```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [data,variables,parameters] = MainSingleTask
% Initialise and organise variables that will be used by 'RunModel.m'
%important functions are stored in subfolders
%for the sake of organisation. addpath() adds these folders to the list of
%directories accessed by Matlab.
addpath('./Initialise')
addpath('./GenerateInputPatterns')
addpath('./Helpers')
addpath('./Boxcar')
addpath('./PresetKs')
%{
'parameters' is a struct containing values that do not change;
'variables' is a struct containing values that might change.
 this distinction is arbitrary, and some variables such as theta were
placed in the 'variables' struct just in case they might need to be
manipulated in the future.
'data' is a struct used to store the history of changes in values
 e.g.
data.z_history is a 3 dimensional array that contains the ith neuron's
output in the jth timestep during the kth trial
in the ith row, jth column, kth page.
%}
%{
A word on structs:
As far as the code in this demo is concerned, structs are data structures
that allow variables to be stored hierarchically. To illustrate, they can
be thought of as containers; all values that are contained in these
containers are referred to as fields.
one can store arrays and scalar values in
these structs, but just as a container can contain other subcontainers, it
is possible to use a struct to store another struct. The following code
illustrates.
>> a_struct = struct;
>> a_struct.a_scalar = 3.141582;
>> a_struct.a_matrix = [1,0;0,1];
>> a_struct
a_struct =
 struct with fields:
   a_scalar: 3.1416
   a_matrix: [2×2 double]
>> b_struct = struct;
>> b_struct.b_subfield1 = 2.7183;
>> b_struct.b_subfield2 = [0,0];
>> a_struct.a_field = b_struct;
>> a_struct
a_struct =
 struct with fields:
   a_scalar: 3.1416
```

```
   a_matrix: [2×2 double]
    a_field: [1×1 struct]
>> a_struct.a_field
ans =
 struct with fields:
   b_subfield1: 2.7183
   b_subfield2: [0 0]
%}
%{
all of the values used in the simulation are stored in these structs to
make the code cleaner, and are accessed as fields. For example, if I
wanted to access 'weights_excite' which is a matrix containing the
excitatory weights (for a network using a fanin connectivity scheme in the
case of this demo), I would first note that the excitatory weights are
changed by the network after each timestep and must therefore be in the
'variables' struct. The matrix can be accessed as
'variables.weights_excite'
%}
% tic
disp(datestr(now, 'dd/mm/yy-HH:MM'))
% changes to boxcar_window will result in rescaling of the following:
% desired_mean_z, stutter, epsilon_weight_nmda/spiketiming/feedback,
% decay_nmda/spiketiming, boxcar_refractory_period
%Sets initial values for all parameters. Edit the file for this function to
%make desired changes.
parameters = InitialiseParameters;
%{
%     'boxcar_window'                    , 4,... %default 1
%     'number_of_neurons'                , 401,... %default 1000
%     'consecutive_successes_to_halt'    , 0,...
%     'toggle_figures'                   , true,...
%     'toggle_spiketiming'               , true,... %!!!!!!!
%     'k_ff_start'                       , 0.0066,... %for bx4, use 0.04
%     'desired_mean_z'                   , 0.1,...
%     'epsilon_k_0'                      , 0,... %default 0.5
%     'n_patterns'                       , 4,...
%     'toggle_record_z_train'            , true,...
%     'toggle_record_z_test'             , true,...
%     'toggle_record_k0'                 , true,...
%     'toggle_record_weights'            , false... %record excitatory weights
after each update
%     );
%Initalise Core Vars and Data are run in Initialise_PresetBoxcar
%variables = InitialiseCoreVars(parameters);
%data = InitialiseData(parameters);
% npat = 28; k0 = 0.5; kff = 0.0085; ktable = table(npat,k0,kff); save
PresetKs/ktable ktable timescale
%}
if parameters.toggle_fxn_stopwatch
```

```
   tic
end
seed = parameters.network_construction_seed;
disp(['seed 1 = ',num2str(seed)])
rng(seed)
retrieve_k = false;
parameters.nrn_viewing_range = [1,80];%must be less than number of neurons;
default 700
[parameters,variables,data] = Initialise_PresetBoxcar(parameters,retrieve_k);
%parameters.epsilon_k_0 = 0;
%variables.k_ff = 0.02;
%having prepared all the variables in the 3 structs, the information needed
%for running the simulation is now passed onto the function 'RunModel'
[variables,data] = RunModel(parameters,variables,data);
if parameters.toggle_fxn_stopwatch
   toc
end
save('recentrun')
try
   evalin('base','load(''recentrun.mat'')')
catch
   warning('could not load variables to workspace')
end
end


function parameters = InitialiseParameters(varargin)
%constant parameters
parameters = struct(...
   %{
   'cycle_limit',...
       [10; ... %length of each trial
       100; ... %number of trials
       1; ...
       1],...
   'lpmi',                        1,... loop position of main iteration; i.e.
if external input changes to next time step when inc_vect is [0,1,0] then lpmi
is 2.
   %}
   ... % visualization toggles
   'toggle_figures',              true,...
   'toggle_display_failure_mode',  true,...
   ...
   ... % data recording settings
   'toggle_record_success',        true,...
   'toggle_record_training',       true,...
   'toggle_record_y',              true,... % y and z dimensions: # neurons x
timesteps x trials
   'toggle_record_z_train',        true,...
```

```
   'toggle_record_z_test',          true,...
