NeuroField User Manual

Complex Systems
School of Physics
University of Sydney

March 31, 2015

`NeuroField` is a `C++` program (accompanied with helper scripts) that solves the neural field model of Robinson et al., where each of the simultaneous equations are handled by an object:

$$P = \nu_{ab}\phi_{ab}, \qquad \texttt{Couple}$$
$$D_{ab}V_{ab} = P, \qquad \texttt{Dendrite}$$
$$Q_a = S_a\big[\sum_b V_{ab}\big], \qquad \texttt{QResponse}$$
$$\mathcal{D}_{ab}\phi_{ab} = Q_b. \qquad \texttt{Propag}$$

`NeuroField` generalizes the neural field theory by allowing users to:

1. Specify an arbitrary population model: an arbitrary number of populations and connections them may be specified;
2. Choose different types of populations, including neural or stimulus populations. For each neural population, the type of firing response, dendritic response type may be specified.
3. Choose different types of connections, including the type of axonal propagation, and synaptic coupling.
4. For each object, specify the parameter values.

This users guide covers the obtaining and setting up (Sec. 1), configuring (Sec. 2.1) and launching of `NeuroField` (Sec. 2), as well as postprocessing (Sec. 3) and tips and tricks (Sec. **??**).

Within this documentation, specific terminology as appeared in the computer is in `typewriter font` . Commands are denoted as

```
Command to put in computer
```

Contents

1 Obtaining and setting up NeuroField

1.1 Obtaining NeuroField

The code for `NeuroField` is managed by the version control system `git` , and is presently hosted on GitHub together with documentation, bug reports and feature requests. To obtain access to the repository, please send an email to Romesh Abeysuriya (`r.abeysuriya@physics.usyd.edu.au`).

To set up the latest version of `NeuroField` , execute

```
git clone git@github.com:RomeshA/neurofield.git
```

after having added your SSH key to your GitHub account. Alternatively, you can install the GitHub desktop client and click the 'Clone in Desktop' on the GitHub website. For additional instructions, check the GitHub tutorials for downloading repositories - the NeuroField repository can be downloaded following GitHub standard procedures.

1.2 Directory layout

After downloading NeuroField, the user can find:

`src/`	C++ source code.
`obj/`	All compiled object files. This directory will be deleted by `make clean`, so user data should not be stored here.
`bin/`	The compiled binary `neurofield` is created here.
`Configs/`	Stores example configuration files for `NeuroField`.
`Documentation/`	Contains documentations `Documentation/user.pdf` and `Documentation/developer.pdf`. Running `make doc` generates these documentations.
`Helper_scripts/`	Stores helper scripts, including plotting routines and other post-processing of data procedures.
`Matlab_structures/`	A supplementary set of Matlab files that are tailored for the Robinson et. al. corticothalamic model. These will be merged into the `Helper_scripts` folder in the near future.
`Output/`	When using the launcher script to sweep over parameters, the launcher script produces this directory, which stores all output files `neurofield.*` in independent subdirectories.
`Test/`	Directory for development testing and is irrelevant for users.

1.3 Compiling NeuroField

You can compile NeuroField simply by running

```
make
```

from the root directory containing all of the source files. The compiler command is specified in `Makefile` and should be edited if you wish to use a different compiler, or if your compiler does not support some of the compiler flags.

If you are compiling on Windows, the suggested route is to cross-compile using MinGW on a Unix-like system, and then copy across any missing DLLs from the Unix system into the same directory as the executable file on the Windows machine. Compiling with the Visual C++ compiler has not been tested.

Other build targets are available:

- To delete the binary files and temporary LaTeX files, run

  ```
  make clean
  ```

  which will delete the `bin` and `obj` folders, as well as files produced by LaTeXin `Documentation/`.

- This documentation can be generated by running

  ```
  make doc
  ```

which generates `./Documentation/user.pdf` and `./Documentation/developer.pdf`. You should not need to use these directly because pre-compiled PDF files are distributed with NeuroField.

2 Running NeuroField

You can run NeuroField directly using

```
./bin/neurofield
```

from the main NeuroField folder. For ease of use, you may consider adding the `neurofield` binary to your system path.

