Simulating Neurite Growth using MATLAB

Igor Guranović

11 October 2021

Abstract

The aim of this project is to simulate neurite growth as a discrete-time stochastic process. These neurites can become axons or dendrites, which can form synapses to allow for communication between neurons. Neurite wandering is random but also depends on chemical and mechanical signals (Maskery & Shinbrot 2005). Overall, the physiology and features of the nervous system allow for some intricate computational simulations.

1 Basic Features

1.1 Restricted Wandering of a Neurite

One of the two requirements of this project is to make each neurite tip randomly wander but be restricted by a maximum angle. In this program, two maximum angles were incorporated: a maximum elevation angle and a maximum azimuthal angle. These correspond to $\phi_{\max}$ and $\theta_{\max}$, respectively, in spherical coordinates. Each neurite is programmed to grow at a fixed rate of one unit per time step, and these angles are the maximum angles by which a neurite's trajectory can change over the course of one time step. The elevation angle by which a neurite's trajectory deviates from its current direction is denoted by ϕ, which ranges from $-\phi_{\max}$ to $\phi_{\max}$ following a uniform probability distribution when the simulation is in "standard" mode. Similarly, the azimuthal angle by which a neurite's trajectory deviates from its current direction is denoted by θ, which ranges from $-\theta_{\max}$ to $\theta_{\max}$ following a uniform probability distribution when the simulation is in "standard" mode.

1.2 Branching and Pruning of Neurites

The second requirement of this project is to have a certain probability of branching (splitting into two neurite tips) in a given time step, denoted by λ_b, and another probability of pruning in a given time step, denoted by λ_p. In fact, this can be modeled as a Markov chain, specifically a branching process. This way, one can determine the probability of a neurite (and all of its branches) dying out after n time steps given λ_b and λ_p. It is vital to consider this extinction probability when setting λ_b and λ_p values on the program, so that the runtime does not get excessively long. Let $\phi(s) = \sum_i p_i s^i$ (not to be conflated with ϕ denoting elevation angle), where p_i is the probability of a single neurite forming i branches after a time step. Intuitively, $p_0 = \lambda_p$, $p_1 = (1 - \lambda_p - \lambda_b)$, and $p_2 = \lambda_b$. There is a theorem stating that $a_n = \phi(a_{n-1})$, where a_n is the extinction probability after n time steps (Lawler 2018). In other words, a_n is the probability that a neurite and its branches will all be dead at time n. The proof can be found in Gregory Lawler's book, *Introduction to Stochastic Processes*. $a_0 = 0$, because a live neurite obviously cannot die out after zero time steps. This allows for a recursive solution for a_n, provided λ_p, λ_b, and n. This is how a_n should be interpreted in the context of the program: If N is the total simulation time, a_N should be fairly close to 1 if N is a large value, to shorten Matlab runtime. Contrarily, if a_n is close to 1 when $n << N$, then λ_b is too small or λ_p is too large, resulting in a near-dead system long before $n = N$.

Modeling this in Matlab was made possible thanks to structures and recursion. Each structure corresponds to an individual neurite tip, and contains a cell array corresponding to the data of said neurite tip. When branching occurs, the neurite tip's propagation "shuts off" but its structure gains two additional sub-structures which will propagate in the exact same way as the original neurite tip. When pruning occurs, the neurite tip's propagation simply "shuts off" without creating additional structures. The program also contains a recursive function to search for every possible nested structure in the system and propagate appropriately.

Figure 1: Branching and Pruning

2 Advanced Features

2.1 Chemical Effect on Growth

One of the more advanced features incorporated is chemical influence on neurite growth. When the simulation mode is selected to "chemical" as opposed to "standard", the chemical sources play a role in influencing the probabilities of the random trajectories of neurites, as opposed to having uniform elevation/azimuthal angle distributions. In the real world, chemical effects on neurons are extremely intricate, so this feature is quite simplified in the program.