   'toggle_record_weights_exc',     true,...
   'toggle_record_weights_inh',     true,...
   'toggle_record_k0',              true,...
   'toggle_record_kff',             true,...
   ...
   ... % stopwatch toggle
   'toggle_fxn_stopwatch',          true,... % toggle true to record elapsed
time
   ...
   ... % general settings
   'toggle_training',               true,...
   'toggle_testing' ,               true,...
   'number_of_trials',              140,...
   'consecutive_successes_to_halt',5,... % after this many successes-in-a-row,
halt the program (this saves time); will only run if the previous toggle is on.
   ...
   ... % boxcar settings
   'toggle_divisive_refractory',    true,...
   'boxcar_refractory_period',      4,... %when set to 1 there is no refractory
period
   'boxcar_window',                 1,... %excitation boxcar
   'boxcar_scales',                 [1],... % should sum to 1
   'boxcar_window_inh',             1,...
   'boxcar_scales_inh',             [1],...
   ...
   ... % random number seeds
   'network_construction_seed',     randi(100,1),...
   'z_0_seed',                      randi(100,1),...
   ...
   ... % network topology settings
   'number_of_neurons',             uint16(1000),...
   'connectivity',                  0.1,... %fan in is exact i.e. each postsyn.
nrn is enervated by same number of nrns
   'desired_mean_z',                0.1,...
   ...
   ... % input settings
   'toggle_external_input',         true,...
   'ext_activation',                uint16(30),... %default 30
   'stutter',                       uint16(5),...
   'shift',                         uint16(15),... %default 15
   'n_patterns',                    uint16(24),... %!!!!!!!
   'extra_timesteps_train',         0,... % add this many timesteps with no
external activation to the end of each trial; use test_length_of_each_trial to
add timesteps during testing
   'on_noise',                      0.005,... %probability between 0-1
   'off_noise',                     0.5,... %probability between 0-1
   'test_off_noise',                0.5,...
   'test_on_noise',                 0.0,...
```

```
   ...
   ... % testing parameters
   'test_length_of_each_trial',    43,... %set this equal to stutter * number
of patterns (for simple sequence)
   'first_stimulus_length',        uint16(5),... %in general, set this equal to
stutter * boxcar window
   ...
   ... % trace conditioning
   'trace_interval',               uint16(0),... % (this can be deleted because
user sets trace interval length in genTrace.m)
   'toggle_trace',                 true,... %true trace, false normal;
   ...                                         %If trace is toggled, there are
special options, including noise, success
   ...                                         %See genTrace.m,
genTraceNoise.m, DetermineTrialSuccessTrace.m
   ...
   ... % modification rates
   'toggle_k0_preliminary_mod',    false,...%adjust k0 before training begins
to attune to desired activity
   'toggle_k0_training_mod',       false,...%modify k0 at the end of each trial
to maintain desired activity
   'epsilon_pre_then_post',        0.01,....005,...0.0015
   'epsilon_post_then_pre',        0.01,...
   'epsilon_feedback',             0.5,...
   'epsilon_k_0',                  0.3,... %modification rate used to adapt k0
before and/or during training; make sure appropriate toggles above are turned
on
   'epsilon_k_ff',                 0,...%kff will not be updated unless pre
then post is off
   ...
   ... % synaptic modification settings
   'toggle_pre_then_post',         true,...
   'toggle_post_then_pre',         false,... %!!!!!!!
   'toggle_stutter_e_fold_decay',  false,... %set to false when changing decay
rates
   'fractional_mem_pre',           0.82,... %nmda; used with pre_then_post,
controls Zbar pre decay, -1/log(fractional) = exponential time constant
   'fractional_mem_post',          0.82,... %spiketiming; used with
post_then_pre, controls Zbar post decay
   'offset_pre_then_post',         0,...
   'offset_post_then_pre',         0,...
   ...
   ... % starting values for some variables
   'k_0_start',                    0.8,...
   'k_fb_start',                   0.065,...
   'k_ff_start',                   0.02,... 0.0066 ...
   ...
   ... % weight settings
```

```
   'toggle_rand_weights',          false,... %toggle between random and uniform
excitatory weights
   'weight_start',                 0.4,... %starting value if using uniform
weights
   'weight_high',                  0.2,... %upper limit for weight values if
using random distribution
   'weight_low',                   0.8,... %lower limit for weight values if
using random distribution
   'weight_inhib_start',           1 ... %choose starting weight value for
inhibitory synapse; default 1
   ...
);
if nargin>0
   parameters = WriteToStruct(parameters,varargin{:});
end
parameters = AdjustParameters(parameters);
end
function parameters = AdjustParameters(parameters)
   if parameters.toggle_stutter_e_fold_decay
       stutter = double(parameters.stutter);
       parameters.fractional_mem_pre = exp(-1/(stutter-2));
       parameters.fractional_mem_post = exp(-1/(stutter-2));
   end

   if parameters.toggle_trace == true
       % User must manually set trial length here (see genTrace.m)
       parameters.length_of_each_trial = 43;
   else
       % Default (simple sequence): sets trial length = patterns * stutter
       parameters.of_each_trial = parameters.n_patterns * ...
           (parameters.stutter + parameters.trace_interval); %ignore
trace_interval here (not used)
   end
   %network and topology settings
   n_nrns = double(parameters.number_of_neurons);
   parameters.n_fanin = round(n_nrns*parameters.connectivity);
   %set range of neurons that are to be viewed during the simulation
   parameters.nrn_viewing_range = [1,parameters.number_of_neurons];
end

% Purpose: creates a training input sequence to simulate trace conditioning.