By default, NeuroField will check if a configuration file called `neurofield.conf` exists in the current directory. If this file exists, NeuroField will run it. Similarly, by default NeuroField will write output to the file `neurofield.output` in the current directory. When these default files are used, a warning is displayed:

```
romesha@romeshalt: neurofield > ./bin/neurofield
Warning: Using neurofield.conf for input by default
Warning: Using neurofield.output for output by default
```

Note that NeuroField will overwrite the output file if it exists, so it is not a good idea to use `neurofield.output` for your work.

You can optionally specify input and output files, using the `-i` and `-o` switches. For example,

```
./bin/neurofield -i Configs/example.conf -o example.output
```

will use the configuration file `example.conf` within the `Configs` folder, and will write output to `example.output` in the current folder.

A list of available switches can be displayed by using the `-h` or `--help` option i.e., `./bin/neurofield -h`

2.1 Writing a configuration file

`NeuroField` allows an arbitrary number of populations and connections between them, with all objects taking arbitrary parameter values. These are all configured via a configuration file. This section documents the specifications of configuration files, where we use `Configs/example.conf` as an illustrative example.

To write a configuration file, follow these steps:

1. Determine your population model by drawing a schematic diagram, thereby constructing a connection matrix. An example is shown in Fig. **??**.
2. Look up existing configuration files in `Configs/` . By checking the comment located at the top of a configuration file, and also the connection matrix, a user should find the most suitable existing file to construct his own. This is less tedious (and less error prone) than writing a new one from scratch.
3. Specify the global parameters and connectivity matrix (Sec. 2.2).
4. Specify all populations (Sec. 4.2.2).
5. Specify all propagators (Sec. 2.4).
6. Specify all couples (Sec. 4.2.4).
7. Specify all output requests (Sec. 4.4).

General rules on the entries within a configuration file:

1. The structure of a configuration file is: 1) comment 2) global information 3) object specification 4) output specification.
2. Object specification involves first specifying the object type. The syntax is the object identifier, followed by its type, then a hyphen:

   ```
   Object 1: Type -
   ```

   Then the object parameters are specified, following this pattern:

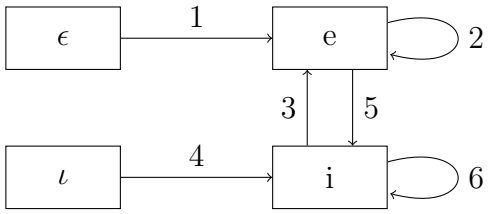
   ```
   Object parameter: value
   ```

3. Most parameters are essential. Failure to provide these parameters would result in `NeuroField` terminating with an error message. A minority of the parameters are optional.
4. The ordering of the parameters are important. Wrong parameter ordering results in `NeuroField` terminating with an error message.
5. The configuration file is white-space independent, e.g., there can be either no spaces, many spaces, or new lines between parameters.
6. For readability, users are encouraged to arrange parameter entries for different objects (via new lines and indentations) and aligning corresponding parameters between different objects.
7. Tip for `vi` users: `./Helper_scripts/neurofield.vim` implements syntax highlighting for configuration files in `vi` . See comments within for installation instructions.

2.1.1 Example configuration

The remainder of this section refers to an example configuration for the illustrative example system shown in Fig. 1. The corresponding configuration file is also shown below. This configuration file is included in the repository as `Configs/cortex.conf` and can be run from the `NeuroField` repository root using

```
./bin/neurofield -i Configs/cortex.conf -o cortex.output
```



