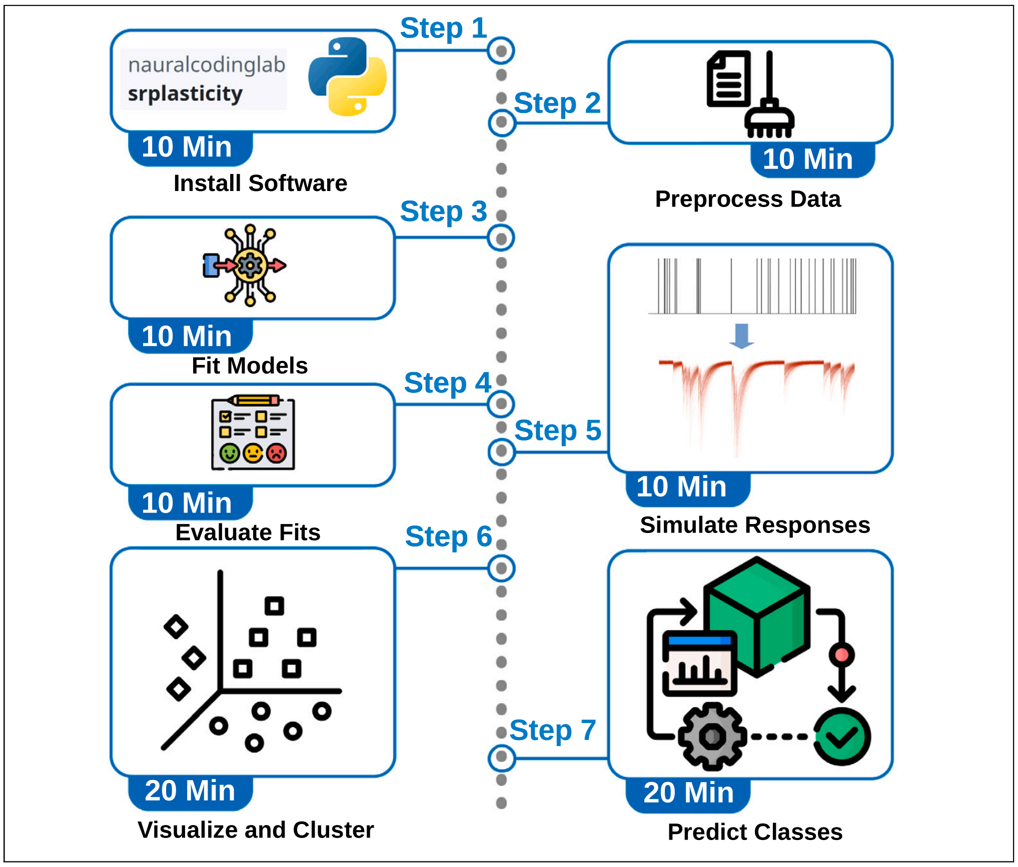
Protocol

Computational protocol for modeling and analyzing synaptic dynamics using SRPlasticity



Jade Poirier, John Beninger, Richard Naud

jgbeninger@gmail.com (J.B.)
rnaud@uottawa.ca (R.N.)

Highlights

Steps for flexibly capturing synaptic dynamics using SRPlasticity software

Instructions for automated fitting to infer model parameters from experimental data

Guidance on predicting synaptic responses to novel presynaptic spike trains *in silico*

Procedures for visualization, clustering, and predicting classes

Transient changes in synaptic strength, known as short-term plasticity (STP), play a fundamental role in neuronal communication. Here, we present a protocol for using SRPlasticity, a software package that implements a computational model of STP. SRPlasticity supports automatic characterization of electrophysiological data and simulation of synaptic responses. We describe steps for installing and utilizing SRPlasticity, preprocessing data, fitting models, and simulating responses. We then detail procedures for analyzing spike response plasticity (SRP) model parameters to infer functional groupings of STP.

Poirier et al., STAR Protocols 6, 103652
March 21, 2025 © 2025
Published by Elsevier Inc.
https://doi.org/10.1016/j.xpro.2025.103652

Protocol

Computational protocol for modeling and analyzing synaptic dynamics using SRPlasticity

Jade Poirier,[1,3] John Beninger,[1,3,4,*] and Richard Naud[1,2,5,*]

[1]Center for Neural Dynamics and Artificial Intelligence, uOttawa Brain and Mind Research Institute, Department of Cellular and Molecular Medicine, University of Ottawa, Ottawa, ON K1H 8M5, Canada

[2]Department of Physics, University of Ottawa, Ottawa, ON K1H 8M5, Canada

[3]These authors contributed equally

[4]Technical contact

[5]Lead contact

*Correspondence: jgbeninger@gmail.com (J.B.), rnaud@uottawa.ca (R.N.)

https://doi.org/10.1016/j.xpro.2025.103652

SUMMARY

Transient changes in synaptic strength, known as short-term plasticity (STP), play a fundamental role in neuronal communication. Here, we present a protocol for using SRPlasticity, a software package that implements a computational model of STP. SRPlasticity supports automatic characterization of electrophysiological data and simulation of synaptic responses. We describe steps for installing and utilizing SRPlasticity, preprocessing data, fitting models, and simulating responses. We then detail procedures for analyzing spike response plasticity (SRP) model parameters to infer functional groupings of STP.

For complete details on the use and execution of this protocol, please refer to Rossbroich et al.[1] and Beninger et al.[2]

BEFORE YOU BEGIN

Neuronal communication is often intricately linked to transient changes in synaptic efficacy, known as Short-Term Plasticity (STP).[3–5] It is therefore crucial to understand how STP shapes information flow in neuronal networks to generate a complete description of network function. However, synapses exhibit considerable heterogeneity in their STP properties.[6,7] Moreover, this heterogeneity is believed to underlie classes of connections[2,8,9] that may have distinct information processing roles.[10–14] Models of STP can help to parse this complexity by allowing for the identification of functionally similar synaptic dynamics, and by predicting how synapses will respond to novel inputs. Previous computational models have been successful in capturing STP dynamics[15] but often rely on specific assumptions about synaptic dynamics. These assumptions can render them too complex to generalize functionally similar trends between synapses into comparable representations.[1] Simultaneously, the specific assumptions behind many models render them too restrictive to capture the large diversity of dynamics observed experimentally.[1] Here, we describe how to use a mathematically flexible linear-nonlinear model known as the Spike Response Plasticity (SRP) model that efficiently characterizes synaptic dynamics and predicts their behavior under novel conditions.

The SRPlasticity package characterizes synaptic dynamics by estimating SRP model parameters[1] from limited amounts of experimental data. These parameters can be used to predict synaptic responses to novel stimuli and quantify the degree of difference between different synapses.[2] Models created in SRPlasticty can capture a large range of synaptic behaviors, including short-term facilitation, short-term depression, biphasic plasticity, sub- and supra-linear facilitation, and post-burst potentiation.[1] While the SRP model can be configured in various ways, we describe how to use a simple version with "amplitude" parameters that capture the direction (increased or decreased efficacy) and magnitude of synaptic effects and a "baseline" parameter that helps determine how these effects accumulate (i.e., linearly, supra-linearly, sub-linearly). We also present several ways that the model can be used to predict variability in synaptic efficacy. This protocol directly enables users to model existing data, generalize inferences from their modeling, and run simulations that include synaptic dynamics. Users with more advanced needs can use this protocol as an introduction to the SRP model and the SRPlasticity package.

We have provided users with example data, which can be found inside the "data/star_protocol_example" folder in the GitHub SRPlasticity repository. Although the example data used in this protocol described to a single synapse, users can easily apply the same steps to multiple synapses by iterating over each one. We strongly encourage readers to use our Jupyter notebook, which can be accessed from the "Notebooks" folder of the repository for the most efficient experience learning to use SRPlasticity.

KEY RESOURCES TABLE

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Software and algorithms		
Python >= 3.8	Python Software Foundation	https://www.python.org
SRPlasticity	Rossbroich et al.[1]	https://github.com/nauralcodinglab/srplasticity.git
NumPy >= 1.24.4	van der Walt et al.[16]	https://github.com/numpy/numpy
SciPy >= 1.5.2	Virtanen et al.[17]	https://github.com/scipy/scipy
Matplotlib	Hunter[18]	https://github.com/matplotlib/matplotlib
Pandas	Mckinney[19] The Pandas Team[20]	https://github.com/pandas-dev/pandas
Scikit-learn	Pedregosa et al.[21]	https://github.com/scikit-learn/scikit-learn
Anaconda (recommended)	Anaconda, Inc.	https://www.anaconda.com/products/individual

MATERIALS AND EQUIPMENT

- Pre-processed data organized in a matrix as previously described (see preprocess synaptic responses).

  Note: The SRPlasticity package is designed to estimate model parameters from the amplitudes of postsynaptic current (PSC) or postsynaptic potential (PSP) traces elicited by the successive stimulation of a specific pre-synaptic neuron. For detailed electrophysiological protocols, see Seeman, Campagnola et al.,[22] Campagnola, Seeman et al.,[6] or Chamberland et al.[23]

  ⚠ CRITICAL: Formatting the recorded responses adequately is essential for the software to work as intended. Organize a Numpy array with rows corresponding to different recorded traces and columns corresponding to the amplitude of the change in post-synaptic current or voltage elicited by sequential presynaptic action potentials. Save each stimulation protocol as an inter-spike interval (ISI) vector (refer to step-by-step method details). Since the quality of each synapse model depends on the data on which it is fitted, obtain as many sweeps as possible from as many different protocols as possible for each synapse.

- Python environment and required packages.
  - Python >= 3.8.
  - NumPy >=1.240.40.
  - Scipy >=1.5.2.
  - Matplotlib.
  - Pandas.
  - Scikit-learn.

⚠ CRITICAL: To create and execute code using the SRPlasticity package, it is necessary to have a functional and current Python installation, along with an integrated development environment (IDE) or editor. There are various methods of obtaining a working Python installation. The section "installing SRPlasticity" explains how to use Anaconda for this purpose. We highly recommend that new users follow these instructions, while experienced users can set up their workspace as they see fit.

- Tools for plotting figures in Rossbroich et al.[1]
  - Seaborn.[24]
  - Spiffyplots[25] >= 0.5.
- Hardware.
  - A standard computer capable of supporting the software requirements described above.

STEP-BY-STEP METHOD DETAILS

Install SRPlasticity

Timing: 10 min

Install the SRPlasticity package within a virtual environment.