% If the user sets toggle_trace=true in InitialiseParameters.m (line 71),
% the function genTrace will be called in InitialiseCoreVars.m (line 6).
% Noise: Noise (on and off) can be adjusted by the user in
% InitialiseParameters.m (lines 61 and 62) for the training trace sequence.
function input_sequence = genTrace(pm)
% GENTRACE: this function creates a training sequence pattern starting with
% conditioned stimulus neurons turned on for a specified duration of time,
```

```
% followed by a trace interval (specified duration of time with no firing
neurons),
% ending with unconditioned stimulus neurons (positioned orthogonal to the
% CS neurons) turned on for a specified duration of time
% PARAMETERS: takes in parameters from InitialiseParameters.m and requires
% users to set additional parameters (cs_nrns, cs_duration, us_nrns,
% us_duration)
% RETURN: returns the training trace conditioning sequence (without noise)
% Global Variables: These variables are accessed in other files in this
% workspace (genNoiseTrace.m, DetermineTrialSuccessTrace.m).
% It is important to note that when global variable values are
% altered, they are altered in all files.
global cs_nrns;
global us_nrns;
global cs_duration;
global trace_duration;
global off_noise_during_cs;
global off_noise_during_us;
global on_noise_during_cs;
global on_noise_during_trace;
global on_noise_during_us;
global off_noise_during_cs_test;
global on_noise_during_cs_test;
global on_noise_after_cs_test;
global total_success_neurons;
global distinct_success_neurons;
global required_number_of_firings_per_neuron;
global success_timestep_1;
global success_timestep_2;
   %-------------------
   % PARAMETERS
   %-------------------
   % Parameters from InitialiseParameters.m:
   n_nrns          = pm.number_of_neurons;
   timescale       = pm.boxcar_window; %for boxcar adjustments

   % Parameters to be set by user:
   cs_nrns        = 40; %number of conditioned stiumulus neurons
   us_nrns        = 40; %number of unconditioned stiumulus neurons
   cs_duration    = 5; %number of timesteps conditioned stimulus neurons are on
   trace_duration = 30; %number of timesteps of the trace interval
   us_duration    = 8; %number of timesteps unconditioned stimulus neurons are
on
   extra_steps    = 0; %padding zeros at the end of training sequence
   len_trial      = cs_duration + trace_duration + us_duration + extra_steps;
%33

   % Noise options (make sure probabilities are set for on_noise
   % and off_noise in InitialiseParameters.m)
```

```
   % There are multiple noise options because there are three different
   % stages in a training trial and two different in the test trial

       % training noises
   off_noise_during_cs     = true;
   off_noise_during_us     = true;
   on_noise_during_cs      = false; %cs neurons not eligible
   on_noise_during_trace   = true; %option to turn on noise on only during
trace interval. Make sure on_noise is a value between 0-1 in
InitialiseParameters.
   on_noise_during_us      = false; %us neurons not eligible

       % test noises
   off_noise_during_cs_test = true;
   on_noise_during_cs_test  = false; %cs neurons not eligible
   on_noise_after_cs_test   = false; %all neurons are eligible


   % Success parameters to be set by user:
   total_success_neurons    = 0; %total number of neurons within a temporal
window for a successful prediction
   distinct_success_neurons = 12; %distinct number of neurons within a temporal
window for a successful prediction
   required_number_of_firings_per_neuron = 2; %required number of times a
distinct neuron fires (does not have to be sequential)
   % Temporal window in which prediction neurons need to fire
   % Prediction too early is in a moving window that precedes the successful
prediction window
   success_timestep_1   = 26; %timestep at which prediction window begins
   success_timestep_2   = 33; %last timestep of the prediction window

   toggle_produce_visualisation_of_input = true;
   toggle_save_input_sequence_to_file    = false;


   % BOXCAR RESCALE
%     cs_duration = cs_duration * timescale;
%     trace_duration = trace_duration * timescale;
%     us_duration = us_duration * timescale;
%     extra_steps = extra_steps * timescale;
%     len_trial = len_trial * timescale;

   %--------------------------------
   % Function Preconditions
   %--------------------------------
   if n_nrns < (cs_nrns + us_nrns)
       disp('n_nrns must be greater than or equal to cs_nrns + us_nrns. Try
increasing n_nrns, decreasing cs_nrns, or decreasing us_nrns')
   end
```

```
   if success_timestep_2 > (cs_duration + trace_duration)
       disp('the user-defined window of a successful prediction should not
overlap with unconditioned simulus neurons firing. Check and/or decrease
success_timestep_2.');
   end
   if len_trial ~= pm.length_of_each_trial
       error(['Check InitialiseParameters.m to ensure
parameter.length_of_each_trial is correct. Trial length should equal
CS_duration + trace_duration + US_duration + extra_steps. The correct length is
', num2str(len_trial), '. Find or Ctrl+F: parameter.length_of_each_trial']);
   end
   if len_trial ~= pm.test_length_of_each_trial
       error(['Check InitialiseParameters.m to ensure test_length_of_each_trial
is correct. The test trial length should equal CS_duration + trace_duration +
US_duration + extra_steps. The correct length is ', num2str(len_trial), '. Find
or Ctrl+F: test_length_of_each_trial']);
   end
   if cs_duration ~= pm.first_stimulus_length
       error('Check InitialiseParameters.m under testing to ensure the first
stimulus length is correct. first_stimulus_length should equal cs_duration.
