wm) ↵
                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);
```

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]))

        if i == 1
            ylabel("Weight unfolding\n in data space")
        end
        plot_som(v, wh, vcolor=vcolors);
        subplot(2,length(idx),i+length(idx))

        if i == 1
            ylabel("1st dim. weight")
        end
        imshow(reshape(wh[:, 1], (num_w_sqrt, num_w_sqrt)));
    end
    tight_layout()
```

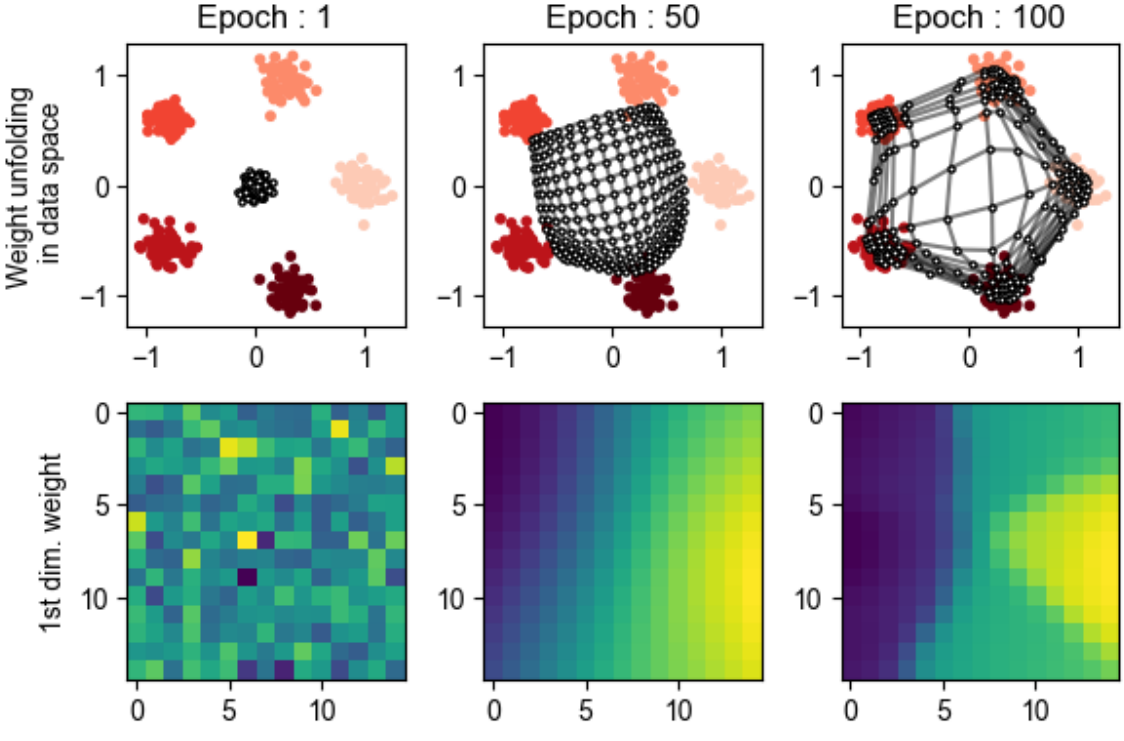

図 5.5 cell017.png

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 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]);
```

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()
```

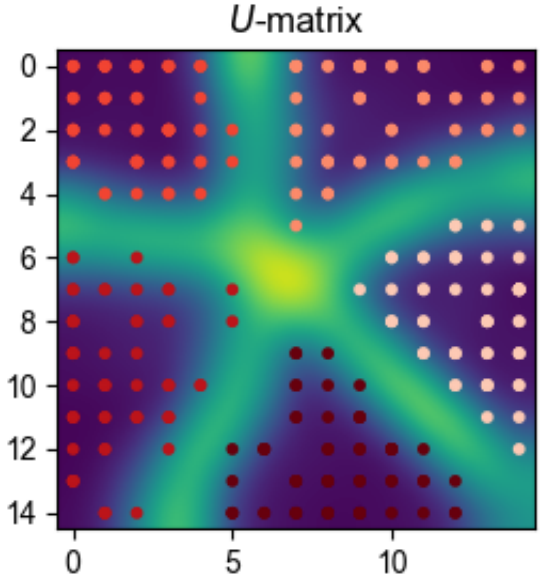

図 5.6 cell024.png

5.2.2 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.

```
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
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 2l*(rand(num_w) .- 0.5) hcat(pol2cart.(4π*(rand(num_w) .- ↵
    0.5), r*rand(num_w))...)']
#w = reshape(w, (num_w_sqrt, num_w_sqrt, dims));
```

```
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);
```