Note: The steps in this section and the instructions in the text version of this protocol explain how to install and use the SRPlasticity package locally. Our code repository (see key resources table) also contains a link to a cloud hosted Jupyter notebook which implements all the sections after this one and has been tested to run without requiring users to perform any local configurations. Many users may find that copying and modifying this notebook is sufficient for their needs. Users selecting this option can jump directly to step 9.

Note: We suggest that new users install Anaconda as it offers an up-to-date version of Python and a popular integrated development environment (IDE) called Spyder. We also recommend that new and advanced users set up a separate virtual environment before installing the SRPlasticity package. This setup enables users to allocate specific dependencies to the project, thereby avoiding conflicts between different projects, as these dependencies remain isolated from the system-wide packages. All steps will be shown, allowing the user to select the most suitable starting point for them.

1. Download and install the Anaconda Python distribution compatible with your operating system. Detailed instructions on how to do so can be found at https://docs.anaconda.com/anaconda/install/.
2. Once Anaconda is installed, open an Anaconda Prompt by following the instructions at https://docs.anaconda.com/anaconda/user-guide/getting-started/.
3. To create a virtual environment, enter the following commands into the Anaconda Prompt window:

Note: We advise readers to exercise caution when copying code directly from text boxes in the manuscript.pdf as copying from boxes that are split over two pages can result in copying unwanted elements from the footer. Instead, we direct readers to the Jupyter notebook in our repository where the code is presented continuously.

```
conda create -n srplasticity python=3.8

conda activate srplasticity
```

4. Enter the following lines to download the SRPlasticity repository:

```
git clone https://github.com/nauralcodinglab/srplasticity

cd srplasticity
```

5. Install the packages listed in the "key resources table" and "materials and equipment setup".
   a. First, ensure that pip is installed in your Conda environment:

```
conda install pip
```

   b. Install the packages:

```
pip install -e
```

6. Verify the installation process was successful. To do so, follow these steps:
   a. Execute the following code:

```
python3

import srplasticity
```

   b. If the import runs without any errors, the installation is successful. However, if there are any issues, refer to the CRITICAL section below.

Note: Windows users should enter `python` instead of `python3`.

7. Launch an IDE.

Note: We recommend the Spyder IDE to new users, which is in the "Home" tab of Anaconda Navigator. Experienced users can opt to use their preferred programming environments.

8. Within the IDE, change the Python interpreter to the one corresponding to the new Anaconda environment.

Note: The option will most likely be found in the "Settings" tab in most IDEs. For Spyder users, click on "Tools," "Preferences," then "Python interpreter." This will allow you to select the path to the "python.exe" file within the "srplasticity" folder. Select this file as your new Python interpreter.

Optional: Step 6 can be repeated here by running a script containing the line `import srplasticity`. The absence of errors would indicate a proper change of Python interpreter.

Note: The SRPlasticity package can be installed directly using the command `pip install srplasticity`. However, the most up-to-date functionality will be available by installing the GitHub version and periodically pulling the latest version. Users are free to use whichever method best suits their workflow.

⚠ CRITICAL: To ensure that SRPlasticity package imports and works correctly, it is crucial to install it using the appropriate Python installation. If the installation verification encounters any issues, refer to the "troubleshooting" section for assistance.

Preprocess synaptic responses

Timing: 10 min

Responses are normalized to the average of all first responses.

Note: We assume postsynaptic response amplitudes have already been computed and the data do not require extensive cleaning. Depending on the dataset, additional preprocessing steps (e.g., removing outliers) may be needed. We direct the user to Beninger et al.[2] for more extensive pre-processing methods.

9. Open an editor.

Note: Users who followed the SRPlasticity installation instructions, can do so by opening Anaconda Navigator, going to the "Home" tab, locating "Spyder," and then launching the Spyder integrated development environment (IDE). Visit the following link for further instructions on how to use Spyder: https://docs.spyder-ide.org/current/quickstart.html. Experienced users can open the editor of their choice.

10. Import the following packages:

```
import pickle

from pathlib import Path

from srplasticity.srp_extrafuncs import norm_responses
```

11. To import the preprocessed data and assign it to a variable, run the following code.

Note: The user's unnormalized target dictionary file should replace the one used here as an example, as shown by this example using "chamberland2018_data.p"[23] from the SRPlasticity directory.

```
#example path construction

main_folder = Path("~").expanduser() #if srplasticity is in home directory

file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/chamberland2018_
data.p")

with open(file_path, "rb") as file:

            target_dict = pickle.load(file)
```

CRITICAL: "target_dict" should be structured as specified in the "data collection" section. When using a single synapse, running "print(target_dict)" should produce an output similar to this: `{'10020': array([[1.02, 0.48, ..., 7.65, 5.26], ..., [1.38, 0.84, ..., 4.14, 6.03]]), ''100'': ...}`. Note: in the sample output we round entries to two floating points and provide only a portion of full printed output. The keys correspond to the names of the stimulation protocols, and their values are NumPy arrays containing the recorded responses. Each row in a NumPy array represents a different run, and each column corresponds to a stimulation number.

Note: If the user is working with multiple synapses, they should include first-level keys in "multi_target_dict" corresponding to the synapses" identification. These keys should each encompass a dictionary of protocols, such as the one given in this example. For example: `{''synapse_1'': {'10020': array([[1.02, 0.48, …, 7.65, 5.26], …, [1.38, 0.84, …, 4.14, 6.03]]), ''100'': …}, ''synapse_2'': {'10020':…}, …}.`

Note: This example is specific to pickle files. Therefore, the user may need to install other packages, depending on the file type they are trying to import. Analogous code can be used to import data from a CSV file. This requires first importing the pandas package instead of the pickle one and using "`pd.read_csv(''filename.csv'')`" instead of "`pickle.load(open(''filename.p'', ''rb''))`", where "filename" corresponds to the name of the user's file.

12. Use the following function to normalize all responses at a certain synapse to the average of all its first responses:

```
normed_all = norm_responses(target_dict)
```

Note: This example demonstrates how to normalize responses for a single synapse. To handle multiple synapses, users should iterate over each synapse's identification in "multi_target_dict" (refer to the first note from the previous step for the structure of "multi_target_dict" when dealing with multiple synapses) and apply "norm_responses" to each of their values:

```
normed_all = {pair_id: norm_responses(pair_target_dict) for pair_id, pair_target_dict in
multi_target_dict.items()}
```

13. Save the normalized responses as follows:

```
with open("normalized_responses.p'', ''wb'') as file:

pickle.dump(normed_all, file)
```

⚠ CRITICAL: At this point, the data should be preprocessed and structured as specified in the "data collection" section. The dictionary "normed_all" should have the same structure as "target_dict," (refer to step 11) the only difference being in the values of the responses themselves.

Fit the EasySRP model to synaptic responses

⏲ Timing: 10 min

Model parameters are estimated based on a synapse's post-synaptic responses to stimulation, thus characterizing the synaptic dynamics of that synapse.

Note: We recommend that users try this version of the model before moving on to more advanced versions, described below.

14. Open an editor.

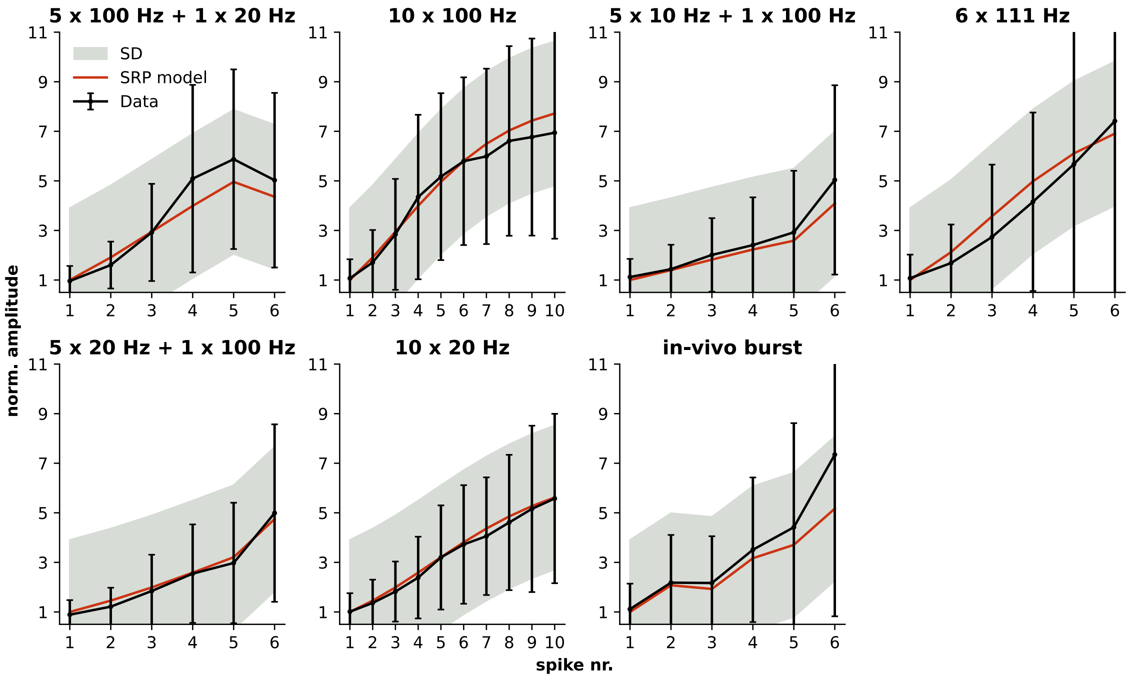


Figure 1. Example plot showing EasySRP model predictions against experimental data

The red lines and shaded areas show the model's predicted mean responses and their standard deviation, respectively. The black lines and their error bars show the mean of the normalized recorded responses and their standard deviation. Data from Chamberland et al.[23]

Note: If the user fully followed the SRPlasticity installation instructions, open Anaconda Navigator, go to the "Home" tab, locate "Spyder," and then launch the IDE. Visit the following link for further instructions on how to use Spyder: https://docs.spyder-ide.org/current/quickstart.html. Experienced users can open the IDE of their choice.