Find or Ctrl+F: first_stimulus_length');
   end
   %------------------------------------------------
   % Implementation (calls function genPrototypes)
   %------------------------------------------------
   input_sequence =
genPrototypes(n_nrns,len_trial,cs_nrns,cs_duration,trace_duration, us_nrns,
us_duration);
   if extra_steps > 0
       % padding of zeros at the end
       input_sequence = [input_sequence, zeros(n_nrns,extra_steps)];
   end

   %--------------------------------
   % Visualize Sequence
   %--------------------------------
    if toggle_produce_visualisation_of_input == true
       spy(input_sequence)
    end

   %--------------------------------
   % Save Sequence
   %--------------------------------
   if toggle_save_input_sequence_to_file == true
       save('Input.mat','input_sequence')
   end
end
```

```
function prototypes =
genPrototypes(n_nrns,len_trial,cs_nrns,cs_duration,trace_duration, us_nrns,
us_duration)
%GENPROTOTYPES: This function does the main implementation of the trace
%conditioning sequence. It is called by genTrace.

   % Initialize an empty matrix with dimensions number of neurons by
   % length of trial
   prototypes = zeros(n_nrns, len_trial);

   % Conditioned Stimulus neurons:
   % A matrix representing the firing neurons between the conditioned stimulus
onset
   % and conditioned stimulus offset
   cs_block = ones(cs_nrns, cs_duration);

   % Unconditioned Stimulus neurons:
   % A matrix representing the firing neurons between the unconditioned
   % stimulus onset and uncondition stimulus offset
   us_block = ones(us_nrns, us_duration);

   % these variables act as moving pointers to indicate
   % where the next pattern of firing neurons is located
   initial_cs_nrn = 1;
   final_cs_nrn = cs_nrns;
   start = 1;
   stop = cs_duration;


   % --- Main Training Sequence Generator ---
   % Fills in CS neurons
   % x-axis (start:stop) is the duration of CS neurons firing
   % y-axis (initial_cs_nrn:final_cs_nrn) are the neurons firing
   prototypes(initial_cs_nrn:final_cs_nrn, start:stop) = cs_block;
   % Trace Interval
   % Sets up start and stop for for next pattern of US neurons (along x-axis)
   start = start + cs_duration + trace_duration;
   stop = stop + trace_duration + us_duration;
   % Sets up US neurons firing location (along y-axis)
   initial_us_nrn = final_cs_nrn + 1;
   final_us_nrn = initial_us_nrn + us_nrns - 1;
   % Fill in US neurons
   % x-axis (start:stop) is the duration of US neurons firing
   % y-axis (initial_us_nrn:final_us_nrn) are the neurons firing
   prototypes(initial_us_nrn:final_us_nrn, start:stop) = us_block;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%added option of halting the program after a configurable number of
%successes in a row, and included data.success which is a boolean vector
%that records the trials that are successful
function [vars,data] = RunModel(param,vars,data)
   %nested loop lengths are determined by the number of trials and the length
   %of each trial
   disp(param)

   seed = param.z_0_seed;
   disp(['seed 2 = ', num2str(seed)]);
   rng(seed)
   n_trials = param.number_of_trials;
   len_trial = param.length_of_each_trial;

   data.successful_learning = false;

   if param.toggle_trace == true && param.toggle_record_success == true %
success data for trace conditioning
       data.successful_predictions = 0;
       data.prediction_too_soon = 0;
       data.prediction_too_late = 0;
       data.failure_to_predict = 0;
       data.list_of_results = strings(n_trials,1);
   end

   if param.toggle_record_success > 0
       n_consec_successes = 0;
   end
   %tic %starts stopwatch, to measure time it takes to run through all trials

   for trial_number = 1:n_trials
       [vars,data] = InitialiseTrial(param,vars,data);
       for timestep = 1:len_trial
           %update variables for the current timestep
           vars = UpdateSpike(param,vars,timestep);
           vars = UpdateWeights(param,vars);
           %check for errors
           CatchNegativeValues(vars)
           %track changes in variables
           vars = RecordMeanZ(vars,timestep);
           data = RecordTimestep(vars,data,timestep);
       end

       vars = UpdateK0(param,vars);

       %Record variables of interest into the 'data' struct.
       if param.toggle_record_training == true
           data = RecordTrial(param,vars,data,trial_number);
       end
```

```
        %Keeping the current weights constant, let the network
        %run on its own and see if it is able to remember the input sequence.
        if param.toggle_testing == true
            data = TestNetwork(param,vars,data,trial_number);
        end
        if param.epsilon_k_ff > 0 && (param.toggle_pre_then_post == false || ...
                param.epsilon_pre_then_post == false)
            vars = UpdateKff(param,vars,data); %kff will not be updated unless
pre then post is off
        end
        % Simple sequence success
        if param.toggle_record_success == true && param.toggle_trace == false
            current_trial_success =
DetermineTrialSuccess(param,vars,data,trial_number);
            [data,n_consec_successes] =...