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)

    num_w, dims = size(w)
    num_w_sqrt = Int(sqrt(num_w))
    rw = reshape(w, (num_w_sqrt, num_w_sqrt, 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, -0.16, "Left", ha="left", va="center")
    ins1.text(0, 0.16, "Right", ha="left", 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 = num_w_sqrt, num_w_sqrt
    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)
```

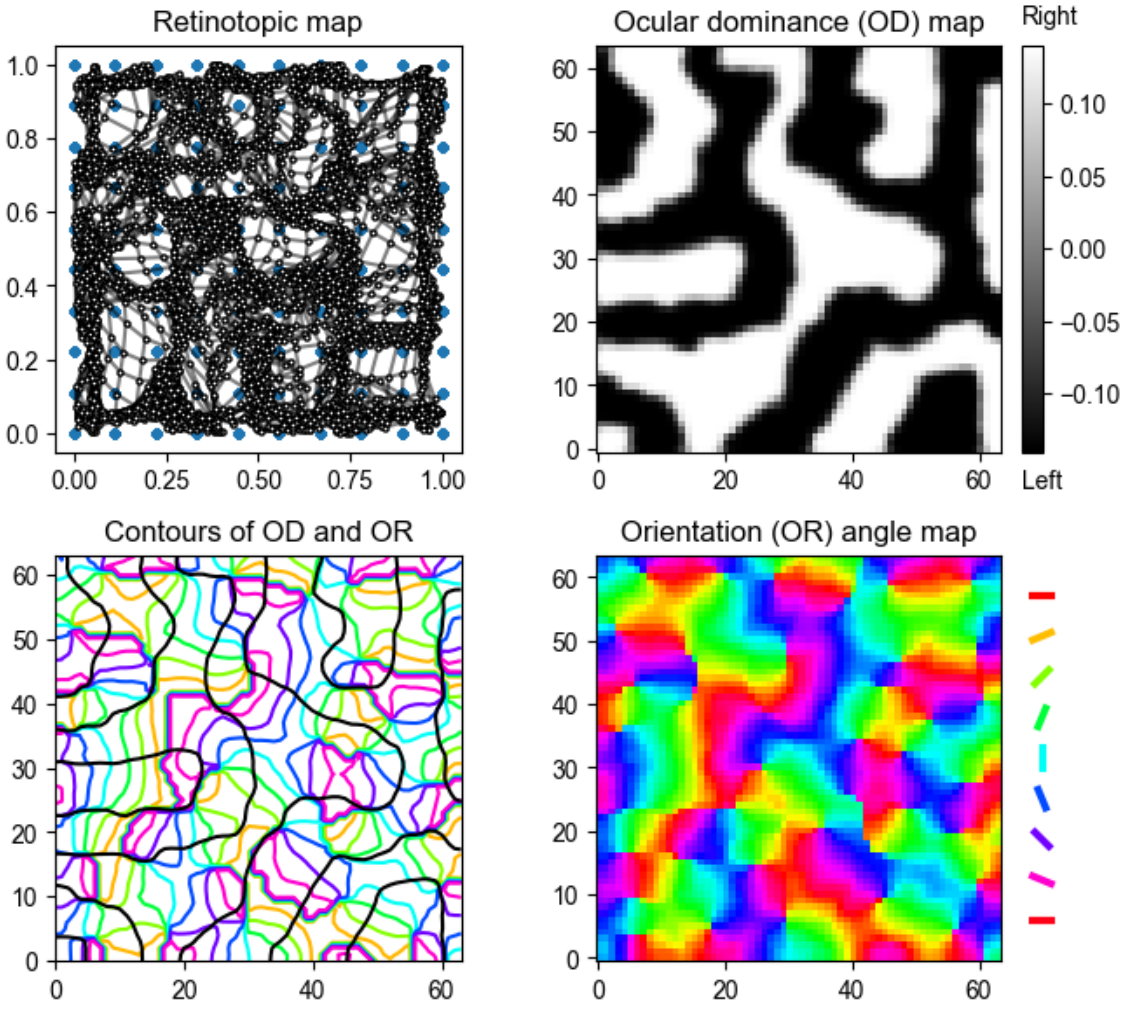

図 5.7 cell033.png

第 6 章

生成モデルとエネルギーベースモデル

第 7 章

貢献度分配問題の解決策

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; [**?**]) 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，問題は次のようになる．

$$\text{Minimize} \quad \frac{1}{2}\mathbf{x}^\top \mathbf{P}\mathbf{x} + \mathbf{q}^\top \mathbf{x} \tag{8.1}$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \tag{8.2}$$

Lagrange の未定乗数法を用いると解は

$$\begin{bmatrix} \mathbf{P} & \mathbf{A}^\top \\ \mathbf{A} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{q} \\ \mathbf{b} \end{bmatrix} \tag{8.3}$$

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 \ 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 [?] 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にする．

$$\underset{u(t)}{\text{minimize}} \quad \|u(t)\|_2 \tag{8.4}$$

$$\text{subject to} \quad \dot{\mathbf{x}}(t) = A\mathbf{x}(t) + Bu(t) \tag{8.5}$$

ただし，$\|\cdot\|_2$ は L_2 ノルムを意味し，$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \mathbf{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix}, u(t) = \dddot{x}(t)$ とする．すなわち，制御信号 $u(t)$ は躍度 $\dddot{x}(t)$ 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), ↵
    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()
```