15. Once in an editor, import the necessary packages by adding the following lines of code:

```
import pickle

import numpy as np

import matplotlib.pyplot as plt

from pathlib import Path

from srplasticity.srp import easySRP

from srplasticity.srp_extrafuncs import (

    _EasySRP_dict_to_tuple,

    easy_fit_srp,

    mse_loss,

    plot_fit_easySRP,

    plot_mse_fig,

    plot_kernel_easySRP,

    plot_srp_easySRP)
```

16. To import the preprocessed data and assign it to a variable, run the following code.

    Note: The user's normalized target dictionary file should replace the one used here as an example here.

```
#example path construction

main_folder = Path("~").expanduser() #if srplasticity is in home directory

file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/chamberland2018_
data.p")

with open(file_path, "rb") as file:

              target_dict = pickle.load(file)
```

17. Assign to a variable (e.g., stimulus_dict) a dictionary in which each key represents the identification of a stimulation protocol, and each value is a list of its corresponding inter-spike intervals (ISIs). Use the following as an example:

```
stimulus_dict = {

         "20": [0] + [50] * 9,

         "100": [0] + [10] * 9,

         "20100": [0, 50, 50, 50, 50, 10],

         "10020": [0, 10, 10, 10, 10, 50],

         "10100": [0, 100, 100, 100, 100, 10],

         "111": [0] + [5] * 5,

         "invivo": [0, 6, 90.9, 12.5, 25.6, 9],

}
```

Optional: Define a dictionary called "protocol_names" in which the abbreviated names of the protocols are associated with their descriptive version. It is meant to be used later in plotting functions to set more descriptive titles. It is therefore optional, its absence only resulting in these titles being set to the abbreviated names of the protocols from "stimulus_dict".

```
#Optional dictionary of abbreviated protocol names and their descriptions

protocol_names = {

         "100": "10 x 100 Hz",

         "20": "10 x 20 Hz",

         "111": "6 x 111 Hz",

         "20100": "5 x 20 Hz + 1 x 100 Hz",

         "10100": "5 x 10 Hz + 1 x 100 Hz",

         "10020": "5 x 100 Hz + 1 x 20 Hz",

         "invivo": "in-vivo burst",

}
```

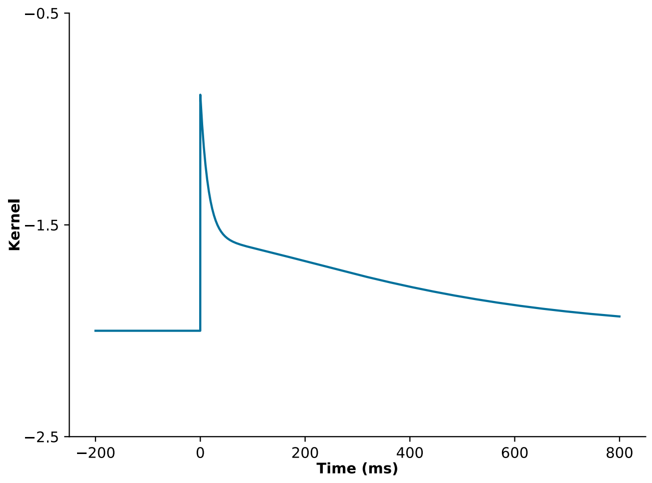


Figure 2. Example visualization of fitted EasySRP kernel which can be used to help understand the inferred dynamics of the synapse

18. Define a variable a list of the time constants corresponding to the efficacy kernel:

```
mu_taus = [15, 200, 300]
```

Note: This efficacy kernel results from the sum of three exponentials, each defined by its respective time constant. These constants are hyperparameters, meaning they remain fixed throughout the fitting process. When predicting responses, the SRP model will convolve this efficacy kernel with a sum of Dirac delta functions, corresponding to the spike train defined by the inter-spike intervals used. Intuitively, this means that longer time constants are more likely to influence predicted response amplitudes temporally distant from a given spike (compared to shorter ones), while shorter time constants predominantly affect predicted amplitudes temporally close to the spike. For more details, refer to Rossbroich et al.[1]

Note: The time constants `[15, 200, 300]` are the ones used by default by the "`easy_fit_srp`" function in step 19.20 Therefore, unless the user wishes to use different time constants, they can ignore this step and directly move on to the next.

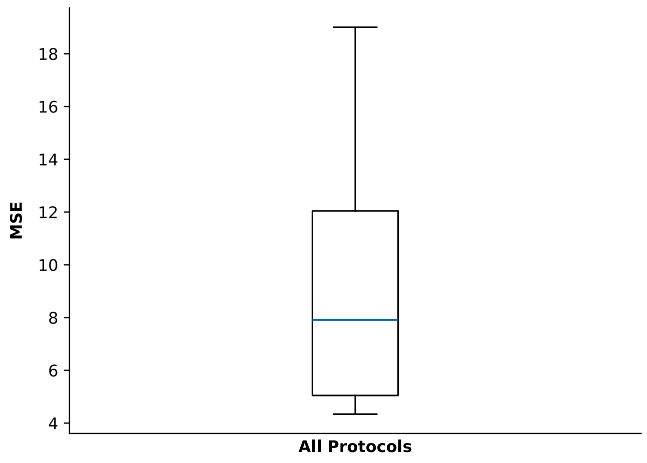


Figure 3. Example boxplot showing distribution of EasySRP model fitting error

Note: The model can use any number of time constants ("mu_taus"). We have found that using three time constants yields the best fitting performance in several contexts. Increasing the number of time constants allows the model to capture more complex dynamics, which may improve fitting performance. However, it also increases the risk of overfitting the data by encoding changes that result from experimental noise. Additionally, using more time constants may slightly increase the run time of the fitting algorithm. We recommend starting with the version presented above and adjusting the parameters as needed. That said, time constants partially overlap: the effect of exponentials with shorter time constants will decay more rapidly than the one of exponentials with longer time constants. This results in periods during which all exponentials contribute with various degrees to the effect, followed by periods during which only those with longer time constants are influential. Therefore, these time constants do not map precisely to the actual time constants in the dynamics. For instance, the dynamics may evolve according to a time constant of 100 ms which can be well captured by the SRP model blending time constants of 75 and 125 ms.

19. To infer model parameters, use this function, which iterates over the baseline parameter space:

```
params, best_loss = easy_fit_srp(stimulus_dict, target_dict, mu_taus)
```

⚠ CRITICAL: Running "print(params)" should output a dictionary similar to the following, containing these five parameters:

```
{'mu_baseline': -2.0, 'mu_amps': array([ 10.24, -146.55, 349.49]), 'mu_taus': array([ 15,
200, 300]), 'SD': 2.91, 'mu_scale': None}
```

Note: The "easy_fit_srp" function uses the fixed time constants defined by "mu_taus" to build the efficacy kernel used to predict a synapse's response to a spike train. The dictionary "stimulus_dict" contains these spike trains in the form of inter-spike intervals corresponding to stimulation protocols, whereas "target_dict" contains the synaptic responses to these stimulations. Therefore, "easy_fit_srp" predicts how the synapse would respond to the stimulation protocols in "stimulus_dict," compares the predicted responses to the recorded ones from "target_dict," adjusts its parameters, and tries again iteratively, minimizing the error between the predicted and recorded responses while doing so. The dictionary "params" corresponds therefore to the set of parameters minimizing this error, while "best_loss" is the minimal mean squared error computed.

Note: By default, the "`easy_fit_srp`" function uses parameter bounds best calibrated with the time constants [15, 200, 300]. However, users can set their own parameter bounds and input them as a keyword argument ("bounds") in the "easy_fit_srp" function. Please refer to the "`_default_parameter_bounds`" function within the "srp_extrafuncs" module for proper formatting of the bounds.

Note: The "fit_srp_model" function from the "srp_extrafuncs" module can be used by itself (i.e., without iterating over the baseline parameter space). Likewise, iterating over the space of other parameters at the same time could increase fitting quality, but will also increase computation time. We leave it to the user's discretion to adapt the protocol as best suits their work.

20. Convert the parameters to a tuple by first converting the dictionary to this type and then save them:

```
params_to_save = _EasySRP_dict_to_tuple(params)
with open("srp_params.p", "wb") as file:
    pickle.dump(params_to_save, file)
```

Optional: Plot and save the predicted and recorded PSP or PSC amplitudes by using the following code:

```
model = easySRP(**params)
fig, ([ax1, ax2, ax3, ax4], [ax5, ax6, ax7, ax8]) = plt.subplots(2, 4, figsize=(10, 6))
axes = (ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8)
plot_srp_easySRP(axes, params, target_dict, stimulus_dict, protocols=protocol_names)
fig.savefig("plot_srp_easySRP.svg", dpi=1200, transparent=True, bbox_inches="tight")
```

Note: The variable "`protocol_names`" is a dictionary in which the abbreviated names of the protocols are associated with their descriptive version. See step 17 for more details.

Optional: Plot the predicted and recorded PSP or PSC amplitudes corresponding to a certain protocol, a boxplot of the mean squared errors (per protocol) of the fit, and the estimated efficacy kernel by using these lines. The resulting plots should be similar to Figure 1 (predicted plot), Figure 2 (kernel plot), and Figure 3 (mean squared errors (MSE) boxplot):

```
# Compute MSEs
mean_dict = {protocol : model.run_ISIvec(isivec)[0] for protocol,
            isivec in stimulus_dict.items()}
mses = []
for protocol, responses in target_dict.items():
    mse = mse_loss({protocol:responses}, mean_dict)
    mses.append(mse)
# Plot fit
chosen_prot = "20"
fig1, axis1 = plt.subplots()
plot_fit_easySRP(axis1, model, target_dict, stimulus_dict, chosen_prot, protocols=protocol_
names)
fig1.savefig("plot_fit.svg", transparent=True)
# Plot MSEs Boxplot
fig2, axis2 = plt.subplots()
plot_mse_fig(axis2, mses)
fig2.savefig("plot_mses.svg", transparent=True)
# Plot Efficacy Kernel
```

```
fig3, axis3 = plt.subplots()

plot_kernel_easySRP(axis3, model)

fig3.savefig("plot_kernel.svg", transparent=True)
```

Alternate step: Manually define EasySRP model parameters

Timing: 10 min

The user manually selects and adjusts values for each parameter to fit the model to a synapse's post-synaptic responses.

Note: This protocol is best used in cases where the user believes manually setting and adjusting the parameters will best capture the synaptic dynamics or would like to experiment with the effects of different parameters.