RecordSuccess(param,data,current_trial_success,n_consec_successes,trial_number)
;
            if param.consecutive_successes_to_halt > 0
                if n_consec_successes >= param.consecutive_successes_to_halt
                    disp([num2str(param.n_patterns),' success at trial
',num2str(trial_number)])
                    data.successful_learning = true;
                    break
                end
            end
        end

        % Trace sequence success
        if param.toggle_record_success == true && param.toggle_trace == true
            current_trial_success = DetermineTrialSuccessTrace(param, data,
trial_number);
            success_boolean = 0;
            % for programmer viewing (probably change to display on spy)
            disp(['trial ', num2str(trial_number),': ', current_trial_success]);
            data.list_of_results(trial_number) = current_trial_success;

            if strcmp(current_trial_success, 'success')
                data.successful_predictions = data.successful_predictions + 1;
                success_boolean = 1;
            end

            [data,n_consec_successes] =...

RecordSuccessTrace(param,data,success_boolean,n_consec_successes,trial_number);

            if strcmp(current_trial_success, 'failure - prediction too soon')
                data.prediction_too_soon = data.prediction_too_soon + 1;
            end
```

```
            if strcmp(current_trial_success, 'failure - prediction too late')
                data.prediction_too_late = data.prediction_too_late + 1;
            end

            if strcmp(current_trial_success, 'failure to predict')
                data.failure_to_predict = data.failure_to_predict + 1;
            end

            if param.toggle_display_failure_mode == true
                xlabel(current_trial_success);
            end

            % stop after hitting x number of consecutive successes
            % (defined in InitialiseParameters.m)
            if n_consec_successes >= param.consecutive_successes_to_halt
                disp(['success at trial ',num2str(trial_number)])
                data.successful_learning = true;
                break
            end
            %waitforbuttonpress

        end

    end

    if true
        pause(0.0000000000000001)
    end
end
%time_taken = toc;
%disp(['Runtime: ', num2str(time_taken), ' seconds'])
%% Subfunctions %%
%{
(When viewing on Matlab, collapsing parts of the script by clicking on the
boxed minus sign to the left of 'function' in blue letters helps remove
the clutter and clarify how the functions are organised)
%}
%%%%%%%%%%%%%%%%%%%%%% Training %%%%%%%%%%%%%%%%%%%%%%
function [vars,data] = InitialiseTrial(param,vars,data)
    %1. introduces noise to the prototype input sequence;
    %2. sets z to a random vector with a mean z that is roughly equal to
    %the desired average spike probability (i.e. equal to vars.desired_mean_z).
        %The entries of the random vector are chosen by
        %generating a number on the uniform distribution between 0 and 1,
        %then checking whether that number is less than the
        %desired_mean_z, which is also a number between 0 and 1.
        %This event has a probability equal to the desired mean z
    %3. reset the decay variables back to zero
```

```
        %Note: there are scenarios in which the network might not stop at the
        %last pattern, going back to the first and cycling through the sequence
        %again during a test run. This happens when the decay variables or
        %the z_prev variable are not reset to zero at the beginning of each
        %training trial, in which case the network will believe that the first
        %pattern of a sequence comes immediately after the last pattern of the
        %preceding trial and create an association between the two.
    vars.z_prev     = zeros(size(vars.z_prev));
    vars.boxcar_exc = ones(size(vars.boxcar_exc)) .*
mean(mean(vars.weights_excite))...
        * param.desired_mean_z * param.n_fanin;
    vars.boxcar_inh = ones(size(vars.boxcar_inh)) * param.desired_mean_z...
        * param.weight_inhib_start;
    if param.toggle_trace == true
         vars.input_current_trial =
genNoiseTrace(vars.input_prototypes,param.on_noise,param.off_noise);%trace
conditioning
    else
        vars.input_current_trial =
genNoise(vars.input_prototypes,param.on_noise,param.off_noise); %simple
sequence
    end
    vars.z = rand(param.number_of_neurons,1) < param.desired_mean_z; % z_0 :
initialize random z vector at time 0
    vars.z_pre_then_post_decay = zeros(param.number_of_neurons,1);
    vars.z_post_then_pre_decay = zeros(param.number_of_neurons,1);
    %use if testing the network's ability to control its activity;
    %this turns off external inputs and lets the network run on its own.
    if param.toggle_external_input == 0
        vars.input_current_trial=zeros(size(vars.input_prototypes));
    end
end
function vars = UpdateSpike(param,vars,timestep)
    %if error indicates 'index in position 2 is invalid', check if t is -1.
    %if it is, then there is a problem with the function triggers. Refer to
    %ControlFxnTriggerIdx
    %% Get Inputs:
    %assign variables from the cv struct into variables accessed directly
    %by this function
    x                = vars.input_current_trial(:,timestep);
    z                = vars.z;
    connection_mat   = vars.connections_fanin;
    k_0              = vars.k_0;
    k_fb             = vars.k_fb;
    k_ff             = vars.k_ff;
    w_fbinhib        = vars.weights_feedback_inhib;
    w_excite         = vars.weights_excite;
    boxcar_exc       = vars.boxcar_exc;
    boxcar_inh       = vars.boxcar_inh;
```

```
    exc_scales       = vars.boxcar_scales;
    inh_scales       = vars.boxcar_scales_inh;
    refractory       = vars.boxcar_refractory_period;

    %% update Spike at timestep t
    %as the output vector z hasn't been updated yet for the current timestep,
    %it still corresponds to the activation of the previous timestep.
    z_prev = z;

    %recall that connection_mat is a fanin connection matrix. Therefore,
    %every ith row vector specifies the indices of the presynaptic neurons
    %that ennervate the ith neuron.