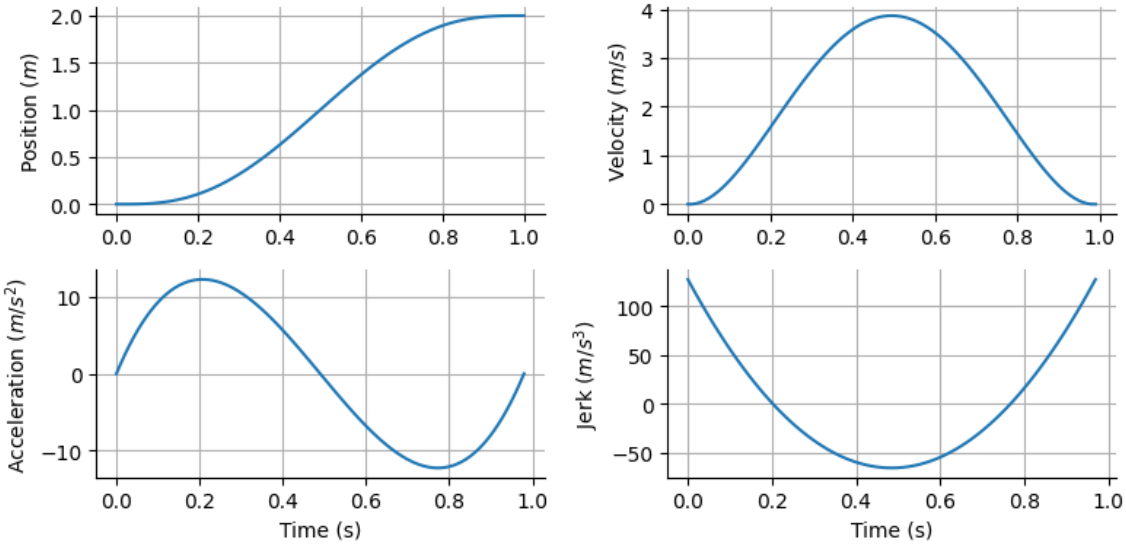

図 8.1 cell015.png

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     # inital 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()
```

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; [**?**]) を実装する．

$\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{B} \in \mathbb{R}^n$ とする．$\dot{x} = \mathbf{A}_c\mathbf{x} + \mathbf{B}_c(u + w)$ について，差分化すると

$$\mathbf{x}(t + dt) = \mathbf{x}(t) + \dot{\mathbf{x}}dt \tag{8.6}$$

$$\mathbf{x}_{t+1} = \mathbf{I}\mathbf{x}_t + (\mathbf{A}_c dt)\mathbf{x}_t + (\mathbf{B}_c dt)(u + w) \tag{8.7}$$

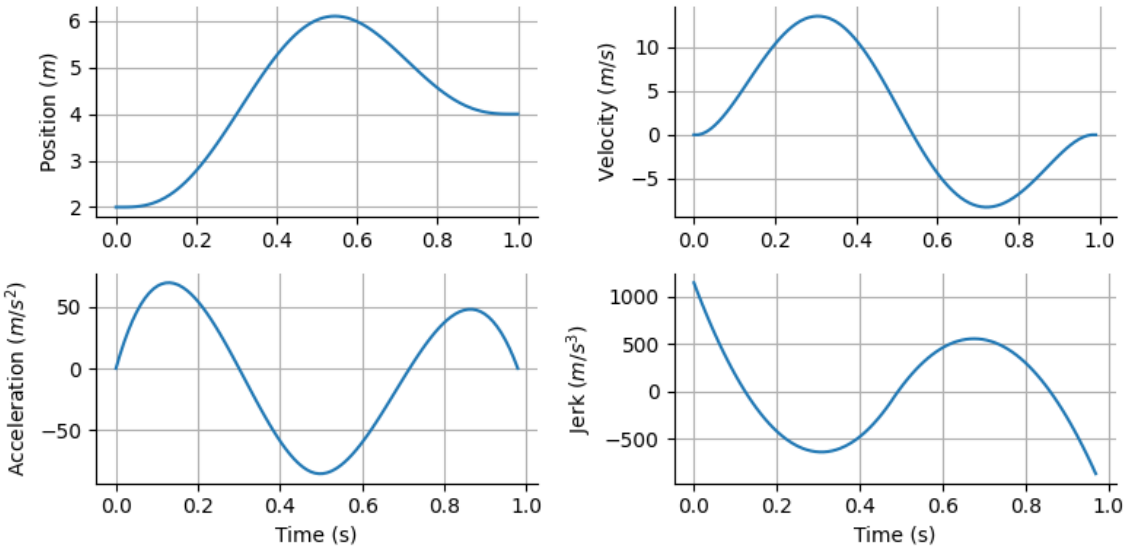

図 8.2 cell019.png

となる (ここで $\mathbf{I}$ は単位行列) ので，$\mathbf{A} = \mathbf{I} + \mathbf{A}_c dt, \mathbf{B} = \mathbf{B}_c dt$ として

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}(u_t + w_t) \tag{8.8}$$

と表せる．$\mathbf{x}_t$ の平均は

$$\mathbb{E}\left[\mathbf{x}_t\right] = \mathbf{A}^t \mathbf{x}_0 + \sum_{i=0}^{t-1} \mathbf{A}^{t-1-i} \mathbf{B} u_i \tag{8.9}$$

$\mathbf{x}_t$ の分散は

$$\mathrm{Cov}\left[\mathbf{x}_t\right] = k \sum_{i=0}^{t-1} \left(\mathbf{A}^{t-1-i}\mathbf{B}\right) \left(\mathbf{A}^{t-1-i}\mathbf{B}\right)^\top u_i^2 \tag{8.10}$$

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
x0₂ = zeros(2) # initial state (pos=0, vel=0)
xf₂ = [10, 0]  # final state (pos=10, vel=0)
Ac₂ = [0 1; -1/(t1*t2) -1/t1-1/t2];
Bc₂ = [0, 1]

# 3rd order
x0₃ = zeros(3)   # initial state (pos=0, vel=0, acc=0)
xf₃ = [10, 0, 0] # final state (pos=10, vel=0, acc=0)
Ac₃ = [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];
```

```
Bc₃ = [0, 0, 1/tm];
```