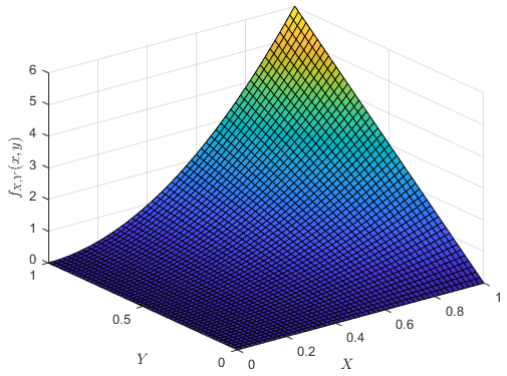
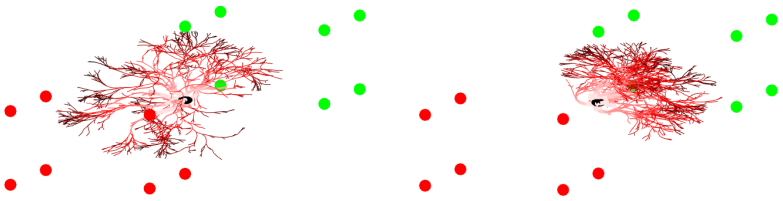


Figure 2: Example Joint Probability Density Function vs. X and Y

Probability density functions are the main concept from probability theory that made this "chemical effect" feature possible. Since these functions are continuous, the probability is calculated by an integral over a region as opposed to a single point. This is how to calculate the probability of an event occurring in an arbitrary 2-dimensional region A.

$$P(A) = \iint_A f_{X,Y}(x,y)\,\mathrm{d}x\mathrm{d}y$$

The chemical effect on neurite growth was modeled using point-like objects acting as sources of chemicals. Each chemical type has a set efficacy, denoted as e, a parameter that is going to determine to what extent neurites are influenced by it, as well the coordinates of the sources. A positive efficacy corresponds to neurites being more likely to grow towards the chemical source, and the opposite holds for negative efficacies. The net effect of all chemicals in the environment is at the core of how the joint probability density function is developed. First, the program determines the unit vector of the most favorable direction in which a neurite can grow. This concept of "favorability" is similar to net electrical force in Coulomb's Law, and the efficacy parameter is analogous to the charge of a point-like object. The most favorable unit vector, $\hat{C}$, is analogous to the unit vector of net force caused by a set of charges. This vector is calculated like this, where $\overrightarrow{n}$ represents the current position vector of the neurite tip, e_i represents the efficacy of the i^{th} chemical source, and $\overrightarrow{c_i}$ represents



(a) Standard Mode (b) Chemical Mode

the position vector of the i^{th} chemical source:

$$\overrightarrow{C} = \sum_i \frac{e_i(\overrightarrow{c_i} - \overrightarrow{n})}{||\overrightarrow{c_i} - \overrightarrow{n}||^3}, \quad \hat{C} = \frac{\overrightarrow{C}}{||\overrightarrow{C}||}$$

Now, the joint probability density function is created using this unit vector. The two inputs to this function are ϕ and θ, which range from $-\phi_{\max}$ to $\phi_{\max}$ and $-\theta_{\max}$ to $\theta_{\max}$, respectively. For each point (ϕ, θ) in this 2-dimensional input space, a unit vector $\hat{v}(\phi, \theta)$ is calculated, which represents the Cartesian direction in which a neurite would grow if it deviated by ϕ and θ from its previous direction. The function itself is found like this: $f_{\Phi,\Theta}(\phi, \theta) = k(\frac{\hat{C} \cdot \hat{v}(\phi,\theta)}{2} + \frac{1}{2})^n$, where n is an arbitrary choice of exponent influencing how powerful this chemical effect is as a whole, and k is a normalizing constant such that:

$$\int_{-\phi_{\max}}^{\phi_{\max}} \int_{-\theta_{\max}}^{\theta_{\max}} f_{\Phi,\Theta}(\phi, \theta)\, \mathrm{d}\theta \mathrm{d}\phi = 1$$

The dot product is divided by 2 and added to $\frac{1}{2}$ in order for it to be in the range of [0,1] instead of [-1,1]. The random values of ϕ and θ are generated through this probability density function, and those are used to propagate a neurite. This process is repeated for every neurite currently growing, and during each time step.