    %z(connection_mat) is a matrix the rearranges the output z into a matrix
    %corresponding to connection_mat, such that every entry on the ith row
    %indicates which presynaptic neurons that ennervate the ith
    %neuron have fired.
    %
    %the excitatory weights (w_excite) are organised according to the same
    %fanin connection scheme, so an element by element multiplication '.*'
    %of the rearranged output matrix and weights as done below should indicate
    %(on every ith row) the individual presynaptic activations that take place
    %before being summed up postsynaptically by the ith neuron.
    Wz_exc = sum(z(connection_mat) .* w_excite,2);
    %{
    %push each boxcar by 1 column to the right and replace the first column
    %of the boxcar matrix with Wz_exc
    boxcar_exc = PushBoxcar(boxcar_exc, Wz_exc); %matrix is in M_{n_nrns\times
n_boxcars}
    %sum boxcar terms according to the distribution specified in boxcar_pmf
    excite = boxcar_exc*boxcar_pmf'; %boxcar_pmf is a vector in
\mathbb{R}^n_boxcars
    %}
    %speeded up the code

    %returns total excitation(scalar) and new boxcar window of Wz values(matrix)
    [excite,boxcar_exc] = BoxcarStep(Wz_exc, boxcar_exc, exc_scales);

    %{
    %function [excite,boxcar_exc] = BoxcarStep(Wz_exc, boxcar_exc, boxcar_pmf,
timestep)
    t_modulo = mod(timestep,numel(boxcar_pmf));
    boxcar_exc(:,t_modulo) = Wz_exc;
    excite = boxcar_exc * boxcar_pmf([t_modulo:end,1:t_modulo-1])';
    %}
    %the same boxcar algorithm is used to calculate inhibition
    Wz_inh = sum(w_fbinhib .* z_prev);
    %boxcar_inh = PushBoxcar(boxcar_inh, Wz_inh);
    %fb_inh=boxcar_inh*boxcar_pmf';
    [inhib_fb,boxcar_inh] = BoxcarStep(Wz_inh, boxcar_inh, inh_scales);
```

```
   %The following inhibition equation is based on Dave's rule.
   %k_0 ensures that the mean_z activity of the network equals the desired
   %   mean_z specified. This is done at the end of each trial by
   %   increasing k_0, and therefore inhibition, whenever there is too much
   %   activity, and decreasing it when there is too little.
   %w_fbinhib, which stands for feedback inhibitory weights, corresponds to
   %   the inhibition based on the activity of individual neurons;
   %k_ff*sum(x), just as k_0, controls overall activity,
   %   but is based on external inputs and is therefore set to 0 when testing
   %   (as the external stimulus is turned on only when training)
   inhib = k_0    +    k_fb*(inhib_fb)    +    k_ff*sum(x);
   %the y postsynaptic activation vector is calculated based
   %on the excitation and inhibition vectors
   y = excite./(excite+inhib);
   %each neuron spikes if its activation is greater than the spike_threshold
   %(which is arbitrarily set to 0.5 in this model)
   z = y>vars.spike_threshold | x == 1;
   if refractory > 0 ...    %if refractory period is nonzero
           && length(exc_scales)>refractory       %and is well defined
       switch param.toggle_divisive_refractory
           case false
               boxcar_exc =
BoxcarRefractory(z,boxcar_exc,refractory,inhib,exc_scales,timestep);
               %subtracts the appropriate amount from the parts of the boxcar
               %corresponding to the refractory value, thereby preventing
               %the neuron from firing during the refractory period
           case true
               [boxcar_exc] = BoxcarRefractory2(z,boxcar_exc,refractory);
               %divides the boxcar by the refractory value
       end

   end
   %% Output
   %assign output to the struct
   vars.boxcar_exc = boxcar_exc;
   vars.boxcar_inh = boxcar_inh;
   vars.z_prev = PushBoxcar(vars.z_prev,z_prev); %vars.z_prev is a matrix of
previous values; insert most recent z_prev in left column and push all other
columns to the right (deleting the rightmost column)
   vars.z = z;
   vars.y = y;
end
function vars = UpdateWeights(params,vars)
   %% retrieve inputs from struct
   fractional_mem_pre       = params.fractional_mem_pre;
   fractional_mem_post      = params.fractional_mem_post;
   e_pre_then_post          = params.epsilon_pre_then_post;
   e_post_then_pre          = params.epsilon_post_then_pre;%st stands for spike
timing
```

```
    e_fb                        = params.epsilon_feedback;%fb stands for feedback
    w_excite                    = vars.weights_excite;
    connection_mat              = vars.connections_fanin;
    z_pre_then_post_decay       = vars.z_pre_then_post_decay;
    z_post_then_pre_decay       = vars.z_post_then_pre_decay;
    z                           = vars.z;
    w_fbinhib                   = vars.weights_feedback_inhib;

    z_memory                    = [z vars.z_prev]; %array of all stored z values;
allows z bar to access current and past timesteps
    z_offset_pre                = z_memory(:,params.offset_pre_then_post + 2);
%always looks at least 1 timestep back
    z_offset_post               = z_memory(:,params.offset_post_then_pre + 1);
    z_prev                      = vars.z_prev(:,1);

    mean_z                      = vars.mean_z;
    desired_mean_z              = params.desired_mean_z;
    %% update weights
    %weights for feedback excitation
    if params.toggle_pre_then_post == true
        %(z_prev == true/false) can be seen as an if statement;
        %e.g.