From:	ϵ	ι	e	i
To ϵ:	0	0	0	0
To ι:	0	0	0	0
To e:	1	0	2	3
To i:	0	4	5	6

Figure 1: Top: schematic diagram of a purely cortical population model comprising excitatory and inhibitory populations, as well as two stimulus populations; each arrow indicates a connection between populations, so that each stimulus connects to a cortical population, and each cortical population connects to all cortical populations. Bottom: connection matrix indicating the connections between populations; zero indicates no connection, and a connection is indicated by a nonzero number, ordered top to bottom, left to right.

```
Example config file of cortical model with excitatory and
   inhibitory neurons
Time: 1 Deltat: 1e-4
Nodes: 400 Longside: 2

Connection matrix:
From: 1 2 3 4
To 1: 0 0 0 0
To 2: 0 0 0 0
To 3: 1 0 2 3
To 4: 0 4 5 6

Population 1: Stimulation
 Length: .5
 Stimulus: Const - Onset: 0 Mean: 5

Population 2: Stimulation
 Length: .5
 Stimulus: Const - Onset: 0 Mean: 5

Population 3: Excitatory neurons
 Length: .5
 Q: 8.87145
 Firing: Sigmoid - Theta: 13e-3 Sigma: 3.8e-3 Qmax: 340
  Dendrite 1: V: Steady alpha: 83 beta: 769
  Dendrite 2: V: Steady alpha: 83 beta: 769
  Dendrite 3: V: Steady alpha: 83 beta: 769

Population 4: Inhibitory neurons
 Length: .5
 Q: 8.87145
 Firing: Sigmoid - Theta: 13e-3 Sigma: 3.8e-3 Qmax: 340
  Dendrite 4: V: Steady alpha: 83 beta: 769
  Dendrite 5: V: Steady alpha: 83 beta: 769
  Dendrite 6: V: Steady alpha: 83 beta: 769

Propag 1: Map  - phi: Steady Tau: 0
Propag 2: Wave - phi: Steady Tau: 0 Range: 80e-3 gamma: 116
Propag 3: Map  - phi: Steady Tau: 0
Propag 4: Map  - phi: Steady Tau: 0
Propag 5: Wave - phi: Steady Tau: 0 Range: 80e-3 gamma: 116
Propag 6: Map  - phi: Steady Tau: 0

Couple 1: Map - nu: .15e-3
Couple 2: Map - nu: 1.5e-3
Couple 3: Map - nu:-1.8e-3
Couple 4: Map - nu: .15e-3
Couple 5: Map - nu: 1.5e-3
Couple 6: Map - nu:-1.8e-3

Output: Node: 1 2 Start: 0 Interval: 1e-4
Population: 4.V
Dendrite: 5
Propag: 1 4.phi
Couple: 3.nu
```

2.2 Global information

- *Initial comments*

```
Example config file of cortical model with excitatory and
   inhibitory neurons
```

Any text before the text `Time:` , is disregarded by `NeuroField` and serves as comment, which is strongly recommended for all configuration files.

- *Integration time and timestep*

```
Time: 10 Deltat: 1e-4
```

`Time` is the simulation duration in seconds.

`Deltat` is the time increment for each time step.

- *Grid size*

```
Nodes: 4 Longside: 2
```

`Nodes` is the number of grid points in the spatial dimension per population of neurons. The code has been explicitly designed to have equal number of neurons per population.

`Longside` is an optional parameter, specifying the longside of the rectangular grid. If it is not supplied, it is assumed to be a square.

Both spatial dimensions have periodic boundary conditions, so that populations have the topology of a torus.

Note that the physical size of each neural population is specified in the definition of the population - see Sec. 4.2.2. For each population, Δx is automatically computed based on the number of grid points and the physical size of the population. This ensures that the physical size of the system does not change when the number of grid points is changed. When a wave propagator is included, the value of Δx is automatically selected from the presynaptic population.

- *Specifying connectivity*

```
Connection matrix:
From: 1 2 3 4
To 1: 0 0 0 0
To 2: 0 0 0 0
To 3: 1 0 2 3
To 4: 0 4 5 6
```

We next specify a square connection matrix, where each entry is the connection from the column population to the row population. Zero indicates no connection, while a nonzero number indicates a connection. This number must be indexed from top to bottom, left to right.

The size of this matrix determines the number of neural populations in the simulation. The number of nonzero connections determines the number of dendrites, couplings and propagators that will be present in the configuration file.

2.3 Population data

This section contains population information sections. There are two types of neural populations: ordinary populations and stimulus populations:

Stimulus populations

`NeuroField` identifies stimulus populations as populations which have no dendrites, i.e., the row for that population contains no nonzero elements. Each stimulus population information section is as follows.

- ```
  Population 1: Stimulation
  ```

  The identifier `Population 1` is required for cross-checking.

