SRNN Reservoir Code Structure Documentation

scripts/setup_paths.m

Purpose

`setup_paths.m` lives in the `scripts/` directory and is the one-stop helper for adding the sibling `src/` tree to the MATLAB path. Call `scripts/setup_paths` (or `setup_paths` from the same directory) before running any script that needs the shared `src` utilities so they can resolve without hardcoded `addpath` calls.

Behavior

- locates the repository root relative to `scripts/`
- ensures `<root>/src/` exists and errors otherwise
- adds `src/` (recursively) to the MATLAB path, so downstream scripts can rely on the `src` functions without having to replicate the path-setup logic

SRNN_reservoir.m

Overview

`SRNN_reservoir.m` implements a rate network with spike-frequency adaptation (SFA) and short-term synaptic depression (STD) for use with MATLAB's ODE solvers (e.g., `ode45`). The function computes the time derivatives of all state variables according to the specified dynamical equations.

Function Signature

```
function [dS_dt] = SRNN_reservoir(t, S, t_ex, u_ex, params)
```

Mathematical Model

The function implements the following system of differential equations:

$$\dot{x}_i = \frac{-x_i + u_i + \sum_{j=1}^{J} w_{ij}\, b_j r_j}{\tau_d}$$

$$r_i = \phi\left(x_i - a_{0_i} - c\sum_{k=1}^{K} a_{ik}\right)$$

$$\dot{a}_{ik} = \frac{-a_{ik} + r_i}{\tau_k}$$

$$\dot{b}_i = \frac{1 - b_i}{\tau_{rec}} - \frac{b_i\, r_i}{\tau_{rel}}$$

Where: - `x_i`: Dendritic state of neuron i - `r_i`: Firing rate of neuron i - `a_{i,k}`: k-th adaptation variable for neuron i - `b_i`: Short-term depression variable for neuron i (0 < b_i ≤ 1) - `activation_function`: Nonlinear activation function (e.g., sigmoid, tanh) - `c_E`, `c_I`: Adaptation scaling factor for E/I neurons - `W(i,j)`: Connection weight from neuron j to neuron i - `u_ex`: External input matrix (n x nt) - `tau_d`: Dendritic time constant (scalar) - `tau_a_E`, `tau_a_I`: Adaptation time constant vectors (1 x n_a_E, 1 x n_a_I) - `tau_b_E_rec`, `tau_b_I_rec`: STD recovery time constant - `tau_b_E_rel`, `tau_b_I_rel`: STD release time constant

---

Input Arguments

`t` (scalar)

Current time point requested by the ODE solver.

`S` ($N\_sys\_eqs \times 1$ column vector)

State vector containing all dynamic variables. **Total size**: `N_sys_eqs = n_E*n_a_E + n_I*n_a_I + n_E*n_b_E + n_I*n_b_I + n`

Organization: `S = [a_E(:); a_I(:); b_E(:); b_I(:); x(:)]`

`t_ex` (nt $\times$ 1 column vector)

Time vector for external input, where `nt` is the number of time points.

`u_ex` (n $\times$ nt matrix)

External input stimulus matrix, where each column corresponds to a time point in `t_ex`.

`params` (struct)

Parameter structure containing all network parameters (see detailed breakdown below).

Output

`dS_dt` ($N\_sys\_eqs \times 1$ column vector)

Time derivatives of all state variables, organized identically to `S`.

Organization: `dS_dt = [da_E_dt(:); da_I_dt(:); db_E_dt(:); db_I_dt(:); dx_dt]`

State Vector Organization

Full State Vector S

```
S = [a_E(:);      % Excitatory adaptation states (n_E * n_a_E × 1)
     a_I(:);      % Inhibitory adaptation states (n_I * n_a_I × 1)
     b_E(:);      % Excitatory STD states (n_E * n_b_E × 1)
     b_I(:);      % Inhibitory STD states (n_I * n_b_I × 1)
     x]           % Dendritic states for all neurons (n × 1)
```

Unpacked State Variables

`a_E` ($n\_E \times n\_a\_E$ matrix, or empty if $n\_a\_E = 0$) Adaptation variables for excitatory neurons. - **Rows**: Individual excitatory neurons (indexed 1 to n_E) - **Columns**: Different adaptation time constants (indexed 1 to n_a_E) - **Example**: `a_E(i, k)` is the k-th adaptation variable for the i-th excitatory neuron

`a_I` ($n\_I \times n\_a\_I$ matrix, or empty if $n\_a\_I = 0$) Adaptation variables for inhibitory neurons. - **Rows**: Individual inhibitory neurons (indexed 1 to n_I) - **Columns**: Different adaptation time constants (indexed 1 to n_a_I) - **Example**: `a_I(i, k)` is the k-th adaptation variable for the i-th inhibitory neuron