```
x₂, u₂, Σ₂ = minimum_variance_model(Ac₂, Bc₂, x0₂, xf₂, tf, tp, dt);
x₃, u₃, Σ₃ = minimum_variance_model(Ac₃, Bc₃, x0₃, xf₃, 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.

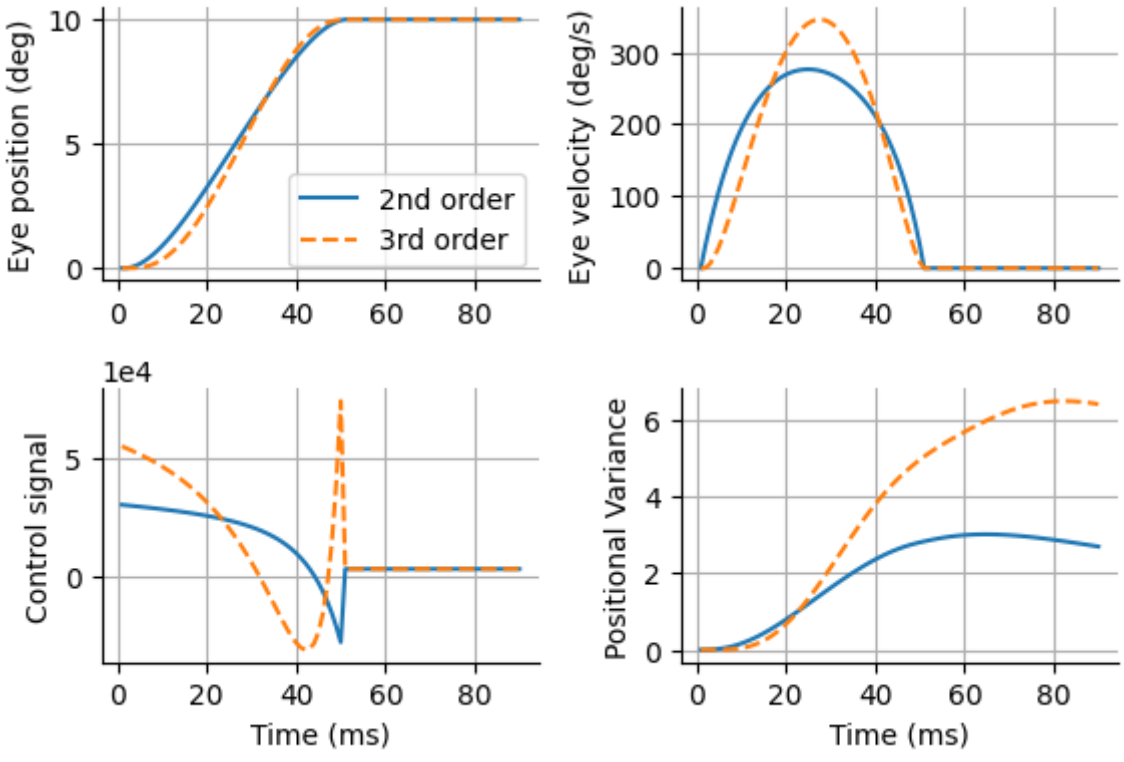

図 8.3 cell008.png

系の状態変化

$$\text{Dynamics} \quad \mathbf{x}_{t+1} = A\mathbf{x}_t + B\mathbf{u}_t + \boldsymbol{\xi}_t + \sum_{i=1}^{c} \varepsilon_t^i C_i \mathbf{u}_t \tag{8.11}$$

$$\text{Feedback} \quad \mathbf{y}_t = H\mathbf{x}_t + \omega_t + \sum_{i=1}^{d} \epsilon_t^i D_i \mathbf{x}_t \tag{8.12}$$

$$\text{Cost per step} \quad \mathbf{x}_t^\top Q_t \mathbf{x}_t + \mathbf{u}_t^\top R \mathbf{u}_t \tag{8.13}$$

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)

$$\mathbf{u}_t = -L_t \widehat{\mathbf{x}}_t \tag{8.14}$$

$$L_t = \left(R + B^\top S_{t+1} B\right)^{-1} B^\top S_{t+1} A \tag{8.15}$$

$$S_t = Q_t + A^\top S_{t+1} \left(A - B L_t\right) \tag{8.16}$$

$$s_t = \mathrm{tr}(S_{t+1}\Omega^\xi) + s_{t+1}; s_T = 0 \tag{8.17}$$

$\boldsymbol{S}_T = Q$

状態推定 (Kalman Filter)

$$\widehat{\mathbf{x}}_{t+1} = A\widehat{\mathbf{x}}_t + B\mathbf{u}_t + K_t\left(\mathbf{y}_t - H\widehat{\mathbf{x}}_t\right) + \boldsymbol{\eta}_t \tag{8.18}$$

$$K_t = A\Sigma_t H^\top \left(H\Sigma_t H^\top + \Omega^\omega\right)^{-1} \tag{8.19}$$

$$\Sigma_{t+1} = \Omega^\xi + \left(A - K_t H\right)\Sigma_t A^\top \tag{8.20}$$

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
    α3 = 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)
    @unpack n, p, A, B, C, D, G = param
    @unpack Q, R, x₁, Σ₁, dt, nt = cost_param

    A = I + A * dt
    B = B * dt