2.2 Aesthetics

The aesthetics of the simulation are mostly based around the "tier" parameter. In this program, the tier refers to how many times that neurite has branched, starting at tier 1 for neurites originating at the cell body. For example, if a neurite originates at the cell body, splits into two, and each of those split into two, those four new neurite tips are of tier 3. The tier of a certain neurite segment influences the colors, as well as the line widths on the plot. Another aesthetic feature implemented is making the size of the chemical sources a function of their efficacies. The color of the chemical sources corresponds to the sign of the efficacies (green = positive, red = negative). In addition, when a synapse forms, the dendrite turns magenta and the axon turns yellow.

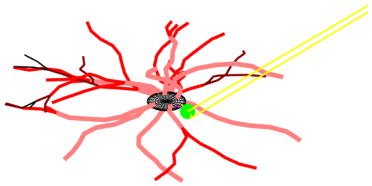
2.3 Multiple Neurons and Synapse Formation/Secretion

This program has the option of initializing multiple cell bodies (represented as black/white spheres) from which neurites grow and branch off. There is also an option of neurons forming synapses. Since this is a heavily simplified simulation of how synapses are formed, the regulations that the program abides by are as follows:

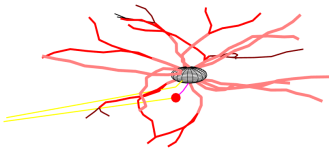
1. Synapses are formed between one axon and one dendrite, and each neuron has a limit of one dendrite formation and one axon formation, resulting in a maximum of two connections per neuron.

2. If Neuron A does not yet have a dendrite, Neuron B does not yet have an axon, and Neurons A and B each have a neurite tip such that the two neurite tips are within a maximum distance of each other, a synapse will form between the neurons in the next time step. Neuron A's neurite becomes a dendrite and Neuron B's neurite becomes an axon.

When a synapse forms, there is a chance of a stimulating chemical (positive efficacy value) being secreted, denoted by λ_{sc}, as well as a chance of an inhibitory chemical (negative efficacy value) being secreted, denoted by λ_{ic}. Secretion of a chemical consists of simply adding a chemical source at the synapse, functioning exactly the same as other chemical sources depicted in section 3.1. This makes synapse formation influence further neurite growth and allows for communication between neurons.



(c) Stimulating Synapse



(d) Inhibitory Synapse

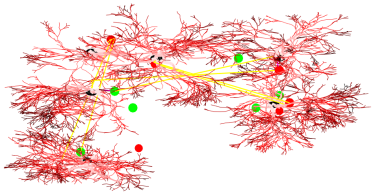
2.4 Total Control

The final goal when creating this program was to make a master control unit that allows for easy modifications and endless customization. Some of these knobs and dials include being able to control the:

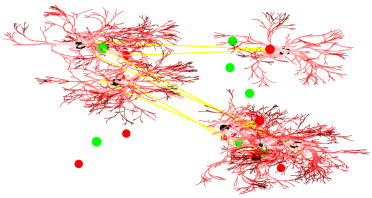
1. Amount of chemical sources, their positions, and their respective efficacy values
2. Amount of neurons, their positions, and number of neurites growing out of each cell body
3. Maximum elevation angle, maximum azimuthal angle, probability of branching, and probability of pruning, all as anonymous functions of the tier value
4. Probability of a synapse being stimulating or inhibitory
5. Maximum distance required for forming a synapse
6. Total time
7. Simulation mode (standard or chemical)
8. Chemical effect exponent (the n exponent found in the equation for $f_{\Phi,\Theta}(\phi,\theta)$ in section 3.1)
9. Aesthetic features (colors, widths) mentioned in section 3.2
10. Coordinate axis limits

3 Putting it All Together

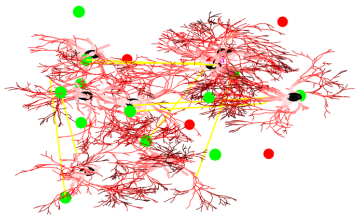
Now that all of the features of the program are explained, here are a few images of the simulation that encompass everything mentioned in this report thus far.