`b_E` ($n\_E \times n\_b\_E$ column vector, or empty if $n\_b\_E = 0$) Short-term depression variables for excitatory neurons. - **Size**: $n\_E \times n\_b\_E$ (typically $n\_b\_E = 1$, so this is $n\_E \times 1$) - **Range**: $0 < b\_E(i) \leq 1$, where $b\_E(i) = 1$ means no depression - **Interpretation**: Multiplicative scaling of firing rate due to synaptic depression - **Dynamics**: b recovers toward 1 with time constant `tau_b_E_rec/tau_b_I_rec`, and decreases with firing rate r with time constant `tau_b_E_rel/tau_b_I_rel`

`b_I` ($n\_I \times n\_b\_I$ column vector, or empty if $n\_b\_I = 0$) Short-term depression variables for inhibitory neurons. - **Size**: $n\_I \times n\_b\_I$ (typically $n\_b\_I = 0$ or 1) - **Range**: $0 < b\_I(i) \leq 1$, where $b\_I(i) = 1$ means no depression - **Interpretation**: Multiplicative scaling of firing rate due to synaptic depression - **Note**: Often set to $n\_b\_I = 0$ since inhibitory synapses typically show less depression than excitatory synapses

x (n × 1 column vector) Dendritic states for all neurons (both excitatory and inhibitory). - **Size**: n = n_E + n_I - **Indexing**: First n_E elements are excitatory, next n_I elements are inhibitory

---

Parameters Structure (`params`)

Network Size Parameters

Field	Type	Size	Description
`n`	scalar	1×1	Total number of neurons (n = n_E + n_I)
`n_E`	scalar	1×1	Number of excitatory neurons
`n_I`	scalar	1×1	Number of inhibitory neurons
`n_a_E`	scalar	1×1	Number of adaptation time constants for E neurons (0 to disable)
`n_a_I`	scalar	1×1	Number of adaptation time constants for I neurons (0 to disable)
`n_b_E`	scalar	1×1	Number of STD timescales for E neurons (0 or 1; 0 to disable)
`n_b_I`	scalar	1×1	Number of STD timescales for I neurons (0 or 1; 0 to disable)

Neuron Indices

Field	Type	Size	Description
`E_indices`	vector	n_E×1	Indices of excitatory neurons in the full network
`I_indices`	vector	n_I×1	Indices of inhibitory neurons in the full network

Connectivity

Field	Type	Size	Description
`W`	matrix	n×n	Connection weight matrix. `W(i,j)` is the weight from neuron j to neuron i. Should obey Dale's law (excitatory neurons have non-negative outgoing weights, inhibitory neurons have non-positive outgoing weights).

Time Constants

Field	Type	Size	Description
`tau_d`	scalar	1×1	Dendritic time constant (seconds)
`tau_a_E`	vector	1×n_a_E	Adaptation time constants for E neurons (seconds). Can be empty if n_a_E = 0.
`tau_a_I`	vector	1×n_a_I	Adaptation time constants for I neurons (seconds). Can be empty if n_a_I = 0.
`tau_b_E_rec`	scalar	1×1	STD recovery time constant for E neurons (seconds). Time constant for b to recover toward 1.

Field	Type	Size	Description
`tau_b_E_rel`	scalar	1×1	STD release time constant for E neurons (seconds). Time constant for depression during firing.
`tau_b_I_rec`	scalar	1×1	STD recovery time constant for I neurons (seconds). Only used if n_b_I > 0.
`tau_b_I_rel`	scalar	1×1	STD release time constant for I neurons (seconds). Only used if n_b_I > 0.

Adaptation and STD Scaling Parameters

Field	Type	Size	Description
`c_E`	scalar	1×1	Adaptation scaling for E neurons. Multiplies the sum of adaptation variables. Default: 1.0. Typical range: 0-3.
`c_I`	scalar	1×1	Adaptation scaling for I neurons. Multiplies the sum of adaptation variables. Default: 1.0. Typical range: 0-3.

Activation Function

Field	Type	Size	Description
`activation_function`	function handle	-	**[Required]** Nonlinear activation function. Should accept a vector and return a vector of the same size. Common choices: `@(x) tanh(x)`, `@(x) 1./(1 + exp(-4*x))` (sigmoid), `@(x) max(0, x)` (ReLU).
`activation_function_derivative`	function handle	-	**[Required]** Derivative of `activation_function`. Required for Jacobian computation (Lyapunov analysis). Should accept a vector and return a vector of the same size. **Example**: For tanh, `@(x) 1 - tanh(x).^2`; for sigmoid, `@(x) 4*sigmoid(x).*(1 - sigmoid(x))`.