    C = C * dt
    D = sqrt(dt) * D
    G = sqrt(dt) * G

    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 += x₁' * S * x₁
    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.

$$\mathbf{u}_t = -L_t \hat{\mathbf{x}}_t \tag{8.21}$$

$$L_t = \left(B^\top S^{\mathrm{x}}_{t+1} B + R + \sum_n C_n^\top \left(S^{\mathrm{x}}_{t+1} + S^{\mathrm{e}}_{t+1} \right) C_n \right)^{-1} B^\top S^{\mathrm{x}}_{t+1} A \tag{8.22}$$

$$S^{\mathrm{x}}_t = Q_t + A^\top S^{\mathrm{x}}_{t+1} \left(A - B L_t \right); \quad S^{\mathrm{x}}_T = Q_T \tag{8.23}$$

$$S^{\mathrm{e}}_t = A^\top S^{\mathrm{x}}_{t+1} B L_t + \left(A - K_t H \right)^\top S^{\mathrm{e}}_{t+1} \left(A - K_t H \right); \quad S^{\mathrm{e}}_T = 0 \tag{8.24}$$

$$s_t = \mathrm{tr} \left(S^{\mathrm{x}}_{t+1} \Omega^\xi + S^{\mathrm{e}}_{t+1} \left(\Omega^\xi + \Omega^\eta + K_t \Omega^\omega K_t^\top \right) \right) + s_{t+1}; \quad s_n = 0. \tag{8.25}$$

$$\hat{\mathbf{x}}_{t+1} = A \hat{\mathbf{x}}_t + B \mathbf{u}_t + K_t \left(\mathbf{y}_t - H \hat{\mathbf{x}}_t \right) \tag{8.26}$$

$$K_t = A \Sigma^{\mathrm{e}}_t H^\top \left(H \Sigma^{\mathrm{e}}_t H^\top + \Omega^\omega \right)^{-1} \tag{8.27}$$

$$\Sigma^{\mathrm{e}}_{t+1} = \left(A - K_t H \right) \Sigma^{\mathrm{e}}_t A^\top + \sum_n C_n L_t \Sigma^{\hat{x}}_t L_t^\top C_n^\top; \quad \Sigma^{\mathrm{e}}_1 = \Sigma_1 \tag{8.28}$$

$$\Sigma^{\hat{\mathbf{x}}}_{t+1} = K_t H \Sigma^{\mathrm{e}}_t A^\top + \left(A - B L_t \right) \Sigma^{\hat{\mathbf{x}}}_t \left(A - B L_t \right)^\top; \quad \Sigma^{\hat{\mathbf{x}}}_1 = \hat{\mathbf{x}}_1 \hat{\mathbf{x}}_1^\top \tag{8.29}$$