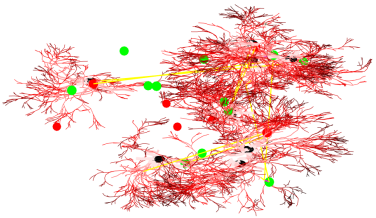
[Simulation 1]



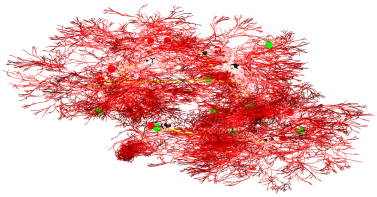
[Simulation 2]



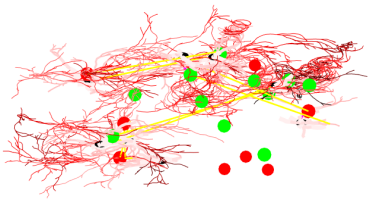
[Simulation 3]



[Simulation 4]



[Simulation 5]



[Simulation 6]

4 Reflection

There definitely are additional features that I would have implemented into this program if I had more time. One of these being a way to incorporate action potentials, since one of the main functions of a neuron is to propagate these potentials. Also, I would have researched the molecular biology of neurons and figured out how chemicals impact neurite development, as well as the formation of dendrites and axons. The features implemented in the program are fully functional from a mathematical modeling perspective but could be improved from a scientific perspective. This is because the chemical sources function similarly to point charges and formation of dendrites/axons is dependent only on proximity of the respective neurites. However, I am quite satisfied with how customizable and versatile I made this program, thanks to the master control unit. Hopefully, I am able to use these ideas to refine this program even further in the near future.