21. Open an IDE. For more details on Spyder, the suggested IDE, refer to step 14.
22. Once in an IDE, import the necessary packages by executing the following lines of code:

```
import pickle

import numpy as np

import matplotlib.pyplot as plt

from pathlib import Path

from srplasticity.srp import easySRP

from srplasticity.srp_extrafuncs import (

    _EasySRP_dict_to_tuple,

easy_fit_srp,

    mse_loss,

    plot_fit_easySRP,

    plot_mse_fig,

    plot_kernel_easySRP,

    plot_srp_easySRP)
```

23. To import the preprocessed data and assign it to a variable, run the following code. The user's target dictionary file should replace the one used as an example here.

```
#example path construction

main_folder = Path("~").expanduser() #if srplasticity is in home directory

file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/chamberland2018_
data.p")

with open(file_path, "rb") as file:

            target_dict = pickle.load(file)
```

24. Assign to a variable (e.g., stimulus_dict) a dictionary in which each key represents the identification of a stimulation protocol, and each value is a list of its corresponding inter-spike intervals (ISIs). Use the following as an example:

```
stimulus_dict = {
    "20": [0] + [50] * 9,
    "100": [0] + [10] * 9,
    "20100": [0, 50, 50, 50, 50, 10],
    "10020": [0, 10, 10, 10, 10, 50],
    "10100": [0, 100, 100, 100, 100, 10],
    "111": [0] + [5] * 5,
    "invivo": [0, 6, 90.9, 12.5, 25.6, 9],
}
```

Optional: Define a dictionary called "protocol_names" in which the abbreviated names of the protocols are associated with their descriptive version. It is meant to be used later in plotting functions to set more descriptive titles. It is therefore optional, its absence only resulting in these titles being set to the abbreviated names of the protocols from "stimulus_dict".

```
protocol_names = { "100": "10 x 100 Hz",
    "20": "10 x 20 Hz",
    "111": "6 x 111 Hz",
    "20100": "5 x 20 Hz + 1 x 100 Hz",
    "10100": "5 x 10 Hz + 1 x 100 Hz",
    "10020": "5 x 100 Hz + 1 x 20 Hz",
    "invivo": "in-vivo burst",
}
```

25. Define a dictionary containing SRP model parameters:

```
params = {"mu_baseline": -2,
         "mu_amps": [10.24, -140.55, 350.49],
         "mu_taus": [15, 200, 300],
         "SD": None,
         "mu_scale": None}
```

Note: The user can change the time constants and amplitudes of the kernel.

26. Plot and save the predicted and recorded PSP or PSC amplitudes.

Note: Consider repeating this step after each adjustment made to the parameters to assess the resulting fit quality.

```
model = easySRP(**params)

mean_dict = {protocol : model.run_ISIvec(isivec)[0] for protocol, isivec in stimulus_
dict.items()}

total_mse_loss = mse_loss(target_dict, mean_dict)

params["SD"] = total_mse_loss ** 0.5

fig, ([ax1, ax2, ax3, ax4], [ax5, ax6, ax7, ax8]) = plt.subplots(2, 4, figsize=(10, 6))

axes = (ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8)

plot_srp_easySRP(axes, params, target_dict, stimulus_dict, protocols=protocol_names)

fig.savefig('plot_srp_easySRP.svg', dpi=1200, transparent=True, bbox_inches='tight')
```

27. Save the parameters:

```
params_to_save = _EasySRP_dict_to_tuple(params)

with open("srp_params.p", "wb") as file:

    pickle.dump(params_to_save, file)
```

Note: Consult the protocol titled "Fitting the SRP model to synaptic responses assuming a Gaussian distribution over the data" for an example of code when fitting multiple synapses.

Optional: Plot the predicted and recorded PSP or PSC amplitudes corresponding to a certain protocol, a boxplot of the mean squared errors (per protocol) of the fit, and the estimated efficacy kernel by using these lines:

```
# Compute MSEs

mses = []

for protocol, responses in target_dict.items():

    mse = mse_loss({protocol:responses}, mean_dict)

    mses.append(mse)

# Plot Fit

chosen_prot = "20"

fig1, axis1 = plt.subplots()

plot_fit_easySRP(axis1, model, target_dict, stimulus_dict, chosen_prot, protocols=protocol_
names)

fig1.savefig("plot_fit.svg", transparent=True)

# Plot MSEs Boxplot

fig2, axis2 = plt.subplots()

plot_mse_fig(axis2, mses)

fig2.savefig("plot_mses.svg", transparent=True)

# Plot Efficacy Kernel

fig3, axis3 = plt.subplots()
```

```
plot_kernel_easySRP(axis3, model)

fig3.savefig("plot_kernel.svg", transparent=True)
```

Alternate step: Fit an SRP model with history-dependent stochasticity

Timing: 20 min

Model parameters, including those corresponding to the variance kernel, are estimated based on a synapse's post-synaptic responses to stimulation.

Note: This protocol estimates parameters for both the efficacy and variance kernels, representing a distinct model compared to the previous ones. Here, responses are assumed to be gamma-distributed and with history-dependent variance. This allows the model to capture changes in response variances over time.

28. Open an IDE. For more details on Spyder, the suggested IDE, refer to step 14.
29. Import the necessary packages:

```
import pickle

import numpy as np

import matplotlib.pyplot as plt

from pathlib import Path

from srplasticity.srp import ExpSRP

from srplasticity.inference import fit_EXPSRP_model, _ExpSRP_tuple_to_dict

from srplasticity.srp_extrafuncs import (_EasySRP_dict_to_tuple,

     mse_loss,

     plot_mse_fig,

     plot_fit_ExpSRP,

     plot_kernel_ExpSRP,

     plot_srp_ExpSRP)
```

30. To import the preprocessed data and assign it to a variable, run the following code.

Note: The user's target dictionary file should replace the one used here as an example here.

```
#example path construction

main_folder = Path("~").expanduser() #if srplasticity is in home directory

file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/chamberland2018_
data.p")

with open(file_path, "rb") as file:

               target_dict = pickle.load(file)
```

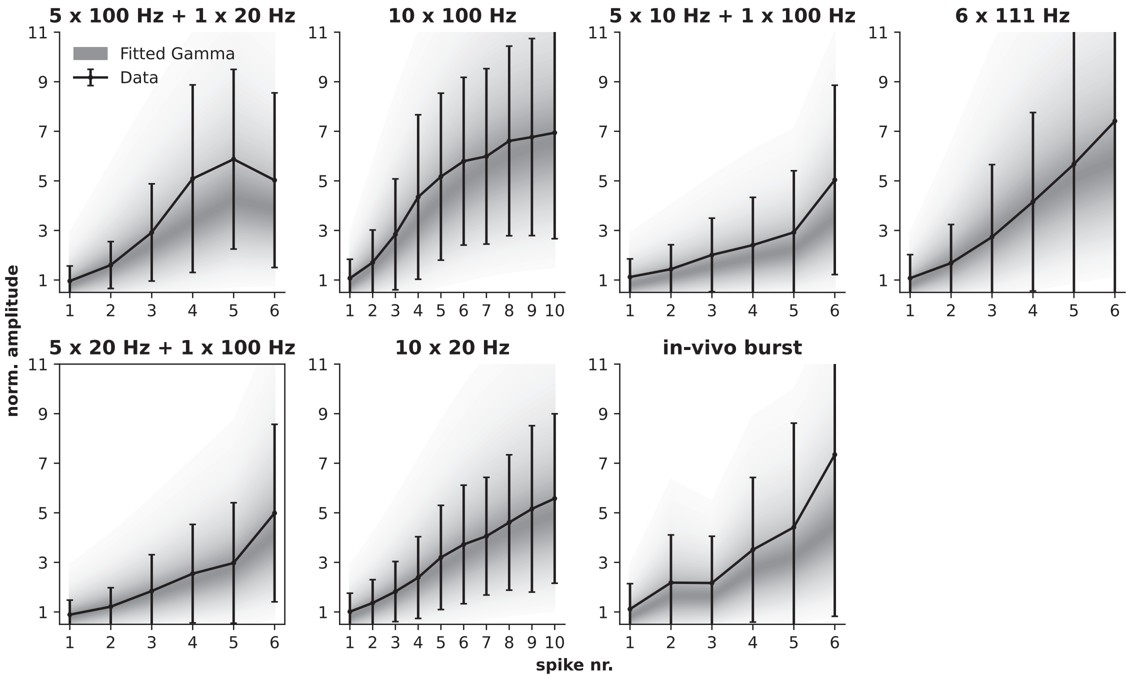


Figure 4. Example plot showing ExpSRP model predictions against experimental data

The shaded areas show the model's predicted gamma-distributed responses. The black lines and their error bars show the mean of the normalized recorded responses and their standard deviation. Data from Chamberland et al.[23]

31. Assign to a variable (e.g., stimulus_dict) a dictionary in which each key represents the identification of a stimulation protocol, and each value is a list of its corresponding inter-spike intervals (ISIs). Use the following as an example:

```
stimulus_dict = {
    "20": [0] + [50] * 9,
    "100": [0] + [10] * 9,
    "20100": [0, 50, 50, 50, 50, 10],
    "10020": [0, 10, 10, 10, 10, 50],
    "10100": [0, 100, 100, 100, 100, 10],
    "111": [0] + [5] * 5,
    "invivo": [0, 6, 90.9, 12.5, 25.6, 9],
}
```

Optional: Define a dictionary called "protocol_names" in which the abbreviated names of the protocols are associated with their descriptive version. It is meant to be used later in plotting functions to set more descriptive titles. It is therefore optional, its absence only resulting in these titles being set to the abbreviated names of the protocols from "stimulus_dict".

```
protocol_names = {
    "100": "10 x 100 Hz",
    "20": "10 x 20 Hz",
    "111": "6 x 111 Hz",
    "20100": "5 x 20 Hz + 1 x 100 Hz",
    "10100": "5 x 10 Hz + 1 x 100 Hz",
    "10020": "5 x 100 Hz + 1 x 20 Hz",
    "invivo": "in-vivo burst",
}
```

32. Define two variables as a list of the time constants corresponding to the efficacy and variance kernels:

```
mu_taus = [15, 200, 300]
sigma_taus = [15, 100, 300]
```

33. Obtain the fitted parameters by running the following line:

```
x0 = [0.1 for i in range(0, len(mu_taus)+len(sigma_taus)+3)]
srp_params = fit_EXPSRP_model(x0, stimulus_dict, target_dict, mu_taus, sigma_taus)[0]
params = _ExpSRP_tuple_to_dict(srp_params)
```

Optional: As an alternative function to "fit_EXPSRP_model", the user can try "fit_EXPSRP_model_gridsearch", which can be found in the "inference" module. This version uses a grid search, which tries multiple initializations and iterates over a large parameter space. This method is more computationally expensive and will require a longer run time but may yield better fits. Users can decrease runtime by allocating multiple cores to this function by setting the optional "workers" parameter (the default value is 1).