Internal Variables

Intermediate Computation Variables

`u` (n × 1 column vector) External input at current time `t`, obtained by interpolating `u_ex` at time `t`.

`x_eff` (n × 1 column vector) Effective dendritic potential after subtracting adaptation:

```
x_eff(E_indices) = x(E_indices) - c_E * sum(a_E, 2)  % For E neurons
x_eff(I_indices) = x(I_indices) - c_I * sum(a_I, 2)  % For I neurons
```

`b` (n × 1 column vector) STD variable for all neurons. Initialized to 1 (no depression):

```
b = ones(n, 1);
b(E_indices) = b_E;  % If n_b_E > 0
b(I_indices) = b_I;  % If n_b_I > 0
```

r (n × 1 column vector) Firing rate of all neurons: `r = b .* activation_function(x_eff)` - The b multiplicative factor implements short-term synaptic depression - When b_i = 1 (no depression), firing rate is unaffected - When b_i < 1, firing rate is reduced proportionally

Derivative Variables

dx_dt (n × 1 column vector) Time derivative of dendritic states:

```
dx_dt = (-x + W * r + u) / tau_d
```

da_E_dt (n_E × n_a_E matrix, or empty) Time derivatives of excitatory adaptation variables:

```
da_E_dt = (r(E_indices) - a_E) ./ tau_a_E
```

Uses MATLAB broadcasting: `r(E_indices)` is n_E×1, `a_E` is n_E×n_a_E, `tau_a_E` is 1×n_a_E.

da_I_dt (n_I × n_a_I matrix, or empty) Time derivatives of inhibitory adaptation variables:

```
da_I_dt = (r(I_indices) - a_I) ./ tau_a_I
```

Uses MATLAB broadcasting: `r(I_indices)` is n_I×1, `a_I` is n_I×n_a_I, `tau_a_I` is 1×n_a_I.

db_E_dt (n_E × n_b_E column vector, or empty) Time derivatives of excitatory STD variables:

```
db_E_dt = (1 - b_E) / tau_b_E_rec - (b_E .* r(E_indices)) / tau_b_E_rel
```

- Recovery term: `(1 - b_E) / tau_b_E_rec` drives b back toward 1
- Depression term: `(b_E .* r(E_indices)) / tau_b_E_rel` reduces b during firing

db_I_dt (n_I × n_b_I column vector, or empty) Time derivatives of inhibitory STD variables:

```
db_I_dt = (1 - b_I) / tau_b_I_rec - (b_I .* r(I_indices)) / tau_b_I_rel
```

Similar to `db_E_dt` but for inhibitory neurons. Empty if n_b_I = 0.

Performance Optimizations

Persistent Variables

The function uses persistent variables for efficient input interpolation: - `u_interpolant`: A `griddedInterpolant` object for fast interpolation of `u_ex` - `t_ex_last`: Stores the last time vector to detect changes and rebuild the interpolant only when necessary

This avoids repeatedly creating the interpolant object on every function call during ODE integration.

Notes

1. **Dale's Law**: The connection matrix `W` should respect Dale's law (excitatory neurons only make excitatory connections, inhibitory neurons only make inhibitory connections).
2. **Disabling Adaptation**: Set `n_a_E = 0` or `n_a_I = 0` to disable adaptation for excitatory or inhibitory neurons, respectively.
3. **Disabling STD**: Set `n_b_E = 0` or `n_b_I = 0` to disable short-term depression for excitatory or inhibitory neurons, respectively. When disabled, firing rates are computed as `r = activation_function(x_eff)` without the b multiplicative factor.
4. **State Vector Size**: The total size of the state vector is:

   ```
   N_sys_eqs = n_E * n_a_E + n_I * n_a_I + n_E * n_b_E + n_I * n_b_I + n
   ```

5. **STD Initial Conditions**: The b variables should be initialized to 1.0 (no depression). They will evolve according to the dynamics during simulation.

6. **Broadcasting**: The adaptation dynamics use MATLAB's implicit broadcasting to efficiently compute element-wise operations across multiple time constants.
7. **Interpolation**: External input is linearly interpolated between time points in `t_ex`. The interpolation method uses `'none'` for extrapolation, which returns NaN for out-of-bounds queries to catch errors if the ODE solver attempts to step outside the defined time range.
8. **Typical STD Time Constants**:
   - Recovery (`tau_b_E_rec`/`tau_b_I_rec`): 0.5-1.5 seconds (slow recovery from depression)
   - Release (`tau_b_E_rel`/`tau_b_I_rel`): 0.01-0.1 seconds (fast depression during activity)

---

compute_Jacobian_fast.m

Overview

`compute_Jacobian_fast.m` assembles the SRNN Jacobian with sparse matrices and Kronecker products. It produces the same matrix as `compute_Jacobian.m`, but the block-wise vectorization makes it far more efficient when Jacobians are needed repeatedly (e.g., Lyapunov spectrum via QR).