5 Code

```
clear
clc

%MASTER CONTROL UNIT

%{
Initializes chemical array (in effect if parameters.mode is set to "chemical")
Parameters:
- Chemical locations
- Chemical efficacies
%}
chemicalArray = {
                {
                    {} 100
                }
                {
                    {} -100
                }
                {
                    {[6;15;0] [-12;-1;0]} 100
                }
                {
                    {[9;-1;0] [13;3;-12]} 50
                }
                {
                    {[13;-2;0] [-11;-14;-12]} -25
                }
            };

%{
Initial conditions of neurons
Parameters:
-Number of neurites branching out of cell body
-Location of cell body
%}
neuronSchematic = {{[12;0;5],15}
                   {[-8;15;12],15}
                   {[-19;4;-7],15}
                   {[13;15;11],15}
                   {[-20;-18;-16],15}
                   {[-19;17;-19],15}};

%Maximum elevation angle by which neurite tip can vary direction
parameters.maxElevation = @(tier) pi/6;

%Maximum azimuthal angle by which neurite tip can vary direction
parameters.maxAzimuth = @(tier) pi/6;

%Probability of pruning as a function of tier
parameters.lambdaPrune = @(tier) 0.2;

%Probability of branching as a function of tier
parameters.lambdaBranch = @(tier) 0.4;

%Probability of synapse secreting stimulating chemical
parameters.lambdaPositiveSecretion = 0.5;
```

```
%Probability of synapse secreting inhibitory chemical
parameters.lambdaNegativeSecretion = 0.5;

%Maximum distance for synapses to form between neurons
parameters.synapseThreshold = 40;

%Total amount of time steps in the simulation
parameters.totalTime = 20;

%Simulation mode - standard (no chemical effects) or chemical
parameters.mode = "chemical";

%How strongly chemicals affect neurite probabilities
parameters.chemicalEffectExponent = 10;

%Width of neurites as a function of tier
neuralWidthScale = 8;
neuralWidthInner = 0.4;
neuralWidth = @(tier) neuralWidthScale*exp(-neuralWidthInner*tier);

%Neurite color as a function of tier and maximum tier
neuralColor = @(tier, maxtier) uint8(getColor(tier, maxtier));

%Chemical size as a function of efficacy (how strong chemical is)
chemicalSizeScale = 8;
chemicalSizeInner = 0.1;
chemicalSize = @(efficacy) chemicalSizeScale*atan(abs(chemicalSizeInner*efficacy));

%x,y,z Axis limits
coordLimits = [-50,50];

%Creates an initial master cell array given initial conditions
initializedCell = initialize(neuronSchematic,parameters);

%Loops through timeStep function and moves the masterCells array through time
[masterCells, chemicalArray] = timeStep(initializedCell,2,parameters,chemicalArray);
for i = 3:parameters.totalTime
    [masterCells, chemicalArray] = timeStep(masterCells,i,parameters, chemicalArray);
end

%Packages masterCells array for easy graphing
packagedCells = cellPackager(masterCells);

%Calculates maximum tier
tiers = [];
for i=1:length(packagedCells)
    tiers = [tiers packagedCells{i}{2}];
end
maxTier = max(tiers);

hold on
%Graphs neurons
for i=1:length(packagedCells)
    dataMatrix = packagedCells{i}{1};
    tier = packagedCells{i}{2};
    if length(packagedCells{i}) == 4
        color = [1 1 0];
        width = 2;
    elseif length(packagedCells{i}) == 5
        color = [1 0 1];
        width = 2;
    else
        color = neuralColor(tier,maxTier);
        width = neuralWidth(tier);
    end
    plot3(dataMatrix(3,:),dataMatrix(4,:),dataMatrix(5,:),'LineWidth',width, ...
        'color', color)
end

%Graphs chemicals
for i = 1:length(chemicalArray)
    for j = 1:length(chemicalArray{i}{1})
        if chemicalArray{i}{2} < 0
            color = 'r';
        else
```

```
            color = 'g';
        end
        scatter3(chemicalArray{i}{1}{j}(1),chemicalArray{i}{1}{j}(2), ...
            chemicalArray{i}{1}{j}(3),color,'LineWidth', ...
            chemicalSize(chemicalArray{i}{2}))
    end
end

%Graphs cell bodies
for i = 1:length(neuronSchematic)
    coord = neuronSchematic{i}{1};
    [X,Y,Z] = sphere;
    X = X + coord(1);
    Y = Y + coord(2);
    Z = Z + coord(3);
    surf(X,Y,Z,ones(21,21,3))
end

xlim(coordLimits)
ylim(coordLimits)
zlim(coordLimits)

function [outputCell, chemicals] = timeStep(inputCell, time, parameters, chemicals)