34. Save the parameters by using the following code:

```
with open("srp_params.p", "wb") as file:
    pickle.dump(params, file)
```

Optional: Plot and save the predicted and recorded PSP or PSC amplitudes by using the following code. These should appear similar to the plots shown in Figure 4.

```
fig, ([ax1, ax2, ax3, ax4], [ax5, ax6, ax7, ax8]) = plt.subplots(2, 4, figsize=(10, 6))
axes = (ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8)
plot_srp_ExpSRP(axes, params, target_dict, stimulus_dict, protocols=protocol_names)
fig.savefig('plot_srp_ExpSRP.svg', dpi=1200, transparent=True, bbox_inches='tight')
```

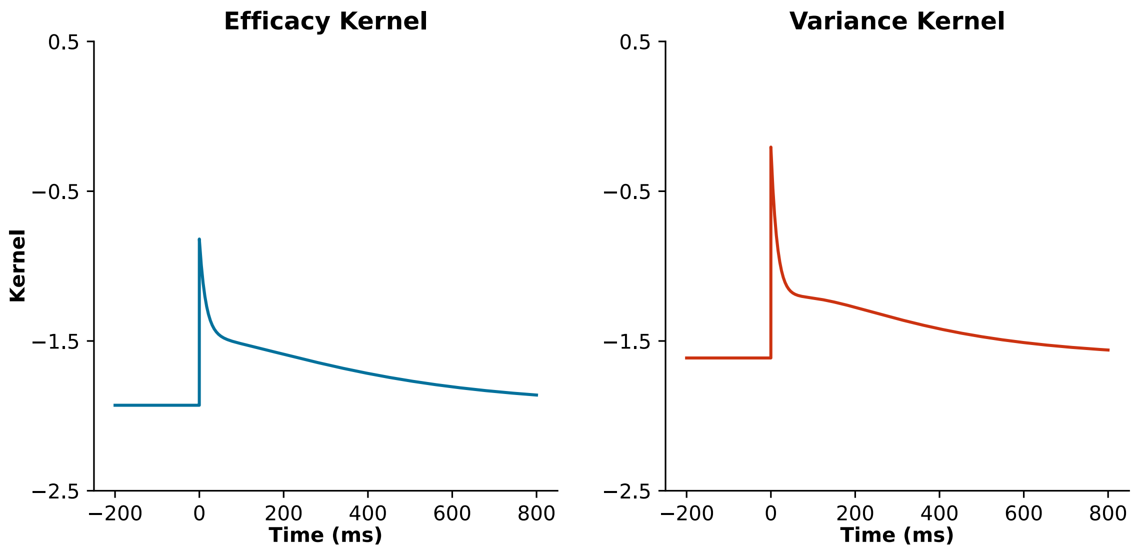


Figure 5. Example visualization of fitted ExpSRP kernels which can be used to help understand the inferred dynamics of the synapse

Optional: Plot the predicted and recorded PSP or PSC amplitudes corresponding to a certain protocol, a boxplot of the mean squared errors (per protocol) of the fit, and the estimated efficacy kernel by using the lines below. These should appear similar to the plots shown in Figures 4 and 5.

```
# Instantiate Model

model = ExpSRP(**params)

Exp_model = ExpSRP(**params)

# Compute MSEs

mean_dict = {protocol : model.run_ISIvec(isivec)[0] for protocol, isivec in stimulus_
dict.items()}

mses = []

for protocol, responses in target_dict.items():

     mse = mse_loss({protocol:responses}, mean_dict)

     mses.append(mse)

# Plot Fit

chosen_prot = "20"

fig1, axis1 = plt.subplots()

plot_fit_ExpSRP(axis1, model, target_dict, stimulus_dict, chosen_prot, protocols=protocol_
names)

fig1.savefig("plot_fit.svg", transparent=True)

# Plot MSEs Boxplot

fig2, axis2 = plt.subplots()

plot_mse_fig(axis2, mses)

fig2.savefig("plot_mses.svg", transparent=True)
```

```
# Plot Efficacy Kernel
fig3, (axis3, axis4) = plt.subplots(1, 2)
plot_kernel_ExpSRP(axis3, axis4, model)
fig3.savefig("plot_kernel.svg", transparent=True)
```

Evaluate model fit quality

Timing: 10 min

The mean squared errors of the model's predictions are computed.

35. Import the following:

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
from srplasticity.srp import easySRP
from srplasticity.srp_extrafuncs import (
    _EasySRP_tuple_to_dict,
    mse_loss,
    plot_mse_fig)
```

36. To import the preprocessed data and assign it to a variable, run the following code. The user's target dictionary file should replace the one used as an example here:

```
#example path construction
main_folder = Path("~").expanduser() #if srplasticity is in home directory
file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/chamberland2018_
data.p")
with open(file_path, "rb") as file:
            target_dict = pickle.load(file)
```

37. Assign to a variable (e.g., stimulus_dict) a dictionary in which each key represents the identification of a stimulation protocol, and each value is a list of its corresponding inter-spike intervals (ISIs). Use the following as an example:

```
stimulus_dict = {"20": [0] + [50] * 9,
    "100": [0] + [10] * 9,
    "20100": [0, 50, 50, 50, 50, 10],
    "10020": [0, 10, 10, 10, 10, 50],
    "10100": [0, 100, 100, 100, 100, 10],
     "111": [0] + [5] * 5,

     "invivo": [0, 6, 90.9, 12.5, 25.6, 9],

}
```

Optional: Define a dictionary called "protocol_names" in which the abbreviated names of the protocols are associated with their descriptive version. It is meant to be used later in plotting functions to set more descriptive titles. It is therefore optional, its absence only resulting in these titles being set to the abbreviated names of the protocols from "stimulus_dict."

```
protocol_names = {"100": "10 x 100 Hz",

     "20": "10 x 20 Hz",

     "111": "6 x 111 Hz",

     "20100": "5 x 20 Hz + 1 x 100 Hz",

     "10100": "5 x 10 Hz + 1 x 100 Hz",

     "10020": "5 x 100 Hz + 1 x 20 Hz",

     "invivo": "in-vivo burst",

}
```

38. Open the saved parameters and convert the tuple to a dictionary as seen in this example below:

```
file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/srp_params_exam-
ple.p") # expands to home directory

with open(file_path, "rb") as file:

              srp_params = pickle.load(file)

params = _EasySRP_tuple_to_dict(srp_params)
```

39. Compute the mean squared errors per protocol of all responses:

```
mean_dict = {protocol : model.run_ISIvec(isivec)[0] for protocol, isivec in stimulus_dict.
items()}

mses = []

for protocol, responses in target_dict.items():

     mse = mse_loss({protocol:responses}, mean_dict)

     mses.append(mse)
```

40. Create a boxplot of the mean squared errors:

```
fig1, axis1 = plt.subplots()

plot_mse_fig(axis1, mses)

fig1.savefig("plot_mses.svg", transparent=True)
```

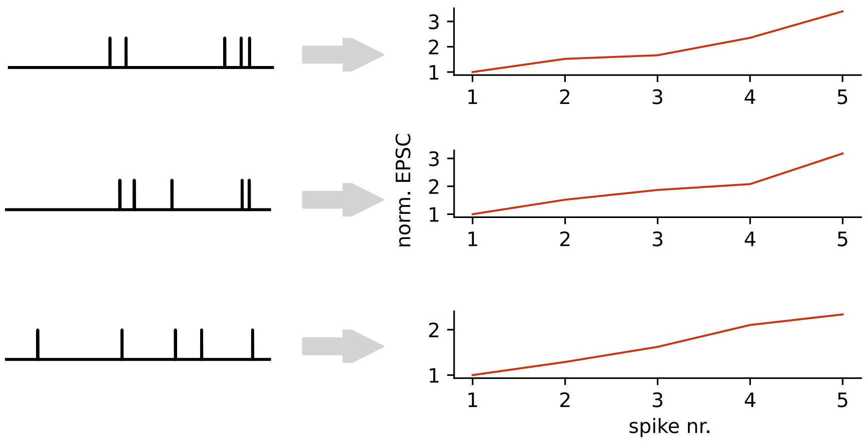


Figure 6. Example showing the use of the EasySRP model to predict responses to novel stimuli
Red line shows predicted mean response of an example fit to different spike trains (left).

Note: Exponentials as basis functions may sometimes be inadequate for capturing certain synaptic dynamics. One such STP type is delayed facilitation at the onset of stimulation.[26] We therefore encourage advanced users to explore the possibility of changing the basis function for another, such as a Gaussian, if needed. This would involve initializing the ProbSRP class with a Gaussian basis function created using the GaussianKernel class, for instance. The user can find additional code supporting this approach in the SRPlasticity package.

Simulate model response to novel stimuli

Timing: 10 min

Responses to novel stimuli are estimated and plotted alongside their corresponding spike trains.

Note: This part of the protocol is intended to be used on characterized synapses. Refer to steps 14 through 41 for instructions to infer appropriate model parameters for this characterization.

41. Import the necessary packages:

```
import pickle
import matplotlib.pyplot as plt
from pathlib import Path
from srplasticity.srp import easySRP
import matplotlib.patches as mpatches
from srplasticity.tools import get_stimvec
from srplasticity.srp_extrafuncs import (
    _EasySRP_tuple_to_dict,
    get_poisson_ISIs,
    plot_spike_train,
    plot_estimates)
```

42. Open the saved parameters and convert the tuple to a dictionary as seen below:

```
file_path = Path(f"{main_folder}/srplasticity/data/star_protocol_example/srp_params_
example.p") # expands to home directory

with open(file_path, "rb") as file:

              srp_params = pickle.load(file)

params = _EasySRP_tuple_to_dict(srp_params)
```

Note: The values assigned in this step are merely examples. Users can import their own inferred and saved model parameters from earlier stages of this protocol by using the `pickle.load()` function and matching the formatting and variable names to the code block above.

43. To stochastically generate spike trains containing n spikes (“nspikes”) at k frequency (“rate”) represented in a vector of inter-spike intervals named isi_vec and compute the synapse’s estimated responses to those spike trains, enter and run the following lines of code:

```
model = easySRP(**params)

isi_vec = list(get_poisson_ISIs(nspikes=30, rate=10))

spike_train = get_stimvec(isi_vec, dt=0.1, null=0, extra=0)

means, efficacies = model.run_ISIvec(isi_vec, ntrials=100)
```

Note: In the function “get_stimvec,” the parameter “dt” sets the interval (in ms) between points in the generated stimulation vector, “null” specifies the duration (in ms) of zeros to prepend to the stimulation vector, and “extra” defines the duration (in ms) of zeros to append after the last stimulus in the generated stimulation vector. “ntrials” is the number of sample runs to generate.

44. Use the following functions to plot and save the figures corresponding to the spike train and the estimated responses.

Note: These should appear similar (though not identical) to those in Figure 6 (which shows deterministic responses):

```
# Spike train

fig1, axis1 = plt.subplots()

plot_spike_train(axis1, spike_train)

fig1.savefig("spike_train.svg", transparent=True)

# Estimated peak responses to novel stimuli

fig2, axis2 = plt.subplots()

plot_estimates(axis2, means, efficacies)

fig2.savefig("sim_responses.svg", transparent=True)
```

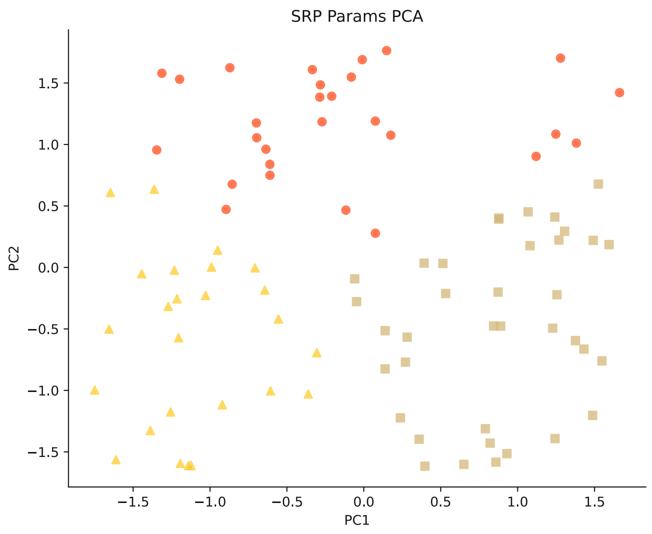


Figure 7. Visualization of how SRP model parameters can be used to assess which synapses are most similar and dissimilar using PCA and k-means clustering
Scatterplot shows classes generated to match different STP profiles in distinct colors and the identities assigned to the data by simple unsupervised machine learning (k-means clustering) as distinct shapes.

Visualization and cluster data

Timing: 20 min

It is possible to use the parameters of the fitted SRP model to infer the relative degree of functional difference or similarity between different synapses. Here the user is shown how to generate one possible two-dimensional visualization of STP from these parameters.

Note: In these final sections we intentionally expose more algorithmic detail so that users can see how the SRP model can be used alongside the Scikit-learn[21] package. Users will benefit from independently exploring the wide range of tools that package provides and should treat these sections as a bridging tutorial, not a complete guide.

45. For illustrative purposes we will use randomly generated SRP parameters to demonstrate visualization and clustering. Define the following function:

```
def gen_uniform_srp(param_bounds, num_fits):
    fitted_models = [[] for i in range(0, num_fits)]
    for i in range(0, num_fits):
        for j in range(0, len(param_bounds)):
            param=np.random.default_rng().uniform(
                low=param_bounds[j][0], high=param_bounds[j][1])
            fitted_models[i].append(param)
    return np.asarray(fitted_models)
```

Note: This section assumes you have already imported the packages described in preceding sections. At minimum Numpy and Matplotlib must be imported.

46. Now define the following SRP bounds to illustrate a case with a population of traditionally facilitating synapses, another population of traditionally depressing synapses, and a third possible ambiguous "transition" population.

```
#classically depressing
rand_bounds1 = [
     (4, 6), # mu baseline
     *[(-150, 0), (-1000, 0), (-3000, 0)], #mu amplitude bounds
     (0, 3) #SD range
]
#Classically facilitating
rand_bounds2 = [
     (-6, -4), # mu baseline
     *[(0, 150), (0, 1001), (0, 3000)], #mu amplitude bounds
     (0, 3) #SD range
]
#generic range
rand_bounds3 = [
     (-3, 3), # mu baseline
     *[(-50, 50), (-500, 500), (-1500, 1500)], #mu amplitude bounds
     (0, 3) #SD range
]
```

Note: We encourage users to experiment with different bounds and repeat the final two sections several times for different randomly generated values. The output and associated quality measures will likely be somewhat different every time because, much like in experiment conditions, we only sample a finite number of examples from the populations above.

47. Finally, call the gen_uniform_srp function defined above for each set of bounds and merge the results into a Numpy array while recording the original identity of each randomly generated synapse in a second Numpy array: **Note**: You can simply substitute your own real fits by assigning them to the "all_params" variable and continuing with the steps below.

```
#generate random fits
rand_params1 = gen_uniform_srp(rand_bounds1, 30)
rand_params2 = gen_uniform_srp(rand_bounds2, 30)
rand_params3 = gen_uniform_srp(rand_bounds3, 30)
#labels
rand_labels1 = np.asarray([1 for i in range(0, 30)])
rand_labels2 = np.asarray([2 for i in range(0, 30)])
```

```
rand_labels3 = np.asarray([3 for i in range(0, 30)])

#merge into one array

all_params = np.concatenate((rand_params1, rand_params2, rand_params3))

all_labels = np.concatenate((rand_labels1, rand_labels2, rand_labels3))
```

48. Ensure you have imported the following packages:

```
from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans
```

49. Use principal component analysis as implemented by the Scikit-learn package[21] to reduce the model parameters to essential features that capture the greatest linearly explainable variation in the data.

```
pca = PCA(whiten=True)

scaler = StandardScaler()

scaler.fit(all_params)

scaled_arr = scaler.transform(all_params)

pca_result = pca.fit_transform(scaled_arr) #scaled
```

Optional: You can now conveniently use a clustering algorithm from Scikit-learn to infer similar groupings in your data. To configure the K-Means algorithm for this purpose run the following code:

```
km = KMeans(n_clusters = 3)
```

Optional: Now gather inferred group labels from KMeans using the two most explanatory features from PCA:

```
cluster_labels = km.fit_predict(pca_result[:, 0:2])
```

Optional: Define marker shapes for plotting to indicate which group KMeans has assigned each synapse to

```
shapes = {-1: "D", 0: "o", 1: "^", 2: "s", 3: "*"}

colours = {-1:"xkcd:dark blue", 0: "xkcd:red orange", 1: "xkcd:golden yellow", 2: "xkcd:
tan", 3: "xkcd:light grey"}
```

50. Finally, visualize your data by plotting them in the first two dimensions given by PCA.

Note: Each point corresponds to a synapse, colors correspond to "true" labels, and shapes indicate the guesses made by KMeans (see Figure 7):

```
f = plt.figure()
ax = plt.subplot(111)
for index, fit in enumerate(pca_result):
    plt.scatter(fit[0], fit[1], marker=shapes[cluster_labels[index]],
                     color=colours[all_labels[index]], alpha=0.7)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.title("SRP Params PCA")
plt.xlabel("PC1")
plt.ylabel("PC2")
f.tight_layout()
plt.show()
```

Note: Because data were randomly generated your plot will not look identical to the one above.

Make model-based predictions

Timing: 20 min

In addition to visualization and clustering you can also use fitted SRP parameters to predict known labels for the synapses such as postsynaptic cell-type (see Beninger et al. 2024[2]). In this case, the accuracy of the predictions can serve as a lower bound metric for the degree to which the set of labels corresponds to functionally distinct types of synapses.

51. For illustrative purposes we will once again rely on randomly generated data to demonstrate prediction.

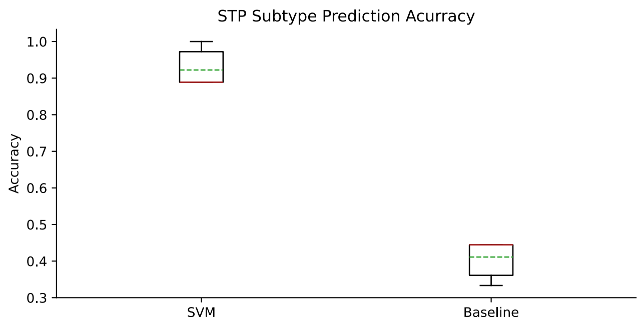


Figure 8. Example boxplot showing how accurately a set of SRP model parameters can support prediction of known labels or features
Y-axis values correspond to test set (unseen) predictive accuracies assesed using stratified cross-validation.

⚠ CRITICAL: If you have not already, repeat steps 48 to 50. Alternatively, you can assign your data to a variable named "all_params" and assign a numpy array of the labels you wish to predict (with parameter-label pairs having the same row number in both arrays) to a variable named "all_labels". For the random example this can be assigned to match the random cluster labels generated above:

```
all_labels = cluster_labels
```

52. Import packages with tools for performing and managing supervised machine learnings:

```
from sklearn.model_selection import StratifiedKFold

from sklearn.svm import SVC

from sklearn.dummy import DummyClassifier
```

53. To robustly estimate predictive accuracy, users should repeatedly fit a predictive model on one subset of their data and then test its performance on another distinct subset. First, create a list to hold the accuracy values for each of these tests.

```
accuracies = []
```

54. To know how likely a model was to get a certain accuracy value by chance, create another list to record these chance level accuracy "baselines".

```
baselines = []
```

55. Use the imported tools to repeatedly "train" (i.e., fit) a supervised machine learning model called a support vector machine (SVC) and a tool for inferring baseline accuracy values called "DummyClassifier":

```
skf = StratifiedKFold(n_splits=10, shuffle=True)

for i, (train_index, test_index) in enumerate(skf.split(pca_result, all_labels)):

    train_data, train_targets = pca_result[train_index], all_labels[train_index]

    test_data, test_targets = pca_result[test_index], all_labels[test_index]

    clf = SVC()

    clf.fit(train_data, train_targets)

    accuracy = clf.score(test_data, test_targets)

    accuracies.append(accuracy)

    dummy_clf = DummyClassifier(strategy='most_frequent')

    dummy_clf.fit(train_data, train_targets) #model like comparison
    baseline = dummy_clf.score(test_data, test_targets)
    baselines.append(baseline)
```

56. Finally, generate a boxplot containing the accuracy and baseline values you have computed using the code below.

    Note: If you used the random data generation method we presented, then you should find accuracy values that are much higher than baseline accuracy but are lower than 100% because of the ambiguous "transition" class of synapses you have generated. The result should look similar to Figure 8:

    Note: Because data were randomly generated your plot will not look identical to the one above.