```
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, x₁, Σ₁, 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
        Sˣ = copy(Q[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
        sqrtdt = √dt
        X^ = zeros(n, nt)
        X^[1, 1] = X[1, 1]
        for t in 1:nt-1
            u[t] = -L[t, :]' * X^[:, t]
            X[:, t+1] = X[:,t] + (A * X[:,t] + B * u[t]) * dt + sqrtdt * (Y ↵
                * u[t] * randn() + G * randn(n))
            dy = C * X[:,t] * dt + D * sqrtdt * 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()

    tight_layout()
end
```

```
param = Reaching1DModelParameter()
cost_param = Reaching1DModelCostParameter();
```

```
L, K, cost = LQG(param, cost_param);
simulation_all(param, cost_param, L, K)
```

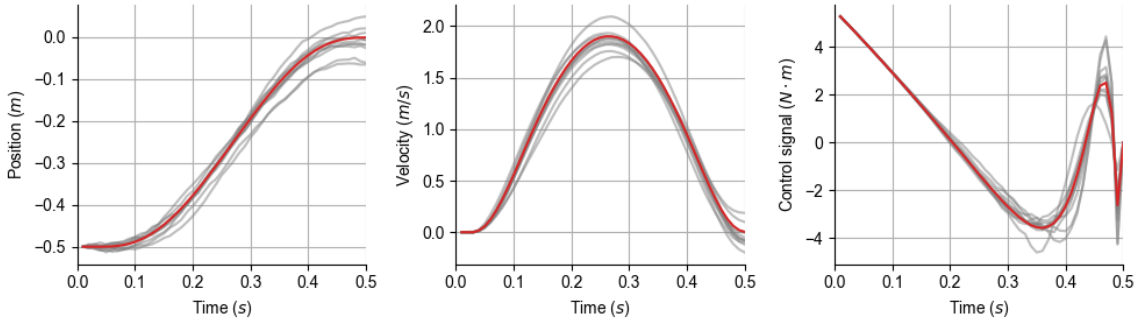

図 8.4 cell016.png

```
L, K, cost = gLQG(param, cost_param);
simulation_all(param, cost_param, L, K)
```

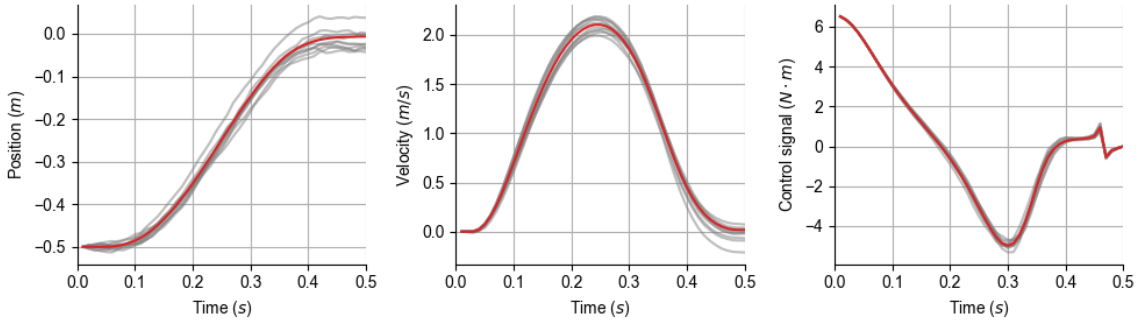

図 8.5 cell017.png

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) [?]

$$dx = (\mathbf{A}x + \mathbf{B}u)dt + \mathbf{Y}ud\gamma + \mathbf{G}d\omega \tag{8.30}$$
$$dy = \mathbf{C}xdt + \mathbf{D}d\xi \tag{8.31}$$
$$d\hat{x} = (\mathbf{A}\hat{x} + \mathbf{B}u)dt + \mathbf{K}(dy - \mathbf{C}\hat{x}dt) \tag{8.32}$$

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

$$\alpha_1 = \frac{b}{t_a t_e I}, \quad \alpha_2 = \frac{1}{t_a t_e} + \left(\frac{1}{t_a} + \frac{1}{t_e}\right)\frac{b}{I} \tag{8.33}$$
$$\alpha_3 = \frac{b}{I} + \frac{1}{t_a} + \frac{1}{t_e}, \quad b_u = \frac{1}{t_a t_e I} \tag{8.34}$$