Key characteristics

- **Sparse blocks**: Each sub-block (`da/da`, `db/dx`, `dx/dx`, …) is built with `spdiags`, `kron`, or sparse triplets. The final Jacobian is sparse, which accelerates the `J * Psi` products integrated inside `lyapunov_spectrum_qr`.
- **Vectorized reuse**: Per-neuron structures are reused through Kronecker scaffolds (e.g., `kron(I_pop, diag(-1./tau_a_E))`), avoiding explicit loops over neurons or adaptation indices.
- **Drop-in compatibility**: Uses the same state ordering `[a_E; a_I; b_E; b_I; x]` and parameter fields. Existing code can switch to the fast version and, when needed, convert it to dense with `full(...)`.
- **STD assumption**: Matches the current SRNN dynamics by supporting at most one STD variable per neuron (`n_b_E, n_b_I` ∈ `{0,1}`).

Block construction highlights

- `da/da` blocks: `kron(I_pop, diag(-1./tau_a)) + kron(diag(-b.*c.*activation_function_derivative), (1./tau_a).*ones(1,n_a))` captures both the diagonal leak and the shared adaptation coupling per neuron.
- `da/db` & `da/dx`: Formed via sparse triplets so each adaptation row only touches its neuron's STD and dendritic states.
- `db/da`, `db/db`, `db/dx`: Use diagonal matrices for per-neuron coefficients combined with `kron` replicators over adaptation columns.
- `dx/da` & `dx/db`: Convert `W` to sparse and multiply by diagonal gain matrices, then replicate across adaptation/STD columns with `kron`. **All terms divided by `tau_d`** to match equation `dx/dt = (-x + W*r + u) / tau_d`.
- `dx/dx`: Implemented as `diag(-1/tau_d) + (W * diag(b .* activation_function_derivative)) / tau_d`, with both diagonal and coupling terms properly scaled by `tau_d`.

Usage

- `full_SRNN_caller.m` now evaluates both Jacobians at the initial state (printing absolute/relative differences) and uses `compute_Jacobian_fast` inside the Lyapunov wrapper.
- `compute_Jacobian_at_indices.m` calls the fast version and converts the sparse result to dense so downstream visualization scripts remain unchanged.
- Other scripts can opt-in by replacing calls to `compute_Jacobian` with the fast variant; for backward compatibility, keep both implementations available.

---

create_W_matrix.m

Overview

`create_W_matrix.m` generates a sparse connectivity matrix W with structured excitatory/inhibitory balance and row-mean centering. Located in `src/connectivity/`, it encapsulates the network connectivity generation logic.

Function Signature

```
function [W, M, G, Z] = create_W_matrix(params)
```

Inputs

- **params** (struct): Contains network connectivity parameters
  - `n`: Total number of neurons
  - `n_E`: Number of excitatory neurons
  - `n_I`: Number of inhibitory neurons
  - `mu_E`: Mean excitatory connection strength
  - `mu_I`: Mean inhibitory connection strength
  - `G_stdev`: Standard deviation of Gaussian perturbations
  - `indegree`: Expected in-degree (number of inputs per neuron)

Outputs

- **W** ($n \times n$ matrix): Final connectivity matrix (sparse, row-mean centered)
- **M** ($n \times n$ matrix): Mean connectivity structure
- **G** ($n \times n$ matrix): Gaussian random perturbations
- **Z** ($n \times n$ binary matrix): Sparsification mask (1 = connection removed)

Algorithm

1. Creates mean structure M: first n_E columns get mu_E, remaining n_I columns get mu_I
2. Adds Gaussian perturbations: W = M + G where G ~ N(0, G_stdev)
3. Applies sparsification: removes connections with probability (1 - indegree/n)
4. Row-mean centering: subtracts mean of non-zero elements in each row

Design Rationale

- **Row-mean centering**: Ensures balanced input to each neuron on average
- **Structured sparsity**: Maintains biologically realistic connectivity patterns
- **Separate outputs**: Returns M, G, Z for analysis while keeping params lightweight

---

initialize_state.m

Overview

`initialize_state.m` creates the initial state vector S0 for the SRNN with adaptation and short-term depression. Located in `src/`, it handles the complex state packing logic.