```
#plot supervised
y_vals = [accuracies, baselines]
x_coordinates = [i for i in range(1, len(y_vals)+1)]
f = plt.figure()
ax = plt.subplot(111)
medianprops = dict(linestyle=None, color='firebrick')
plt.boxplot(y_vals, showmeans=True, meanline=True, medianprops=medianprops)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.title("STP Subtype Prediction Accuracy")
plt.xticks(x_coordinates, ["SVM", "Baseline"])
plt.ylabel("Accuracy")
f.tight_layout()
plt.show()
```

EXPECTED OUTCOMES

Using our full protocol allows users to infer model parameters from real electrophysiological data, assess how well the parameterized model corresponds to their data, and perform both simulations and advanced analysis using the inferred parameters. Specifically, users should be able to generate figures showing how well model predictions correspond to actual data, kernel representations of the inferred dynamics, predicted responses to novel stimuli, a visualization of relative similarity, and an assessment of the degree to which STP is predictive of a known identifier. Figure 9 shows an example of the kind of aggregated modeling figure users will be able to compile based on these steps.

LIMITATIONS

The SRP model is first and foremost meant as a tool for theoretical analysis (i.e., characterization of synaptic dynamics, network simulation, etc.). Thus, little information regarding molecular mechanisms and their relative importance can be obtained from the model. This is due to a trade-off between phenomenological models, such as this one, and biophysical models: to capture the essence of complex experimental data while avoiding over-parameterization and degeneracies, some of the

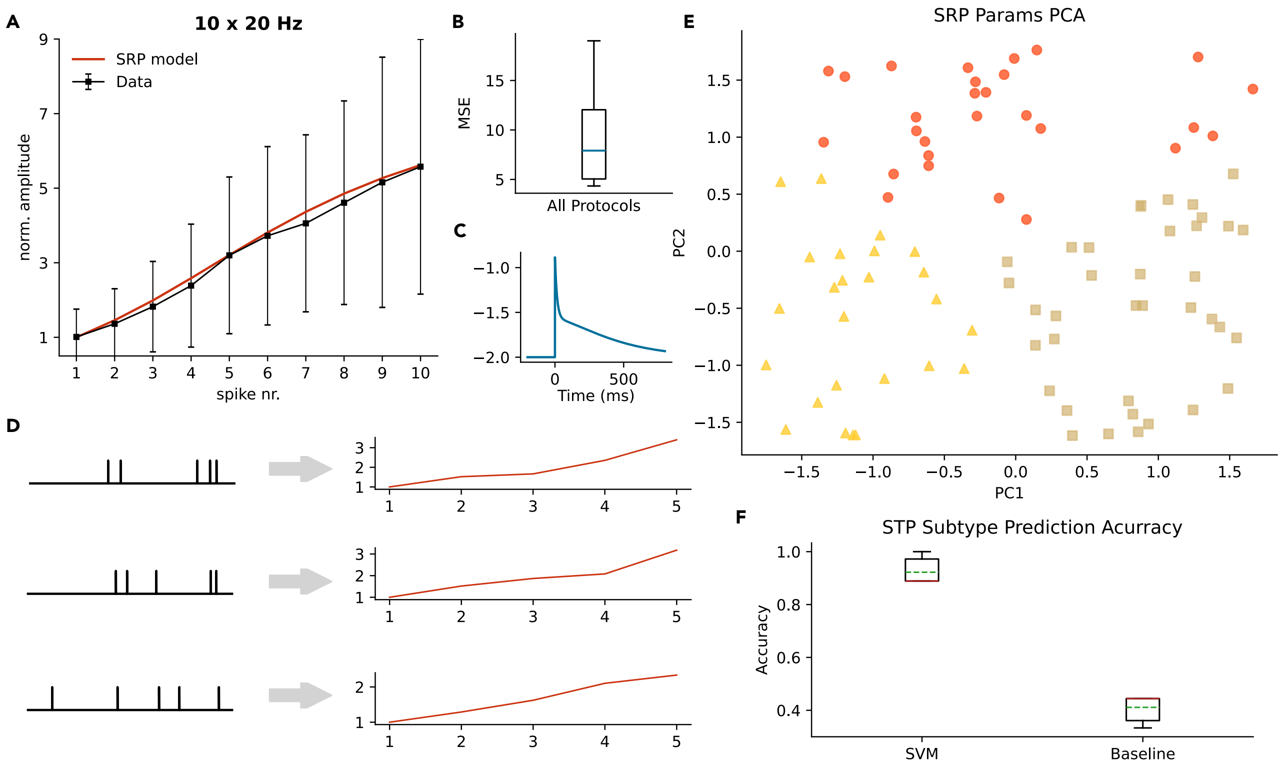


Figure 9. Aggregated plot (based on Figures 1; 2, 3; 6; 7; 8) showing an example of the kind of overall modeling and analysis figure the user can generate from their STP data

A. Example plot showing EasySRP model predictions against experimental data. The red line shows the model's predicted mean response. The black line and the error bars show the mean of the normalized recorded responses and their standard deviation. Data from Chamberland et al.[23] B. Example boxplot showing distribution of EasySRP model fitting error. C. Example visualization of fitted EasySRP kernel. D. Example showing the use of the EasySRP model to predict responses to novel stimuli. E. Visualization of how SRP model parameters can be used to assess which synapses are most similar and dissimilar using PCA and k-means clustering. Scatterplot shows classes generated to match different STP profiles in distinct colors and the identities assigned to the data by simple unsupervised machine learning (k-means clustering) as distinct shapes. F. Example boxplot showing how accurately a set of SRP model parameters can support prediction of known labels or features. Y-axis values correspond to test set (unseen) predictive accuracies assesed using stratified cross-validation.

model's interpretability must be sacrificed.[1] The user should ensure their goals align with this limitation before proceeding.

As with similar analyses, the quality of the electrophysiological data and their processing affect the model's ability to capture synaptic dynamics. Noise, experimental artifacts, intrinsic variability, and extreme responses can distort post-synaptic traces. Before beginning this protocol, users will benefit from visualizing their raw data and performing general quality control. For example, to ensure proper fitting, boundaries can be established, above and below which responses would be deemed outliers. The user can also use curve fitting to calculate PSPs from raw traces and compute the amplitudes from those fits. An example of how to do this can be found in Seeman, Campagnola et al.[22] Fitting quality is also limited by the amount of experimental data available, with more data points associated with a particular synapse or synapse type leading to better characterization of that synapse and as a result, greater predictive power.

Studies have highlighted a likely correspondence between a cell's transcriptome and subtypes of synaptic dynamics.[2,6] Previously, we have observed a partial alignment between groups of excitatory synapses defined by similar synaptic dynamics and underlying transgenically defined cell identities.[2] However, genetically defined neuronal identities include multiple hierarchical levels of description[27] and popular levels of description group cells with heterogeneous synaptic dynamics.[6]

As such, we posit that finer-grain cell identities will support even stronger alignment with synaptic features. Conversely, we expect that any given level of genetic description will impose upper bounds on how accurately genetic identity can be predicted from synaptic dynamics.

During the fitting procedure, the user may present the software with dynamics that the model has difficulty fitting. When dealing with fitting issues we recommend that the user first try adjusting the procedure by changing initial values of the model. In the case of the easySRP model the easiest parameters to adjust are the exponential decay time constants in "mu_taus". That said, despite the model's flexibility, it is bound to experience difficulties when confronted with unusual or unexpected dynamics. While these methods are beyond the scope of this protocol, for more advanced users, the easySRP model has additional tunable parameters, and our code can be modified to use different fitting procedures. The SRPlasticity package also contains other model class definitions that can be used directly or as parent classes for object inheritance. For example, it is possible to change the basis function (e.g., exponential, Gaussian, etc.) or override sampling methods to change stochastic behavior. For most users, the easy_fit_srp function achieves good performance for large datasets by iterating through different ranges of the mu_baseline while searching for parameters using the implementation of simplicial homology global optimization (SHGO)[28] provided by SciPy.[17] We have found this strategy to yield the best balance of speed and performance for "out of the box" fitting.

TROUBLESHOOTING

Problem 1

There is a problem for which this manuscript does not contain a solution.

Potential solution

To address potential issues users might encounter when running the code, we recommend visiting the SRPlasticity GitHub repository. A link to this repository can be found in the key resources table. There, users can check if others have raised similar issues and found solutions, create new issues, or submit pull requests if they have solutions in mind. Users should also consider updating to the latest version of the code, which may resolve their problems. Finally, the repository includes a folder containing scripts that were used to generate the figures in Rossbroich et al.[1] Users may find these scripts useful as examples or templates for additional applications of the package.

Problem 2

The SRPlasticity package is not importable.

Potential solution

To install the SRPlasticity package correctly, the user needs to use the right Python installation. This installation should correspond to the one installed by package managers (e.g., conda) and not the one built-in. It is important to install the package in the environment created by the former, and not the latter.

To locate the binary associated with the Python installation being used, open Anaconda Prompt and activate the virtual environment (i.e., "conda activate srplasticity"). Then, run the line "where python" (on Windows systems) or "where python3" on UNIX-based systems. This will return a directory, which will indicate whether a python executable (i.e., python.exe) is inside the environment. If none appear, try "conda install python = 3.8." If one appears, but the package is still not importable, open your IDE. Search for the Python interpreter tab, which likely will be located within the "Settings" or "Tools" (Spyder) folders of most IDE. Make sure the Python interpreter used corresponds to the one used in the virtual environment. For more information about the correct binary to use and the Python installation in use, refer to your chosen IDE or package manager.

Problem 3

Abnormally high error on a fit (MSE > $\sim$1000).

Potential solution

Inherent variability in responses means some synapses may be harder to fit while yielding a low MSE than others. A rule of thumb is that fits with MSEs below 1 can be considered well-fitted. However, we have encountered in the past fits resulting in MSEs being abnormally high (e.g., well above 1000). In most cases, this is a consequence of the normalization process. Indeed, if the first response of a run or multiple runs before normalization is close to 0 (e.g., if the postsynaptic neuron failed to respond), the average of the first responses can be quite low as well. Therefore, dividing all responses by this value can yield normalized responses that are abnormally large and onto which the SRP model can't be fitted. Unfortunately, this is an inherent flaw of the normalization procedure, and the simplest workaround is to consider removing those problematic runs if the user judges they aren't representative of the synapse's dynamics, or to remove the synapse altogether from the user's project as it can't be properly characterized. The alternative solution is to modify the maximum likelihood approach so as to take non-normalized data as described in Problem 4.