        %>> (z_prev==1) + (z_prev == 0).*(decay_nmda.*z_nmda_decay)
        %can be interpreted as
        %   if (the previous output did fire), then return 1;
        %   if (the previous output did not fire),
        %       then return decay_nmda.*z_nmda_decay
        %           (i.e. make the current value in z_nmda decay)
        %This method was used instead of for loops because it allows such
        %conditional operations to be executed on the entire vector
        %all at once (taking advantage of Matlab's parallel operations on
arrays),
        %which is much faster than looping through each vector entry.
        z_pre_then_post_decay = (z_offset_pre==1)+(z_offset_pre==0).* ...
            (fractional_mem_pre.*z_pre_then_post_decay); %saturate decay rate
        dw_excite =
e_pre_then_post*z.*(z_pre_then_post_decay(connection_mat)-w_excite);
        w_excite = w_excite+dw_excite;
    end
    if params.toggle_post_then_pre == true
        % if there is a spike at the current output, rise to 1,
        % otherwise make the current value decay.
        z_post_then_pre_decay = (z_offset_post==1) + (z_offset_post==0) ...
            .* (fractional_mem_post .* z_post_then_pre_decay); %saturate decay
        dw_excite = -e_post_then_pre .* z_prev(connection_mat) .*
z_post_then_pre_decay .* w_excite;
        w_excite = w_excite + dw_excite;
        % Old version:
```

```
        % z_post_then_pre_decay = (z==1) + (z==0) .* (fractional_mem_post .*
z_post_then_pre_decay); %saturate decay
        % dw_excite = -e_post_then_pre .* z(connection_mat) .*
z_post_then_pre_decay .* w_excite;
   end
   %update weights for feedback inhibition (aka Dave's rule)
   dw_fbinhib = e_fb* z_prev .* (mean_z - desired_mean_z);
   weights_feedback_inhib = w_fbinhib + dw_fbinhib;
   %% write outputs to struct
   vars.weights_excite = w_excite;
   vars.weights_feedback_inhib = weights_feedback_inhib;
   vars.z_pre_then_post_decay = z_pre_then_post_decay;
   vars.z_post_then_pre_decay = z_post_then_pre_decay;
end
function vars = UpdateK0(params,vars)
   %% get inputs from structs
   e_k_0            = params.epsilon_k_0; %rate constant epsilon for k_0
   k_0              = vars.k_0;
   mean_z_train     = vars.mean_z_train_current_trial;
   desired_mean_z   = params.desired_mean_z;
   trial_mean_z     = mean(mean_z_train);
   %^ this is the average spike rate of all neurons throughout the entire
   %trial
   %% update k_0
   %dk_0=e_k_0*(desired_mean_z - trial_mean_z);
   dk_0=e_k_0*(trial_mean_z - desired_mean_z);
   k_0=k_0+dk_0;
   %% record output to struct
   vars.k_0 = k_0;
   vars.mean_z_current_trial = trial_mean_z;
end
function vars = UpdateKff(params,vars,data)
   %% get inputs from structs
   trial_mean_z     = vars.mean_z_current_trial;
   e_k_ff           = params.epsilon_k_ff;
   k_ff             = vars.k_ff;
   z_test           = data.z_test_current_trial;
   test_mean_z      = mean(mean(z_test));
   %vars for k_0 compensation
   %k_0              = vars.k_0;
   %n_inputs           = params.n_active_nrns;
   %off_noise        = params.off_noise;
   %% update k_ff
   dk_ff = e_k_ff*(trial_mean_z-test_mean_z);
   k_ff = k_ff+dk_ff;
   %% compensate k_0
   %Note that as the contribution of feedforward inhibition is
   %k_ff*n_ext_nrns, to keep the total inhibition constant we need to subtract
   %this feedforward inhibition contribution from k_0.
```

```
    %n ext nrns that are active per timestep
    %n_ext_nrns = double(n_inputs-off_noise);
    %k_0 = k_0 - n_ext_nrns*dk_ff;
    %% record output to struct
    vars.k_ff = k_ff;
    %vars.k_0 = k_0;
end
%%%%%%%%%%%%%%%%%%%%%% Post training %%%%%%%%%%%%%%%%%%%%%%
function CatchNegativeValues(vars)
    if vars.k_0<0
        disp(vars)
        vars.k_0=0;
        warning('k_0 < 0; not biologically meaningful. To resolve, decrease k_fb
or k_ff')
    elseif prod(vars.weights_excite>=0) == 0 %error if any weight is negative
        warning('there is a negative synaptic weight; not biologically
meaningful')
    end
end
function vars = RecordMeanZ(vars,timestep)
    vars.mean_z=mean(vars.z);
    vars.mean_z_train_current_trial(timestep)=vars.mean_z;
end
function data = RecordTimestep(vars,data,timestep)
    data.z_train_last_trial(:,timestep) = vars.z;
    data.y_train_last_trial(:,timestep) = vars.y;
end
function data = RecordTrial(params,vars,data,trial_number)
    current_trial = data.z_train_last_trial;
    if params.toggle_record_z_train == true
        data.z_train(:,:,trial_number) = current_trial;
    end
    if params.toggle_record_y
        data.y_train(:,:,trial_number) = data.y_train_last_trial;
    end
    %spy(current_trial)
    if params.toggle_figures == true

        subplot(1,2,1)

        SpySelected(current_trial,params.nrn_viewing_range)

    end
    title(['training trial ', num2str(trial_number)])
    xlabel(['training mean z = ', num2str(vars.mean_z_current_trial)])
    pause(0.000000000001)
    %%added part:
    data.mean_z_train_all_trials(:,trial_number) =
vars.mean_z_train_current_trial;
```

```
    data.mean_z_train_last_trial(:,:) = vars.mean_z_train_current_trial;

    if params.toggle_record_k0 == true
        data.k0_history(trial_number)       = vars.k_0;
    end
    if params.toggle_record_kff == true
        data.kff_history(trial_number)      = vars.k_ff;
    end
    if params.toggle_record_weights_exc == true && ...
