# Stability Analysis Code Structure

This document describes the object-oriented architecture of the StabilityAnalysis codebase, focusing on three core classes that implement simulations of recurrent neural networks with spike-frequency adaptation (**SFA**) and short-term synaptic depression (**STD**). These three classes were used together to create Figure 2.

---

## System Equations

The SRNN dynamics are governed by the following system of differential equations (see system_equations.md):

$$\frac{dx_i}{dt} = \frac{-x_i + u_i + \sum_{j=1}^{N} w_{ij}\, b_j r_j}{\tau_d}$$

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

$$\frac{da_{ik}}{dt} = \frac{-a_{ik} + r_i}{\tau_{a_k}}$$

$$\frac{db_i}{dt} = \frac{1 - b_i}{\tau_{rec}} - \frac{b_i\, r_i}{\tau_{rel}}$$

| Symbol | Description |
|---|---|
| $x_i$ | Dendritic potential of neuron $i$ |
| $r_i$ | Firing rate (output of activation function $\phi$) |
| $a_{ik}$ | SFA variable for neuron $i$ at timescale $k$ |
| $b_i$ | STD variable (synaptic resources available) |
| $w_{ij}$ | Connection weight from neuron $j$ to neuron $i$ |
| $u_i$ | External input to neuron $i$ |
| $\tau_d$ | Dendritic time constant |
| $\tau_{a_k}$ | SFA time constant for timescale $k$ |
| $\tau_{rec}, \tau_{rel}$ | STD recovery and release time constants |
| $c$ | SFA coupling strength |

For a complete parameter reference, see parameter_table.md.

---

## Class: **SRNNModel**

**File:** SRNNModel.m

`SRNNModel` is the main simulation class. It encapsulates network parameters, ODE integration, Lyapunov exponent computation, and visualization.

### Usage Pattern

```
model = SRNNModel('n', 300, 'n_a_E', 3, 'n_b_E', 1, 'level_of_chaos', 1.0);
model.build();   % Initialize W, stimulus, initial conditions
model.run();     % Integrate ODEs (and compute Lyapunov exponents if lya_method != 'none')
model.plot();    % Generate time series plots
```

**Key Properties**

| Category | Properties | Description |
|---|---|---|
| **Network Architecture** | `n, f, indegree` | Total neurons, E fraction, expected in-degree |
| **RMT Parameters** | `mu_E_tilde, mu_I_tilde, sigma_E_tilde, sigma_I_tilde` | Tilde-notation stats (Harris 2023) |
| **SFA** | `n_a_E, n_a_I, tau_a_E, tau_a_I, c_E, c_I` | Number of SFA timescales, SFA time constant vectors (log spaced), coupling constants |
| **STD** | `n_b_E, n_b_I, tau_b_E_rec, tau_b_E_rel`, etc. | Enable/disable STD (0 or 1), and time constants if enabled |
| **Dynamics** | `tau_d` | Dendritic integration time constant |
| **Nonlinearity** | `activation_function, activation_function_derivative` | Firing rate nonlinearity and its derivative |
| **Simulation** | `fs, T_range, ode_solver, ode_opts` | Sampling freq, time interval, solver |
| **Lyapunov** | `lya_method, lya_T_interval` | Method ('benettin', 'qr', 'none'), lyapunov rescaling interval |

**Core Methods**

| Method | Description |
|---|---|
| `build()` | Computes derived params, creates W via `RMTMatrix`, generates stimulus, initializes state |
| `run()` | Integrates ODEs, computes Lyapunov exponents, decimates state for plotting |
| `plot()` | Generates 6-panel time series figure (input, x, r, a, b, Lyapunov) |
| `get_params()` | Returns a struct for compatibility with legacy functions |

**State Vector Organization**

The state vector `S` is packed as:

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

where: - `a_E`: E adaptation variables (n_E x n_a_E, flattened) - `a_I`: I adaptation variables (n_I x n_a_I, flattened) - `b_E`: E STD variables (n_E x n_b_E, flattened) - `b_I`: I STD variables (n_I x n_b_I, flattened) - `x`: Dendritic states for all neurons (n x 1)

**Equation Implementation in `dynamics_fast()`**

The static method `dynamics_fast(t, S, params)` implements the ODEs. Key steps:

1. **Interpolate external input**: `u = params.u_interpolant(t)'`
2. **Unpack state**: Extract `a_E, a_I, b_E, b_I, x` from S
3. **Compute effective potential**: `x_eff = x - c_E * sum(a_E, 2)` for E neurons (and similarly for I)
4. **Apply STD**: `b = ones(n,1)` with `b(E_indices) = b_E` if enabled
5. **Compute firing rate**: `r = activation_fn(x_eff)`
6. **Compute derivatives**:
   - `dx_dt = (-x + W * (b .* r) + u) / tau_d`
   - `da_E_dt = (r(E_indices) - a_E) ./ tau_a_E`
   - `db_E_dt = (1 - b_E) / tau_b_E_rec - (b_E .* r(E_indices)) / tau_b_E_rel`
7. **Pack derivatives**: `dS_dt = [da_E_dt(:); da_I_dt(:); db_E_dt(:); db_I_dt(:); dx_dt]`

**Lyapunov Methods**

The `lya_method` property controls how Lyapunov exponents are computed:

| Method | Output | Description | Time Complexity | Memory |
|---|---|---|---|---|
| `'benettin'` (default) | LLE only | Computes local and finite-time largest Lyapunov exponent using a shadow trace (single perturbation vector rescaled periodically) | $O(T \cdot K\_steps \cdot N\_states)$ | $O(N\_states)$ |
| `'qr'` | Full spectrum | Computes all N_states Lyapunov exponents via QR decomposition of the tangent space; integrates N_states² variational equations | $O(T \cdot K\_steps \cdot N\_states^3)$ | $O(N\_states^2 + T \cdot N\_states)$ |
| `'none'` | — | Skips Lyapunov computation entirely | — | — |

where: - **N_states** = system dimension (`N_sys_eqs`), not to be confused with network size N - **T** = number of rescaling intervals = (`T_end` - `T_start`) / `lya_dt_interval` - **K_steps** = average ODE substeps per rescaling interval (depends on solver tolerances and dynamics stiffness)

> **Recommendation:** Use `'benettin'` for large networks (N_states > 100) or parameter sweeps. Use `'qr'` only when the full spectrum is needed and N_states is small (the N_states³ scaling makes it prohibitive for N_states > ~200).

---

## Class: RMTMatrix

**File:** RMTMatrix.m

`RMTMatrix` constructs sparse connectivity matrices following Random Matrix Theory (Harris et al., 2023). It enforces Dale's law by separating excitatory and inhibitory populations with distinct mean weights.

**Usage Pattern**

```
rmt = RMTMatrix(N);
rmt.alpha = indegree / N;      % Sparsity
rmt.f = 0.5;                    % E fraction
rmt.mu_tilde_e = 3 * D;        % E mean (tilde notation)
rmt.mu_tilde_i = -4 * D;       % I mean (tilde notation)
rmt.sigma_tilde_e = D;         % E std dev
rmt.sigma_tilde_i = D;         % I std dev
W = rmt.W;                      % Access triggers construction
```

**Key Properties**

| Property | Description |
|---|---|
| `N` | Network size |
| `alpha` | Connection probability (sparsity) |
| `f` | Fraction of excitatory neurons |
| `mu_tilde_e`, `mu_tilde_i` | Normalized population means |
| `sigma_tilde_e`, `sigma_tilde_i` | Normalized population std devs |
| `zrs_mode` | Zero row sum mode: `'none'`, `'ZRS'`, `'SZRS'`, `'Partial_SZRS'` |

**Dependent Properties (Computed)**

| Property | Formula | Description |
|---|---|---|
| mu_se | $\alpha \cdot \tilde{\mu}_E$ | Sparse excitatory mean |
| mu_si | $\alpha \cdot \tilde{\mu}_I$ | Sparse inhibitory mean |
| R | $\sqrt{N(f\sigma_{se}^2 + (1-f)\sigma_{si}^2)}$ | Theoretical spectral radius (Eq 18) |
| lambda_0 | $N(f\mu_{se} + (1-f)\mu_{si})$ | Outlier eigenvalue (Eq 17) |

**W Matrix Construction**

The weight matrix W is constructed as:

```
W_dense = (A * D) + M;    % A: Gaussian random, D: variance diagonal, M: low-rank mean
W = S .* W_dense;         % S: sparse binary mask
```

where: - `A` is an N x N Gaussian random matrix (mean 0, variance 1) - `D = diag([sigma_tilde_e, ...,` `sigma_tilde_e, sigma_tilde_i, ..., sigma_tilde_i])` encodes population variance - `M = u * v'` is the rank-1 mean structure with `v = [mu_tilde_e, ..., mu_tilde_e, mu_tilde_i, ..., mu_tilde_i]'` - `S` is a Bernoulli (0 or 1) sparsity mask with connection probability `alpha`

---

# Class: ParamSpaceAnalysis

**File:** ParamSpaceAnalysis.m

`ParamSpaceAnalysis` performs multi-dimensional gridded parameter space analysis of stabiltity (and other metrics). The user can choose any combination of model parameters to vary. By default it simulates all parameter combinations across four conditions: no adaptation, SFA only, STD only, and SFA + STD. This enables a systematic comparison of network behavior under different parameter ranges and conditions. The parallel computing toolbox is highly recommended but not required.