  The descriptor `Stimulation` is not parsed by `NeuroField` , but it is strongly recommended for human referencing.

- ```
  Length: .5
  ```

  The physical (1D) length of the population (which is a 2D sheet), in mitres. This is used in `Wave` propagators and the `Psd` of `White` stimulus.

- ```
  Stimulus: Const - Onset: 0
  ```

  The identifier `Stimulus` is required for cross-checking.

  This is followed by the type of stimulus, to be further elaborated below.

  Optional parameter `Onset` specifies the time onset for the stimulus to begin. If unspecified, stimulus starts at time 0.

  Either `Cease` or `Duration` can be an optional parameter to specify the end time of the stimulus. If unspecified, stimulus ends at 1000 seconds.

  If optional parameter `Node` is specified, only the specified node indices will receive stimulation.

  Possible stimulus patterns:

  Constant

  ```
  Const - Mean: 5
  ```

  Pulse

  ```
  Pulse - Amplitude: 1 Width: 2e-2 Frequency: 1 Pulses: 1
  ```

  White

  Gaussian noise, characterized by the mean and standard deviation:
  ```

```
White - Mean: 1 Std: 20 Ranseed: 10
```

Alternatively, the power spectral density (PSD) may be specified instead of the standard deviation. The advantage is that the PSD is invariant to change in `Deltat` , population `Length` and spatial `Nodes` . Given the PSD, `NeuroField` correctly calculates the standard deviation:

```
White - Mean: 1 Psd: 20 Ranseed: 10
```

In general, it is preferable to specify the noise using PSD rather than Std. The magnitude of nonlinear effects and the total power in the spectrum both depend on the PSD rather than the Std.

The random number generator may be specified in `Ranseed` . If a seed is not specified, an automatically-incremented seed will be used instead, so that multiple stimulus populations will have independent sequences. In general it is not necessary to set the seed manually unless different random numbers are required for otherwise identical runs.

Superimposing stimuli

To superimpose 2 or more stimuli, begin with the keyword `Superimpose` , followed by the number of stimuli. Then list all the stimulus patterns and their parameters, with each stimulus pattern preceded by the keyword `Stimulus` .

```
Stimulus: Superimpose: 2
 Stimulus: White - Mean: 1 Psd: 1
 Stimulus: Pulse - Onset: 0.5 Width: 2e-2 Frequency 1
    Pulses: 1
```

Ordinary populations

Any non-stimulus population is an ordinary population.

- ```
  Population 3: Excitory neurons
  ```

  The identifier `Population 3` is required for cross-checking.

  The descriptor `Excitatory neurons` is not parsed by `NeuroField` , but it is strongly recommended for human referencing.

- ```
  Q: 8.87145
  ```

  The initial firing rate.

- ```
  Firing: Sigmoid - Theta: 13e-3 Sigma: 3.8e-3 Qmax: 340
  ```

  Specify the sigmoidal firing response of the population.

  `Sigma` is sometimes known as $\tilde{\sigma}$. It is already scaled by $\pi/\sqrt{3}$.

  Alternatively, you can specify a linear firing response by using

  ```
  Firing: Linear - Gradient: 1 Intercept: 1
  ```
  ```

- `Dendrite 1: V: Steady alpha: 83 beta: 769`

  The identifier `Dendrite 1` , where the number 1 is the presynaptic connection index, is required for cross-checking. Users should find that these indices are simply ordered as 1, 2, 3, 4, ...

  Optional parameter `V` may be used to specify the initial depolarization contribution from presynaptic activity. If unspecified, or set to `Steady` , `NeuroField` calculates the initial value by $V_{ab} = \nu_{ab}\phi_{ab}$.

  `alpha` and `beta` are the parameters for the depolarization response.

2.4 Propagation data

- `Propag 1:`

  This identifier is required for cross-checking.

- A propagator type is required at this point. Choices are `Map` , `Wave` , and `Harmonic` .

  Map

  `Map - phi: Steady Tau: 0`

  This propagator is the mapping propagator where spatial spreading is negligible. Its form is given by

  $$\phi_{ab}(\mathbf{r}, t) = Q_b(\mathbf{r}, t - \tau_{ab}).$$

  Optional parameter, `Tau` , is the axonal delay term. If unspecified, it is taken as zero. Since all propagators contain this object, its description is given below.