Function Signature

```
function S0 = initialize_state(params)
```

Inputs

- **params** (struct): Network parameters
  - `n`, `n_E`, `n_I`: Network size
  - `n_a_E`, `n_a_I`: Number of adaptation timescales
  - `n_b_E`, `n_b_I`: Number of STD timescales (0 or 1)

Output

- **S0** (N_sys_eqs × 1 vector): Initial state organized as `[a_E(:); a_I(:); b_E(:); b_I(:); x(:)]`

Initialization Strategy

- **a_E, a_I** (adaptation): Initialized to zero (no initial adaptation)
- **b_E, b_I** (STD): Initialized to one (no initial depression)
- **x** (dendritic): Small random values ~N(0, 0.01^2) to break symmetry

Usage

Eliminates repetitive initialization code and ensures consistent state packing across simulations.

---

generate_external_input.m

Overview

`generate_external_input.m` creates sparse random step function inputs for network stimulation. Located in `src/`, it provides flexible control over temporal and spatial input patterns.

Function Signature

```
function [u_ex, t_ex] = generate_external_input(params, T, fs, rng_seed, input_config)
```

Inputs

- **params** (struct): Contains `n` (number of neurons)
- **T** (scalar): Simulation duration (seconds)
- **fs** (scalar): Sampling frequency (Hz)
- **rng_seed** (scalar): Random seed for reproducibility
- **input_config** (struct):
  - `n_steps`: Number of temporal steps
  - `step_density`: Fraction of neurons receiving input per step (0-1)
  - `amp`: Amplitude scaling factor
  - `no_stim_pattern`: Logical array (1 × n_steps) specifying no-stim steps
  - `intrinsic_drive`: Constant background input (n × 1)

Outputs

- **u_ex** (n × nt matrix): External input, rows = neurons, columns = time
- **t_ex** (nt × 1 vector): Time vector

Algorithm

1. Generates random step amplitudes for each (neuron, step) pair
2. Applies spatial sparsity via step_density threshold
3. Zeros out steps specified in no_stim_pattern
4. Creates continuous-time signal by replicating steps
5. Adds constant intrinsic_drive

Design Features

- **Vectorized**: Precomputes all random values for efficiency
- **Flexible patterns**: Supports arbitrary stimulation sequences
- **Reproducible**: Uses dedicated RNG seed independent of network initialization

---

compute_lyapunov_exponents.m

Overview

`compute_lyapunov_exponents.m` is a unified wrapper for Lyapunov exponent computation supporting multiple methods. Located in `src/algorithms/Lyapunov/`, it simplifies the calling interface and includes helper functions.

Function Signature

```
function lya_results = compute_lyapunov_exponents(Lya_method, S_out, t_out, dt, fs, T_interval, params, opt
```

Inputs

- **Lya_method** (string): 'benettin', 'qr', or 'none'
- **S_out** (nt × N_sys_eqs): State trajectory
- **t_out** (nt × 1): Time vector
- **dt, fs**: Time step and sampling frequency
- **T_interval** ([T_start, T_end]): Analysis time window
- **params**: SRNN parameters
- **opts**: ODE solver options
- **ode_solver**: Function handle (e.g., @ode45)
- **rhs_func**: RHS function for integration
- **t_ex, u_ex**: External input data

Outputs

- **lya_results** (struct): Method-dependent results
  - For 'benettin': `LLE`, `local_lya`, `finite_lya`, `t_lya`
  - For 'qr': `LE_spectrum`, `local_LE_spectrum_t`, `finite_LE_spectrum_t`, `t_lya`, `sort_idx`, `params.N_sys_eqs`
  - For 'none': empty struct

Methods

Benettin's Algorithm

- Tracks single perturbation vector to compute largest Lyapunov exponent (LLE)
- Faster, suitable for chaos detection
- Returns time series of local and finite-time exponents

QR Decomposition Method

- Integrates tangent space using QR orthogonalization
- Computes full Lyapunov spectrum
- Automatically sorts by descending real part
- Computes Kaplan-Yorke dimension

Helper Functions

- **compute_kaplan_yorke_dimension**: Calculates fractal dimension from spectrum
- **SRNN_Jacobian_wrapper**: Provides Jacobian for QR method

---

unpack_and_compute_states.m

Overview

`unpack_and_compute_states.m` unpacks the state trajectory and computes firing rates with adaptation and STD effects. Located in `src/`, it consolidates state processing logic.