**Usage Pattern**

```
psa = ParamSpaceAnalysis('n_levels', 5, 'note', 'f_sweep');
psa.add_grid_parameter('f', [0.4, 0.6]);      % Sweep E fraction (range mode)
psa.add_grid_parameter('reps', [1:10]);        % 10 repetitions per point (explicit mode), otherwise defau
psa.model_defaults.n = 300;                     % Set constant parameters
psa.run();                                       % Execute analysis
psa.plot('metric', 'LLE');                       % Visualize results
```

**add_grid_parameter Behavior**

The `add_grid_parameter(param_name, values)` method supports two modes:

| Mode | Input | Behavior |
|---|---|---|
| **Range** | 1x2 vector [min, max] | Evenly divides the range into n_levels values using linspace(min, max, n_levels) |
| **Explicit** | Vector with 3+ elements | Uses the exact values provided (ignores n_levels for this parameter) |

**Examples:**

```
psa.add_grid_parameter('f', [0.4, 0.6]);        % Range mode: 5 levels → [0.4, 0.45, 0.5, 0.55, 0.6]
psa.add_grid_parameter('reps', [1, 2, 3, 4, 5]); % Explicit mode: uses [1, 2, 3, 4, 5] directly
```

**Note:** Each call to `add_grid_parameter` adds a dimension to the parameter space. The total number of simulations grows **multiplicatively**:

```
Total = n_conditions x n_reps x (n_levels_1 x n_levels_2 x ...)
```

For example, 3 grid parameters with 5 levels each, 10 repetitions, and 4 conditions yields: `4 x 10 x 5^3 = 5,000` total simulations.

**Key Features**

- **Multi-dimensional grid**: All combinations of specified parameters are tested
- **Four adaptation conditions** (default): | Condition | n_a_E | n_b_E | Description | |————|——-|——-|————-| | no_adaptation | 0 | 0 | Baseline | | `sfa_only` | 3 | 0 | SFA enabled | | `std_only` | 0 | 1 | STD enabled | | `sfa_and_std` | 3 | 1 | Both enabled |
- **Same W across conditions**: Each grid point uses identical connectivity for fair comparison
- **Batched parallel execution**: Uses `parfor` with checkpointing for resume capability
- **Randomized order**: Allows representative early stopping

**Output Structure**

Results are saved to `data/param_space/<timestamped_folder>/`:

```
param_space_<note>_nLevs_<N>_<timestamp>/
|-- param_space_summary.mat      # Grid configuration
|-- psa_object.mat               # Serialized PSA object
|-- no_adaptation/
|   +-- param_space_results_no_adaptation.mat
|-- sfa_only/
|   +-- param_space_results_sfa_only.mat
|-- std_only/
|   +-- param_space_results_std_only.mat
+-- sfa_and_std/
    +-- param_space_results_sfa_and_std.mat
```

Each result struct contains: `LLE`, `mean_rate`, `mean_synaptic_output`, `config`, `success`, and optionally `local_lya` time series.

---

## Supporting Functions

The `src/` directory contains additional utilities:

| Category | Files | Description |
|---|---|---|
| **Lyapunov** | compute_lyapunov_exponent.m, lyapunov_spectrum_qr.m, lyapunov_benettin.m | LLE and full spectrum computation |
| **Jacobian** | compute_Jacobian_fast.m, compute_Jacobian_at_indices.m | Sparse Jacobian construction |
| **Stimulus** | generate_external_input.m | Random step-function input generation |
| **Activation** | piecewiseSigmoid.m, piecewiseSigmoidDerivative.m | Nonlinearity and its derivative |
| **Plotting** | plot_SRNN_tseries.m, plot_SRNN_combined_tseries.m, etc. | Visualization utilities |
| **Analysis** | load_and_make_unit_histogram.m, load_and_plot_lle_by_stim_period.m | Post-hoc analysis of PSA results |

---

## See Also

- parameter_table.md – Complete parameter reference
- system_equations.md – Mathematical model
- Script_Notes.md – How to run the Figure 2 scripts