            params.epsilon_pre_then_post > 0 && ...
            params.toggle_pre_then_post == true
        data.w_excite_trial(:,:,trial_number) = vars.weights_excite;
    end
    if params.toggle_record_weights_inh == true
        data.w_inhib_trial(:,trial_number) = vars.weights_feedback_inhib;
    end
end
function data = TestNetwork(params,vars,data,trial_number)
    %make sure cv is not an output variable of the function
    [vars,~] = InitialiseTrial(params,vars,000); %the 000 is just a stub
    len_trial = params.test_length_of_each_trial;
    n_nrns    = params.number_of_neurons;

    on_noise = double(params.test_on_noise);
    off_noise = double(params.test_off_noise);
    first_stimulus_no_noise =
vars.input_prototypes(:,1:params.first_stimulus_length); %makes the external
test activation noisy
    first_stimulus = genNoise(first_stimulus_no_noise, on_noise, off_noise);

    vars.input_current_trial = zeros(n_nrns,len_trial,'logical');
%     vars.input_current_trial =
zeros(size(vars.input_current_trial),'logical');
    vars.input_current_trial(:,1:params.first_stimulus_length) = first_stimulus;
     current_test = zeros(n_nrns,len_trial,'logical');
    for timestep = 1:len_trial
        vars = UpdateSpike(params,vars,timestep);
        if params.toggle_trace == true
            %vars.z = genNoiseTrace(vars.z,on_noise,off_noise); %apply noise to
test trace sequence
        else
            vars.z = genNoise(vars.z,on_noise,off_noise); %apply noise to test
simple sequence
        end
        current_test(:,timestep) = vars.z;
    end
    %spy(current_test)
    if params.toggle_figures ==  true
```

```
        subplot(1,2,2)
        SpySelected(current_test,params.nrn_viewing_range)
        %figure
        %plot(data.z_train_last_trial)

    end
    if params.toggle_record_success == true && params.toggle_trace == true
%trace : display success lines
        global success_timestep_1;
        global success_timestep_2;
        global cs_duration;
        global trace_duration;
        start_of_us_nrns = cs_duration + trace_duration + 1; %timestep of US
onset
        xline(success_timestep_1);
        xline(success_timestep_2);
        xline(start_of_us_nrns, '-','US onset');
    end

    title({['test using weights from trial ',num2str(trial_number)],...
            ['mean',num2str(mean(mean(current_test)))]})
    pause(0.00000000000001)
    data.z_test_last_trial = current_test;
    data.mean_z_test_last_trial = mean(current_test)';
    if params.toggle_record_z_test==true
        data.z_test(:,:,trial_number) = current_test;
    end
end
function [data,n_consec_successes] =
RecordSuccess(params,data,result_code,n_consec_successes,trial_number)
    data.success_vect(trial_number) = result_code;
    success_code = FailureModeDictionary('success');
    n_consec_successes = (n_consec_successes+(result_code ==
success_code))*(result_code>0);
    data.max_consecutive_successes =
max(n_consec_successes,data.max_consecutive_successes);
    if params.toggle_display_failure_mode == true
        failmode_msg = [char(FailureModeDictionary(result_code)),
num2str(result_code)];
        xlabel(failmode_msg)
        pause(0.0000000000000001)
    end
end
function [data, n_consec_successes] =
RecordSuccessTrace(params,data,trace_result_code,n_consec_successes,trial_numbe
r)
    data.success_vect(trial_number) = trace_result_code;
    success_code = 1;
```

```
   n_consec_successes = (n_consec_successes+(trace_result_code ==
success_code))*(trace_result_code>0);
end
function [k0,kff,wfbinhib] = PresetK(parameters,variables,data)
   parameters.toggle_figures = true;
   parameters.toggle_record_success = false;
   parameters.toggle_testing = false;
   parameters.toggle_record_k0 = true;
   %parameters.toggle_testing           = false;
   %parameters.toggle_record_training   = false;
   %parameters.toggle_figures           = false;
   parameters.number_of_trials         = ceil(parameters.number_of_trials/2);
   parameters.toggle_post_then_pre     = false;
   parameters.toggle_nmda              = false;
   parameters.toggle_k0_training_mod = true;
   %parameters.epsilon_k_0  = 1;
   parameters.epsilon_weight_feedback = 0;
   %parameters.epsilon_k_ff = 0.01;
   %parameters.stutter = 0; %somehow controls the duration of external input
firings during test trials
   if parameters.epsilon_k_ff == 0
       disp('kff modification turned off')
   end
   [variables,~] = RunModel(parameters,variables,data);
   k0 = variables.k_0;
   kff = variables.k_ff;
   wfbinhib = variables.weights_feedback_inhib;
end
```