```
@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
```

$$\mathbf{X} \triangleq \begin{bmatrix} x \\ \tilde{x} \end{bmatrix}, d\bar{\omega} \triangleq \begin{bmatrix} d\omega \\ d\xi \end{bmatrix}, \bar{\mathbf{A}} \triangleq \begin{bmatrix} \mathbf{A} - \mathbf{BL} & \mathbf{BL} \\ \mathbf{0} & \mathbf{A} - \mathbf{KC} \end{bmatrix} \tag{8.35}$$

$$\bar{\mathbf{Y}} \triangleq \begin{bmatrix} -\mathbf{YL} & \mathbf{YL} \\ -\mathbf{YL} & \mathbf{YL} \end{bmatrix}, \bar{G} \triangleq \begin{bmatrix} \mathbf{G} & \mathbf{0} \\ \mathbf{G} & -\mathbf{KD} \end{bmatrix} \tag{8.36}$$

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．

$$\mathbf{P} \triangleq \begin{bmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{12} & \mathbf{P}_{22} \end{bmatrix} = \mathbb{E}\left[\mathbf{X}\mathbf{X}^\top\right] \tag{8.37}$$

$$\mathbf{V} \triangleq \begin{bmatrix} \mathbf{Q} + \mathbf{L}^\top R\mathbf{L} & -\mathbf{L}^\top R\mathbf{L} \\ -\mathbf{L}^\top R\mathbf{L} & \mathbf{L}^\top R\mathbf{L} + \mathbf{U} \end{bmatrix} \tag{8.38}$$

aaa

$$K = \mathbf{P}_{22}\mathbf{C}^\top \left(\mathbf{D}\mathbf{D}^\top\right)^{-1} \tag{8.39}$$

$$\mathbf{L} = \left(R + \mathbf{Y}^\top \left(\mathbf{S}_{11} + \mathbf{S}_{22}\right)\mathbf{Y}\right)^{-1} \mathbf{B}^\top \mathbf{S}_{11} \tag{8.40}$$

$$\bar{\mathbf{A}}^\top \mathbf{S} + \mathbf{S}\bar{\mathbf{A}} + \bar{\mathbf{Y}}^\top \mathbf{S}\bar{\mathbf{Y}} + \mathbf{V} = 0 \tag{8.41}$$

$$\bar{\mathbf{A}}\mathbf{P} + \mathbf{P}\bar{\mathbf{A}}^\top + \bar{\mathbf{Y}}\mathbf{P}\bar{\mathbf{Y}}^\top + \bar{\mathbf{G}}\bar{\mathbf{G}}^\top = 0 \tag{8.42}$$

$\mathbf{A} = (a_{ij})$ 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

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \tag{8.43}$$

で与えられる $mp \times nq$ 区分行列である．

Roth's column lemma (vec-trick)

$$(\mathbf{B}^\top \otimes \mathbf{A})\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{AXB}) = \mathrm{vec}(\mathbf{C}) \tag{8.44}$$

によりこれを解くと，

$$\mathbf{S} = -\mathrm{vec}^{-1}\left(\left(\mathbf{I}\otimes\bar{\mathbf{A}}^\top + \bar{\mathbf{A}}^\top\otimes\mathbf{I} + \bar{\mathbf{Y}}^\top\otimes\bar{\mathbf{Y}}^\top\right)^{-1}\mathrm{vec}(\mathbf{V})\right) \tag{8.45}$$

$$\mathbf{P} = -\mathrm{vec}^{-1}\left(\left(\mathbf{I}\otimes\bar{\mathbf{A}} + \bar{\mathbf{A}}\otimes\mathbf{I} + \bar{\mathbf{Y}}\otimes\bar{\mathbf{Y}}\right)^{-1}\mathrm{vec}(\bar{\mathbf{G}}\bar{\mathbf{G}}^\top)\right) \tag{8.46}$$

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
    I₂ₙ = I(2n)

    for _ in 1:maxiter
        A¯ = [A-B*L B*L; zeros(size(A)) (A-K*C)]
        Y¯ = [-ones(2) ones(2)] ⊗ (Y*L)
        G¯ = [G zeros(size(K)); G (-K*D)]
        V = BlockDiagonal([Q, U]) + [1 -1; -1 1] ⊗ (L'* R * L)

        # update S, P
        S = -reshape((I₂ₙ ⊗ (A¯)' +  (A¯)' ⊗ I₂ₙ + (Y¯)' ⊗ (Y¯)') \ vec(V), (2n, 2n))
        P = -reshape((I₂ₙ ⊗ A¯ +  A¯ ⊗ I₂ₙ + Y¯ ⊗  Y¯) \ vec(G¯ * (G¯)'), (2n, 2n))

        # update K, L
        P₂₂ = P[n+1:2n, n+1:2n]
        S₁₁ = S[1:n, 1:n]
        S₂₂ = S[n+1:2n, n+1:2n]

        K_{t-1} = copy(K)
```

```
        L_{t-1} = copy(L)

        K = P₂₂ * C' / (D * D')
        L = (R + Y' * (S₁₁ + S₂₂) * Y) \ B' * S₁₁
        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
        sqrtdt = √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 + sqrtdt * (Y ↵
                * u[t] * randn() + G * randn(n))
            dy = C * X[:,t] * dt + D * sqrtdt * 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: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()

    tight_layout()
```