    outputCell = inputCell;
    for i = 1:length(inputCell)
        for j = 1:length(inputCell{i}{1})
            [outputCell{i}{1}{j}, chemicals] = neuriteStep(inputCell{i}{1}{j}, 1, ...
                time, parameters, chemicals);
        end
    end
    verdictOut = 0;
    for i = 1:length(outputCell)
        for j = 1:length(outputCell)
            if (i ~= j) && (outputCell{i}{2} == 0) && (outputCell{j}{3} == 0)
                for k = 1:length(outputCell{i}{1})
                    if verdictOut
                        break
                    end
                    for l = k:length(outputCell{j}{1})
                        [outputCell{i}{1}{k}, outputCell{j}{1}{l}, verdictOut] = ...
                            pairTree(outputCell{i}{1}{k}, outputCell{j}{1}{l}, time, ...
                            parameters);
                        if verdictOut
                            outputCell{i}{2} = 1;
                            outputCell{j}{3} = 1;
                            break
                        end
                    end
                end
            end
        end
    end
end

function [outputStructure1, outputStructure2, verdictOut] = pairTree(inputStructure1, ...
    inputStructure2, time, parameters)
    nameListIK = fieldnames(inputStructure1);
    nameListJL = fieldnames(inputStructure2);
    outputStructure1 = inputStructure1;
    outputStructure2 = inputStructure2;
    for i = 1:numel(nameListIK)
        for j = 1:numel(nameListJL)
            if isstruct(inputStructure1.(nameListIK{i})) && ...
                    isstruct(inputStructure2.(nameListJL{j}))
                [outputStructure1.(nameListIK{i}),  outputStructure2.(nameListJL{j}), ...
                    verdictOut] = pairTree(inputStructure1.(nameListIK{i}), ...
                    inputStructure2.(nameListJL{j}), time, parameters);
                if verdictOut
                    break
                end
            elseif isstruct(inputStructure1.(nameListIK{i}))
                [outputStructure1.(nameListIK{i}),  outputStructure2, verdictOut] = ...
                    pairTree(inputStructure1.(nameListIK{i}), inputStructure2, time, ...
                    parameters);
                if verdictOut
                    break
```

```
            end
        elseif isstruct(inputStructure2.(nameListJL{j}))
            [outputStructure1,  outputStructure2.(nameListJL{j}), verdictOut] = ...
                pairTree(inputStructure1, inputStructure2.(nameListJL{j}), time, ...
                parameters);
            if verdictOut
                break
            end
        else
            cell1 = inputStructure1.(nameListIK{i});
            cell2 = inputStructure2.(nameListJL{j});
            normcheck = norm(cell1{1}(3:5,time+1)-cell2{1}(3:5,time+1));
            if (normcheck < parameters.synapseThreshold) && (cell1{3} == "alive") ...
                    && (cell2{3} == "alive") && (cell2{1}(1,1) ~= cell2{1}(1,time+1))
                cell1{3} = "axon";
                cell2{3} = "dead";
                cell2{5} = 'placeholder';
                cell1{4} = cell2{1}(:,time+1);
                for k=time+2:parameters.totalTime+1
                    cell2{1}(:,k) = cell2{1}(:,time+1);
                end
                outputStructure1.(nameListIK{i}) = cell1;
                outputStructure2.(nameListJL{j}) = cell2;
                verdictOut = 1;
                break
            else
                verdictOut = 0;
            end
        end
    end
    if verdictOut
        break
    end
end
end

function [outputStructure, chemicals] = neuriteStep(inputStructure, tier, time, ...
    parameters, chemicals)
    outputStructure = struct;
    nameList = fieldnames(inputStructure);
    for i = 1:numel(nameList)
        if iscell(inputStructure.(nameList{i}))
            if inputStructure.(nameList{i}){3} == "alive"
                outputStructure.(nameList{i}) = ...
                    matrixPropagation(inputStructure.(nameList{i}){1}, tier, ...
                    parameters, time, chemicals);
                rng1 = rand(1);
                if rng1<parameters.lambdaPrune(tier)
                    for k=time+2:parameters.totalTime+1