Problem 4

Average of first responses post-normalization is not 1.

Potential solution

When the "mu_scale" parameter is set to None (as is the case in this protocol), the synaptic efficacy (i.e., the relative amplitude) of a specific spike in the train is normalized to the first spike in the train. In this case, the fit won't be able to accurately capture the dynamics if the recorded responses aren't properly normalized (i.e., the average of the first responses is not 1 after normalization). If this average is not 1, the user can refer to steps 9–13 to normalize their data. Otherwise, it is possible to fit the model to non-normalized responses by setting "mu_scale", for further details see Rossbroich et al.[1]

Problem 5

Parameters getting "stuck" at bounds during optimization.

Potential solution

Early implementations of the model encountered some issues with parameters getting "stuck" at the bounds defined by the "_default_parameter_bounds" function (if no customized bounds are used as input). We found that iterating through subsets of the baseline parameter space (as is the case in "easy_fit_srp") significantly decreased the probability of that phenomenon occurring. Therefore, it could be worthwhile to try in those cases iterating over subsets of other parameter spaces such as "mu_amps," or try a grid search using different starting points. In some cases, the user might notice that the kernel generated is stuck at the upper baseline, with high amplitudes, resulting in the model predicting static responses (i.e., no changes or very small changes in amplitudes across spikes). If the user believes that the model should be capturing a certain dynamic, then changes could be made to the basis function behind the kernels (see limitations). However, this subject is beyond the scope of this protocol as it only pertains to advanced users.

RESOURCE AVAILABILITY

Lead contact

Further information and requests for resources should be directed to and will be fulfilled by the lead contact, Richard Naud (rnaud@uottawa.ca).

Technical contact

Technical questions on executing this protocol should be directed to and will be answered by the technical contact, John Beninger (jgbeninger@gmail).

Materials availability

This study did not generate any novel reagents.

Data and code availability

The most up to date form of the software described in this package can be found at https://github.com/nauralcodinglab/srplasticity. We have also deposited an archived version of the package at Zenodo: https://doi.org/10.5281/zenodo.14775430.

ACKNOWLEDGMENTS

This research was undertaken thanks in part to funding from the Canada First Research Excellence Fund for the Brain-Heart Interconnectome (CFREF-2022-00007). J.B. was partially supported by an NSERC Doctoral Scholarship and an Ontario Graduate Scholarship (OGS). This research was funded in part by CIHR Discovery Grant RN38364. We thank Katalin Tóth for productive discussions. Our graphical abstract was created using the following resources from Flaticon.com: "Flaticon Knowledge free icon," "Flaticon Predictive free icon," "Flaticon Satisfaction free icon," "Flaticon Engineering free icon," and "Cluster free icon."

AUTHOR CONTRIBUTIONS

Conceptualization: R.N. Data curation: J.B. and J.P. Formal analysis: J.B. and J.P. Funding acquisition: R.N. and J.B. Investigation: J.B. and J.P. Methodology: R.N., J.B., and J.P. Project administration: J.B. and J.P. Resources: R.N. Software: J.B. and J.P. Supervision: R.N. and J.B. Validation: J.P. and J.B. Writing – original draft: J.P. and J.B. Writing – review and editing: J.P., J.B., and R.N.

DECLARATION OF INTERESTS

The authors declare no competing interests.

REFERENCES

1. Rossbroich, J., Trotter, D., Beninger, J., Tóth, K., and Naud, R. (2021). Linear-nonlinear cascades capture synaptic dynamics. PLoS Comput. Biol. *17*, e1008013. https://doi.org/10.1371/journal.pcbi.1008013.
2. Beninger, J., Rossbroich, J., Tóth, K., and Naud, R. (2024). Functional subtypes of synaptic dynamics in mouse and human. Cell Rep. *43*, 113785. https://doi.org/10.1016/j.celrep.2024.113785.
3. Feng, T. (1941). The changes in the end-plate potential during and after prolonged stimulation. Chin. J. Physiol. *13*, 79–107.
4. Eccles, J.C., Katz, B., and Kuffler, S.W. (1941). Nature of the ""endplate potential" in curarized muscle. J. Neurophysiol. *4*, 362–387. https://doi.org/10.1152/jn.1941.4.5.362.
5. Magleby, K.L., and Zengel, J.E. (1975). A dual effect of repetitive stimulation on post-tetanic potentiation of transmitter release at the frog neuromuscular junction. J. Physiol. *245*, 163–182. https://doi.org/10.1113/jphysiol.1975.sp010839.
6. Campagnola, L., Seeman, S.C., Chartrand, T., Kim, L., Hoggarth, A., Gamlin, C., Ito, S., Trinh, J., Davoudian, P., Radaelli, C., et al. (2022). Local connectivity and synaptic dynamics in mouse and human neocortex. Science *375*, eabj5861. https://doi.org/10.1126/science.abj5861.
7. Díaz-Quesada, M., Martini, F.J., Ferrati, G., Bureau, I., and Maravall, M. (2014). Diverse thalamocortical short-term plasticity elicited by ongoing stimulation. J. Neurosci. *34*, 515–526. https://doi.org/10.1523/JNEUROSCI.2441-13.2014.
8. De Pasquale, R., and Sherman, S.M. (2011). Synaptic properties of corticocortical connections between the primary and secondary visual cortical areas in the mouse. J. Neurosci. *31*, 16494–16506. https://doi.org/10.1523/JNEUROSCI.3664-11.2011.
9. Sherman, S.M. (2012). Thalamocortical interactions. Curr. Opin. Neurobiol. *22*, 575–579. https://doi.org/10.1016/j.conb.2012.03.005.
10. Pala, A., and Petersen, C.C. (2015). In vivo measurement of cell-type-specific synaptic connectivity and synaptic transmission in layer 2/3 mouse barrel cortex. Neuron *85*, 68–75. https://doi.org/10.1016/j.neuron.2014.11.025.
11. Ghanbari, A., Malyshev, A., Volgushev, M., and Stevenson, I.H. (2017). Estimating short-term synaptic plasticity from pre-and postsynaptic spiking. PLoS Comput. Biol. *13*, e1005738. https://doi.org/10.1371/journal.pcbi.1005738.
12. Ghanbari, A., Ren, N., Keine, C., Stoelzel, C., Englitz, B., Swadlow, H.A., and Stevenson, I.H. (2020). Modeling the short-term dynamics of in vivo excitatory spike transmission. J. Neurosci. *40*, 4185–4202. https://doi.org/10.1523/JNEUROSCI.1482-19.2020.
13. Friedenberger, Z., Harkin, E., Tóth, K., and Naud, R. (2023). Silences, spikes and bursts: Three-part knot of the neural code. J. Physiol. *601*, 5165–5193. https://doi.org/10.1113/JP281510.
14. Lynn, M.B., Geddes, S., Chahrour, M., Maillé, S., Caya-Bissonnette, L., Harkin, E., Harvey-Girard, E., Haj-Dahmane, S., Naud, R., and Béïque, J.-C. (2024). Nonlinear computation by a habenula-driven recurrent inhibitory network in the raphe. Preprint at bioRxiv. https://doi.org/10.1101/2022.08.31.506056.
15. Tsodyks, M., Pawelzik, K., and Markram, H. (1998). Neural networks with dynamic synapses. Neural Comput. *10*, 821–835. https://doi.org/10.1162/089976698300017502.
16. van der Walt, S., Colbert, S.C., and Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. Comput. Sci. Eng. *13*, 22–30. https://doi.org/10.1109/MCSE.2011.37.
17. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nat. Methods *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2.
18. Hunter, J.D. (2007). Matplotlib: A 2D Graphics Environment. Comput. Sci. Eng. *9*, 90–95. https://doi.org/10.1109/MCSE.2007.55.
19. McKinney, W. (2010). Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference, *445*, pp. 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.
20. The pandas development team (2024). Pandas-dev/pandas: Pandas. Zenodo. https://doi.org/10.5281/zenodo.13819579.
21. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine Learning in Python. J. Mach. Learn. Res. *12*, 2825–2830. https://doi.org/10.48550/arXiv.1201.0490.
22. Seeman, S.C., Campagnola, L., Davoudian, P.A., Hoggarth, A., Hage, T.A., Bosma-Moody, A., Baker, C.A., Lee, J.H., Mihalas, S., Teeter, C., et al. (2018). Sparse recurrent excitatory connectivity in the microcircuit of the adult mouse and human cortex. Elife *7*, e37349. https://doi.org/10.7554/eLife.37349.
23. Chamberland, S., Timofeeva, Y., Evstratova, A., Volynski, K., and Tóth, K. (2018). Action potential counting at giant mossy fiber terminals gates information transfer in the hippocampus. Proc. Natl. Acad. Sci. USA *115*, 7434–7439. https://doi.org/10.1073/pnas.1720659115.
24. Waskom, M.L. (2021). seaborn: statistical data visualization. J. Open Source Softw. *6*, 3021. https://doi.org/10.21105/joss.03021.

25. Rossbroich, J. (2020). Spiffyplots. GitHub. https://github.com/JRBCH/spiffyplots.

26. Neubrandt, M., Olah, V.J., Brunner, J., Marosi, E.L., Soltesz, I., and Szabadics, J. (2018). Single bursts of individual granule cells functionally rearrange feedforward inhibition. J. Neurosci. *38*, 1711–1724. https://doi.org/10.1523/JNEUROSCI.1595-17.2018.

27. Bugeon, S., Duffield, J., Dipoppa, M., Ritoux, A., Prankerd, I., Nicoloutsopoulos, D., Orme, D., Shinn, M., Peng, H., Forrest, H., et al. (2022). A transcriptomic axis predicts state modulation of cortical interneurons. Nature *607*, 330–338. https://doi.org/10.1038/s41586-022-04915-7.

28. Endres, S.C., Sandrock, C., and Focke, W.W. (2018). A simplicial homology algorithm for Lipschitz optimisation. J. Global Optim. *72*, 181–217. https://doi.org/10.1007/s10898-018-0645-y.