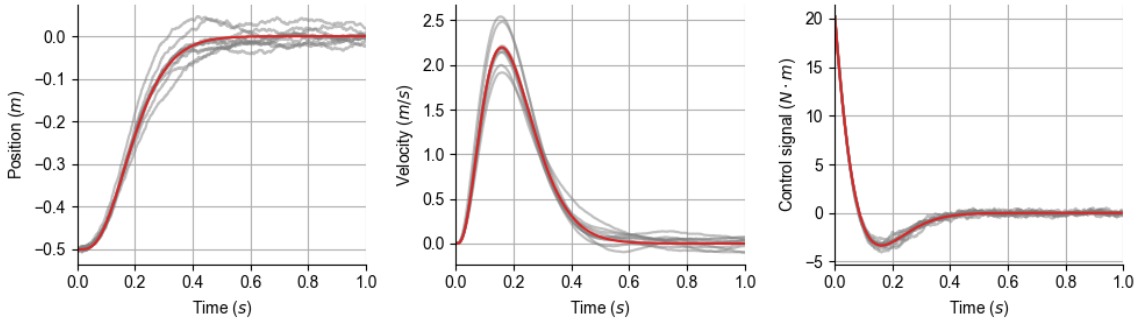

図 8.6 cell017.png

8.4.3 Target jump

target jump する場合の最適制御 [?]. 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
        sqrtdt = √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 + sqrtdt * (Y ↵
                * u[t] * randn() + G * randn(n))
            dy = C * X[:,t] * dt + D * sqrtdt * 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:nsim
    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[1].plot(tarray, target_pos, "tab:green")
    for j in 1:nsim
        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 ($s$)"); ↵
        ax[i].set_xlim(0, T); ax[i].grid()
end
for j in 1:nsim
    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 ($N\cdot m$)"); ax[3].set_xlabel(L"Time ↵
    ($s$)"); ax[3].set_xlim(0, T); ax[3].grid()

tight_layout()
```

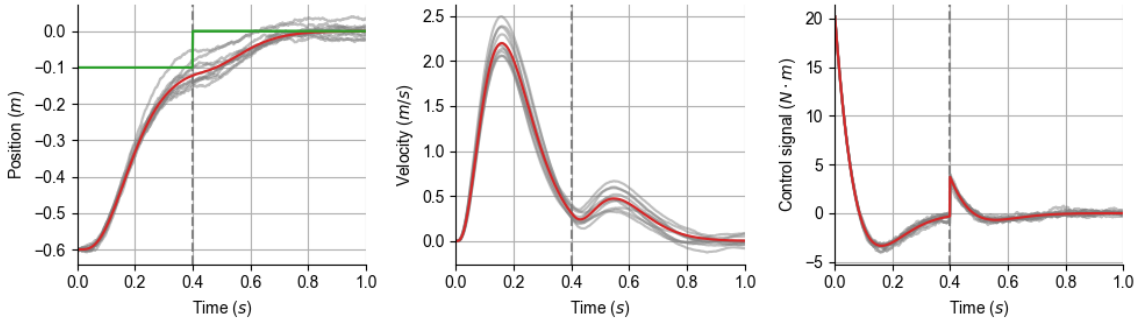

図 8.7 cell023.png

第 9 章

強化学習

第 10 章

神経回路網によるベイズ推論

索引