                        outputStructure.(nameList{i}){1}(:,k) = ...
                            outputStructure.(nameList{i}){1}(:,time+1);
                    end
                    outputStructure.(nameList{i}){3} = "dead";
                elseif (rng1>(1-parameters.lambdaBranch(tier))) ...
                        && (time ~= parameters.totalTime)
                    for k=time+2:parameters.totalTime+1
                        outputStructure.(nameList{i}){1}(:,k) = ...
                            outputStructure.(nameList{i}){1}(:,time+1);
                    end
                    dataMatrixBranch = zeros(5,parameters.totalTime+1)+1;
                    for k=1:5
                        dataMatrixBranch(k,1:time+1)=...
                            outputStructure.(nameList{i}){1}(k,time+1);
                    end
                    outputStructure.branch1.matrix = {dataMatrixBranch tier+1 "alive"};
                    outputStructure.branch2.matrix = {dataMatrixBranch tier+1 "alive"};
                    outputStructure.(nameList{i}){3} = "dead";
                end
            elseif inputStructure.(nameList{i}){3} == "axon"
                outputStructure.(nameList{i}) = inputStructure.(nameList{i});
                for k=time+1:parameters.totalTime+1
                    outputStructure.(nameList{i}){1}(:,k) = ...
                        inputStructure.(nameList{i}){4};
                end
                rng1 = rand(1);
                outputStructure.(nameList{i}){3} = "dead";
```

```
                if rng1<parameters.lambdaPositiveSecretion
                    chemicals{1}{1} = singleAppendCell(chemicals{1}{1}, ...
                        outputStructure.(nameList{i}){1}(3:5,time+1));
                elseif rng1>(1-parameters.lambdaNegativeSecretion)
                    chemicals{2}{1} = singleAppendCell(chemicals{2}{1}, ...
                        outputStructure.(nameList{i}){1}(3:5,time+1));
                end
            else
                outputStructure.(nameList{i}) = inputStructure.(nameList{i});
            end
        elseif isstruct(inputStructure.(nameList{i}))
            [outputStructure.(nameList{i}), chemicals] = ...
                neuriteStep(inputStructure.(nameList{i}),tier+1,time,parameters, ...
                chemicals);
        end
    end
end

function outputCell = matrixPropagation(inputMatrix, tier, parameters, time, chemicals)
    dataMatrix = inputMatrix;
    [dataMatrix(1,time+1),dataMatrix(2,time+1)] = randomspherical(parameters, ...
        dataMatrix(:,time), chemicals, tier);
    [xTemp,yTemp,zTemp] = sph2cart(dataMatrix(1,time+1),dataMatrix(2,time+1),1);
    transformY = [cos(-dataMatrix(2,time)),0,sin(-dataMatrix(2,time));
        0,1,0;
        -sin(-dataMatrix(2,time)),0,cos(-dataMatrix(2,time))];
    transformZ = [cos(dataMatrix(1,time)),-sin(dataMatrix(1,time)),0;
        sin(dataMatrix(1,time)),cos(dataMatrix(1,time)),0;
        0,0,1];
    transformedCartesian = transformZ*transformY*[xTemp;yTemp;zTemp];
    [dataMatrix(1,time+1),dataMatrix(2,time+1)] = cart2sph(transformedCartesian(1), ...
        transformedCartesian(2),transformedCartesian(3));
    dataMatrix(3:5,time+1) = transformedCartesian(1:3) + dataMatrix(3:5,time);
    outputCell = {dataMatrix, tier, "alive"};
end

function outputCell = initialize(inputCell,parameters)
    outputCell = {};
    for i = 1:length(inputCell)
        for j = 1:inputCell{i}{2}
            matrixInit = zeros(5,parameters.totalTime+1);
            matrixInit(1,2)=(rand(1)-0.5)*2*pi;
            matrixInit(2,2)=(rand(1)-0.5)*pi;
            [matrixInit(3,2),matrixInit(4,2),matrixInit(5,2)] = ...
                sph2cart(matrixInit(1,2),matrixInit(2,2),1);
            matrixInit(3:5,1) = matrixInit(3:5,1) + inputCell{i}{1};
            outputCell{i}{1}{j}.matrix{1} = matrixInit;
        end
    end
    for i = 1:length(inputCell)
        for j = 1:length(outputCell{i}{1})
            outputCell{i}{1}{j}.matrix{1}(3:5,2) = ...
                outputCell{i}{1}{j}.matrix{1}(3:5,2) + inputCell{i}{1};
            outputCell{i}{1}{j}.matrix{3} = "alive";
        end
        outputCell{i}{2} = 0;
        outputCell{i}{3} = 0;
    end
end

function [azimuthalAngle,elevationAngle] = randomspherical(parameters, ...