Function Signature

```
function [x, a, b, r] = unpack_and_compute_states(S_out, params)
```

Inputs

- **S_out** (nt × N_sys_eqs): State trajectory from ODE solver
- **params** (struct): Network parameters

Outputs

All outputs are structs with `.E` and `.I` fields: - **x**: Dendritic states (`.E` is n_E × nt, `.I` is n_I × nt) - **a**: Adaptation variables (`.E` is n_E × n_a_E × nt, `.I` is n_I × n_a_I × nt, may be empty) - **b**: STD variables (`.E` is n_E × nt, `.I` is n_I × nt, defaults to ones if disabled) - **r**: Firing rates (`.E` is n_E × nt, `.I` is n_I × nt)

Algorithm

1. **Unpack**: Extracts a_E, a_I, b_E, b_I, x from state vector
2. **Compute x_eff**: Applies adaptation: x_eff = x - c * sum(a)
3. **Compute r**: Applies STD and activation: r = b .* activation_function(x_eff)
4. **Split E/I**: Organizes variables into excitatory and inhibitory components

Design Benefits

- **Unified processing**: Combines unpacking and dependent variable computation
- **Consistent interface**: All outputs use `.E`/`.I` struct pattern
- **Handles edge cases**: Correctly manages empty adaptation/STD arrays

---

Plotting Functions

The `src/plotting/` directory contains five visualization functions with a consistent interface. All functions: - Accept time vector and data structs with `.E` and `.I` fields - Use custom colormaps: `inhibitory_colormap(8)` (reds/magentas) and `excitatory_colormap(8)` (blues/greens) - Plot inhibitory neurons first (background), then excitatory on top - Operate on current axes (for subplot compatibility) - Handle empty data gracefully

plot_external_input.m

```
function plot_external_input(t, u)
```

- **Inputs**: t (time), u (struct with .E and .I fields containing n_E/n_I × nt input)
- **Purpose**: Visualizes external stimulation patterns

plot_dendritic_state.m

```
function plot_dendritic_state(t, x)
```

- **Inputs**: t (time), x (struct with .E and .I fields containing n_E/n_I × nt dendritic states)
- **Purpose**: Shows dendritic potential dynamics

plot_adaptation.m

```
function plot_adaptation(t, a, params)
```

- **Inputs**: t (time), a (struct with .E and .I fields, n_E/n_I × n_a × nt), params
- **Purpose**: Displays adaptation variable time courses
- **Special handling**: Loops over neurons and adaptation timescales, shows 'No adaptation variables' if disabled

plot_firing_rate.m

```
function plot_firing_rate(t, r)
```

- **Inputs**: t (time), r (struct with .E and .I fields containing n_E/n_I × nt firing rates)
- **Purpose**: Shows neural activity patterns

plot_std_variable.m

```
function plot_std_variable(t, b, params)
```

- **Inputs**: t (time), b (struct with .E and .I fields containing n_E/n_I × nt STD variables), params
- **Purpose**: Visualizes synaptic depression dynamics
- **Special handling**: Checks if b is all ones (no actual depression), shows 'No STD variables' if disabled

Colormap Design

- **excitatory_colormap**: Blues, greens, cyans (cool colors, positive connotation)
- **inhibitory_colormap**: Reds, magentas, purples (warm colors, negative connotation)
- **8 discrete colors**: Provides good visual distinction for typical neuron counts in plots
- **Interpolation**: Automatically extends to more colors if needed

---

full_SRNN_caller.m

Overview

`full_SRNN_caller.m` is the main simulation script for the SRNN reservoir model with spike-frequency adaptation and short-term synaptic depression. It demonstrates a complete workflow including network setup, external input generation, ODE integration, Lyapunov exponent computation, and visualization. The script is highly modular, using dedicated functions from `src/` for each major component.

Script Workflow

1. Initialization and Parameter Setup

- Clears workspace and sets random seeds for reproducibility
- Defines network parameters: size (`n`), E/I fraction (`f`), connectivity (`indegree`), chaos level
- Configures adaptation timescales (`n_a_E`, `n_a_I`, `tau_a_E`, `tau_a_I`, `c_E`, `c_I`)
- Configures short-term depression timescales (`n_b_E`, `n_b_I`, `tau_b_E_rec/rel`, `tau_b_I_rec/rel`)
- Sets activation function (e.g., piecewise sigmoid, tanh)

2. Connectivity Matrix Creation Uses `create_W_matrix(params)` to generate: - **W**: Sparse connectivity matrix with E/I structure and row-mean centering - **M**: Mean connectivity structure (mu_E for E→all, mu_I for I→all) - **G**: Gaussian random perturbations (stdev = G_stdev) - **Z**: Binary sparsification mask

Computes spectral abscissa of unscaled W and applies gamma scaling to achieve desired chaos level: - **gamma = 1 / abscissa_0**: Scaling factor to reach edge of chaos (where spectral abscissa = 1) - **W_scaled = params.level_of_chaos * gamma * W**: Final scaled connectivity matrix - **tau_d = 0.025 s**: Fixed dendritic time constant (25 ms)

With the equation `dx/dt = (-x + W*r + u) / tau_d`, the edge of chaos occurs when the spectral abscissa of W equals 1. The gamma scaling normalizes the unscaled matrix to this condition, then `params.level_of_chaos` controls whether the system is subcritical (<1), at the edge (=1), or chaotic (>1).

3. Initial Conditions Uses `initialize_state(params)` to create initial state vector `S0`: - Adaptation variables (`a_E`, `a_I`): initialized to zero - STD variables (`b_E`, `b_I`): initialized to one (no depression) - Dendritic states (`x`): small random values (~0.1)

4. External Input Generation Uses `generate_external_input(params, T, fs, rng_seed, input_config)` to create sparse random step function: - **n__steps**: Number of temporal steps - **step__density**: Fraction of neurons receiving input per step - **amp**: Amplitude scaling - **no__stim__pattern**: Logical array specifying steps with no stimulation - **intrinsic__drive**: Constant background input

5. ODE Integration

- Uses `ode45` (or other solver) with adaptive step size
- Jacobian provided via `compute_Jacobian_fast` for improved performance
- Integrates `SRNN_reservoir(t, S, t_ex, u_ex, params)` over time interval

6. Lyapunov Exponent Computation Uses `compute_lyapunov_exponents(...)` supporting three methods: - **'benettin'**: Computes largest Lyapunov exponent (LLE) via perturbation tracking - **'qr'**: Computes full Lyapunov spectrum via QR decomposition - **'none'**: Skips Lyapunov analysis

Returns `lya_results` struct with exponents, local/finite-time estimates, and time vector.

7. State Unpacking and Analysis Uses `unpack_and_compute_states(S_out, params)` to: - Unpack state trajectory into individual variables - Split into excitatory and inhibitory components - Compute firing rates with adaptation and STD effects - Returns structs `x`, `a`, `b`, `r` each with `.E` and `.I` fields

8. Visualization Creates 6-panel figure using dedicated plotting functions: 1. **External Input**: `plot_external_input(t, u)` - shows E/I stimulation 2. **Dendritic States**: `plot_dendritic_state(t, x)` - shows x dynamics 3. **Adaptation**: `plot_adaptation(t, a, params)` - shows adaptation variables 4. **Firing Rates**: `plot_firing_rate(t, r)` - shows neural activity 5. **STD Variables**: `plot_std_variable(t, b, params)` - shows synaptic depression 6. **Lyapunov**: Plots local/filtered Lyapunov exponents or full spectrum

All plots use custom colormaps: blues/greens for excitatory, reds/magentas for inhibitory.

9. Jacobian Eigenvalue Analysis

- Computes Jacobian at multiple time points (around step changes)
- Extracts eigenvalues and plots on complex plane
- Visualizes stability and oscillatory modes

Key Design Principles

1. **Modularity**: Major components extracted into reusable functions in `src/`
2. **Lightweight params**: Only W stored in params (not M, G, Z) for integration efficiency
3. **Consistent data structures**: State variables returned as structs with `.E` and `.I` fields
4. **Visualization consistency**: All plots use E/I colormaps, I plotted first (background), then E

Dependencies

The script uses the following functions from `src/`: - `create_W_matrix.m`: Connectivity matrix generation - `initialize_state.m`: Initial condition setup - `generate_external_input.m`: Input stimulus creation - `compute_lyapunov_exponents.m`: Lyapunov analysis wrapper - `unpack_and_compute_states.m`: State unpacking and firing rate computation - `plot_external_input.m`, `plot_dendritic_state.m`, `plot_adaptation.m`, `plot_firing_rate.m`, `plot_std_variable.m`: Visualization functions - `SRNN_reservoir.m`: ODE right-hand side - `compute_Jacobian_fast.m`: Jacobian computation

---

SRNN__reservoir__caller.m

Overview

`SRNN_reservoir_caller.m` is a simpler example script demonstrating how to use `SRNN_reservoir.m` with MATLAB's `ode45` solver. It provides a simplified workflow compared to `full_SRNN_caller.m` (which includes Lyapunov exponent calculations and more advanced features), making it ideal for getting started with the SRNN reservoir model.