    previousVector,chemicals,tier)
    if parameters.mode == "standard"
        elevationAngle = (rand(1)-0.5)*parameters.maxElevation(tier)*2;
        azimuthalAngle = (rand(1)-0.5)*parameters.maxAzimuth(tier)*2;
    elseif parameters.mode == "chemical"
        probabilityCell = {};
        totalProbability = 0;
        probabilityArray = [];
        force = 0;
        for i = 1:length(chemicals)
            for j = 1:length(chemicals{i}{1})
                force  = force + (chemicals{i}{2})*(chemicals{i}{1}{j}-...
                    previousVector(3:5))/((norm(chemicals{i}{1}{j}-...
                    previousVector(3:5)))^3);
            end
```

```
        end
        normForce = force/norm(force);
        azimuthalGrid = -parameters.maxAzimuth(tier):0.1:parameters.maxAzimuth(tier);
        elevationGrid = -parameters.maxElevation(tier):0.1:parameters.maxElevation(tier);
        transformY = [cos(-previousVector(2)),0,sin(-previousVector(2));
                      0,1,0;
                      -sin(-previousVector(2)),0,cos(-previousVector(2))];
        transformZ = [cos(previousVector(1)),-sin(previousVector(1)),0;
                      sin(previousVector(1)),cos(previousVector(1)),0;
                      0,0,1];
        for i = 1:length(azimuthalGrid)
            for j = 1:length(elevationGrid)
                [xTemp,yTemp,zTemp] = sph2cart(azimuthalGrid(i),elevationGrid(j),1);
                transformedCartesian = transformZ*transformY*[xTemp;yTemp;zTemp];
                cartesianPosition = transformedCartesian(1:3) + previousVector(3:5);
                normCartesian = cartesianPosition/norm(cartesianPosition);
                probabilityDensity = dot(normCartesian, normForce);
                probabilityDensity = ...
                    ((probabilityDensity/2)+1/2)^parameters.chemicalEffectExponent;
                probabilityCell = singleAppendCell(probabilityCell, ...
                    [probabilityDensity, i, j]);
                totalProbability = totalProbability + probabilityDensity;
                probabilityArray = [probabilityArray probabilityDensity];
            end
        end
        randomProbability = rand(1)*totalProbability;
        cumulativeProbabilityArray = [];
        for i = 1:length(probabilityCell)
            cumulativeProbabilityArray(i) = sum(probabilityArray(1:i));
        end
        for i=1:length(cumulativeProbabilityArray)
            if randomProbability < cumulativeProbabilityArray(i)
                azimuthalAngle = azimuthalGrid(probabilityCell{i}(2));
                elevationAngle = elevationGrid(probabilityCell{i}(3));
                break
            end
        end

    end
end

function outputCell = cellPackager(inputCell)
    outputCell = {};
    for i = 1:length(inputCell)
        for j = 1:length(inputCell{i}{1})
            outputCell = multiAppendCell(outputCell, ...
                matrixExtractor(inputCell{i}{1}{j}));
        end
    end
end

function cellArray = matrixExtractor(s)
    cellArray={};
    nameList = fieldnames(s);
    for i=1:numel(nameList)
        if iscell(s.(nameList{i}))
            cellArray = singleAppendCell(cellArray,s.(nameList{i}));
        elseif isstruct(s.(nameList{i}))
            cellArray = multiAppendCell(cellArray,matrixExtractor(s.(nameList{i})));
        end
    end
end

function cellArrayOut = multiAppendCell(cellArrayIn1,cellArrayIn2)
    cellArrayOut = cellArrayIn1;
    for i=length(cellArrayIn1)+1:length(cellArrayIn1)+length(cellArrayIn2)
        cellArrayOut{i}=cellArrayIn2{i-length(cellArrayIn1)};
    end
end

function cellArray = singleAppendCell(cellArray,matrix)
    cellArray{length(cellArray)+1}=matrix;
end

function outColor = getColor(tier, maxtier)
    brightness = maxtier-tier;
```

```
    midPoint = maxtier/2;
    if brightness < midPoint
        outColor(1) = 255*brightness/midPoint;
        outColor(2:3) = 0;
    else
        outColor(1) = 255;
        outColor(2:3) = 255*(brightness-midPoint)/midPoint;
    end
end
```

6 References

Lawler, G. F. (2018). Introduction to stochastic processes. Chapman & Hall/CRC.
Maskery, S., & Shinbrot, T. (2005). Deterministic and stochastic elements of axonal guidance. Annu. Rev. Biomed. Eng., 7, 187-221.