Script Structure

1. Initialization

- Clears workspace and sets random seed for reproducibility
- Uses `clear all` to reset persistent variables in `SRNN_reservoir.m`

2. Network Setup Uses helper functions from `reference_files/`: - **`generate_M_no_iso(n, w, sparsity, EI)`**: Creates a sparse, strongly-connected connectivity matrix - `n = 10`: Total number of neurons - `EI = 0.7`: 70% excitatory, 30% inhibitory - `sparsity`: Controlled by mean in/out degree (5 connections) - `w`: Structure defining weight scaling for EE, EI, IE, II connections - **`get_EI_indices(EI_vec)`**: Extracts indices of excitatory and inhibitory neurons

3. Time Parameters

- Sampling frequency: `fs = 1000` Hz (1 ms resolution)
- Time interval: `T = [-2, 5]` seconds (negative time allows for initial transients)
- Time vector: `t = linspace(T(1), T(2), nt)'` (column vector)

4. External Input Design The script creates a 2-component input: 1. **Stimulus**: Sine wave applied to neuron 1 - Baseline: 0.5, Amplitude: 0.5 - Frequency: 1 Hz, Duration: 2 seconds - Starts at t = 1 second 2. **DC Offset**: Constant background input (0.1) - Ramps up smoothly over 1.5 seconds to avoid initial transients - Prevents network from settling at zero activity

5. Adaptation Configuration

- **Excitatory neurons**: 3 adaptation time constants
  - `tau_a_E = logspace(log10(0.3), log10(15), 3)`
  - Spans from 0.3s (fast) to 15s (slow)
- **Inhibitory neurons**: No adaptation (`n_a_I = 0`)
- **Dendritic time constant**: `tau_d = 0.025` seconds (25 ms)

6. Integration with ode45

- Wraps `SRNN_reservoir` to include extra parameters: `t`, `u_ex`, `params`
- ODE options: `RelTol = 1e-6`, `AbsTol = 1e-8`
- Uses `ode45` (non-stiff solver, good for most networks)

7. Post-Processing Manually unpacks the state vector since `SRNN_reservoir` uses a simplified state organization: - State organization: `S = [a_E(:); a_I(:); x(:)]` - Extracts: - `a_E_ts`: n_E $\times$ n_a_E $\times$ nt (adaptation variables for E neurons) - `a_I_ts`: n_I $\times$ n_a_I $\times$ nt (adaptation variables for I neurons, empty if n_a_I = 0) - `x_ts`: n $\times$ nt (dendritic states) - Computes firing rates using `compute_dependent_variables` (passes empty arrays for `b_E_ts`, `b_I_ts` since synaptic depression is not included)

8. Visualization Creates a 3-panel figure: 1. **External Input**: Shows `u_ex` for neurons 1-2 2. **Firing Rates**: Plots `r(t)` for all neurons 3. **Adaptation Variables**: Shows all adaptation variables for the first E neuron with time constants in legend

Key Parameter Choices and Rationale

Parameter	Value	Rationale
`n = 10`	Small network	Fast simulation, easy visualization
`EI = 0.7`	70% excitatory	Biologically realistic ratio
`mean_in_out_degree = 5`	Moderate connectivity	Ensures strong connectivity without being fully connected
`scale = 0.5/0.79782`	Weight scaling	Provides stable dynamics (not too weak, not too strong)

Parameter	Value	Rationale
`tau_d = 0.025` s	Fast dendritic time constant	Typical for rate models, allows quick responses
`n_a_E = 3`	Multiple timescales	Captures rich adaptation dynamics
`DC = 0.1`	Small positive offset	Keeps network in sensitive regime

Modifying for Different Configurations

Change Network Size

```
n = 100;  % Larger network
mean_in_out_degree = 10;  % Scale connectivity accordingly
```

Disable Adaptation

```
n_a_E = 0;  % No adaptation
n_a_I = 0;
```

Use ReLU Activation Instead of Tanh

```
params.activation_function = @(x) max(0, x);
```

Change Stimulus Pattern

```
% Square wave instead of sine
u_ex(1, stim_start_idx:stim_end_idx) = stim_b0 + amp * sign(sin(2*pi*f_sin*t_stim));

% Multiple neurons stimulated
u_ex(1:3, stim_start_idx:stim_end_idx) = stim_b0 + amp * sin(2*pi*f_sin*t_stim);
```

Use Stiff Solver (for larger networks)

```
[t_out, S_out] = ode15s(SRNN_wrapper, t, S0, ode_options);
```

Expected Outputs

1. **Console Output**:
   - Integration progress message
   - Completion message with simulation time
   - Simulation speed (ratio of compute time to real time)
2. **Figure**:
   - Three-panel visualization showing input, firing rates, and adaptation
   - Clear legends and labels for interpretation
3. **Workspace Variables**:
   - `S_out`: Full state trajectory (nt × N_sys_eqs)
   - `t_out`: Output time vector from ODE solver
   - `r_ts`: Firing rates (n × nt)
   - `a_E_ts`, `a_I_ts`, `x_ts`: Unpacked state variables
   - `params`: Parameter structure (useful for further analysis)