  Optional parameter `phi` may be used to specify the initial axonal firing rate. If unspecified, or set to `Steady` , `NeuroField` calculates the initial value by $\phi_{ab} = Q_b$.

  Wave

  This propagator is the wave equation propagator governed by the equation

  $$\left[\frac{1}{\gamma_{ab}^2}\frac{d^2}{dt^2} + \frac{2}{\gamma_{ab}}\frac{d}{dt} + 1 - r_{ab}^2\nabla^2\right]\phi_{ab}(\mathbf{r}, t) = Q_b(\mathbf{r}, t - \tau_{ab}).$$

  `NeuroField` checks whether the Courant condition must be satisfied, i.e.

  $$\Delta t/\Delta x < \sqrt{2}/r_e\gamma_e,$$

  where Δx is the population length per node.

  The propagator input is given by

  `Wave - phi: Steady Tau: 0 Range: 80e-3 gamma: 116`

Optional parameter `phi` may be used to specify the initial axonal firing rate. If unspecified, or set to `Steady`, `NeuroField` calculates the initial value by $\phi_{ab} = Q_b$.

`Range` is r_{ab} in the wave equation.

`gamma` is the damping coefficient. Alternatively, `velocity` may be specified, and `gamma` is calculated via $\gamma_{ab} = \mathrm{v}_{ab}/r_{ab}$.

In case there is only one node, this degenerates into a `Harmonic` propagator.

Harmonic This is a harmonic oscillator implementation of the damped wave equation, with no spatial variations:

$$\left[\frac{1}{\gamma_{ab}^2}\frac{d^2}{dt^2} + \frac{2}{\gamma_{ab}}\frac{d}{dt} + 1\right]\phi_{ab}(\mathbf{r},t) = Q_b(\mathbf{r}, t-\tau_{ab}).$$

The input form is given by

```
Harmonic - phi: Steady Tau: 0 gamma: 116
```

Tau

The axonal time delay between populations. If it is spatially homogeneous, then it is a number with units of seconds. If it is spatially inhomogeneous, then input n numbers, where $n =$ `Nodes`.

2.5 Coupling data

- ```
  Couple 1:
  ```

  Identifier for cross-checking.

- A couple type is required at this point. Choices are `Map`, `CaDP`, `BCM` and `Matrix`.

  Map

  Nonplastic synaptic coupling wtih a single constant parameter `nu`,

  ```
  Map - nu: 0.0012
  ```

  `nu` is the synaptic coupling parameter. It corresponds to the product of the mean synaptic strength s_{ab} and N_{ab}, the mean number of connections from cells of type b to cells of type a.

  Matrix

  Coupling becomes connection matrix, where connection strength does *not* change with time. The format of the nu matrix is the same as the population connection matrix, each row is to the same node, each column is from the same node. When outputting, each specified outputting node output the indexed row.

  ```
  nu:
    13e-6 0
    0 13e-6
  ```
  ```

CaDP

Calcium dependent plasticity according to Fung and Robinson.

```
CaDP - nu: 13e-6 nu_max: 80e-6 Dth: .25e-6 Pth: .45e-6
   xyth: 1e-4 x: 2.3e-2 y: 2e-2 B: 30e3 glu_0: 200e-6
   gNMDA: 2e-3
```

`Dth` and `Pth` are the calcium-plasticity thresholds; `xyth` , `x` and `y` are the plasticity rates; `B` , `glu_0` and `gNMDA` are NMDA receptor parameters.

To use `CaDP` , glutamate dynamics must be specified for the postsynaptic population. In the end of the relevant population entry, append

```
Glutamate dynamics - Lambda: 150e-6 tGlu: 30e-3
```

`Lambda` is the glutamate concentration rise per presynaptic spike; `tGlu` is the decay timescale for glutamate dynamics.

BCM

Extends `CaDP` with metaplasticity according to Fung and Robinson. it has an additional parameter `t_BCM` , the timescale of metaplasticity.

```
CaDP - nu: 13e-6 nu_max: 80e-6 Dth: .25e-6 Pth: .45e-6
   xyth: 1e-4 x: 2.3e-2 y: 2e-2 B: 30e3 glu_0: 200e-6
   gNMDA: 2e-3 t_BCM: 7
```

2.6 Output data

`NeuroField` outputs field quantities (i.e. a neurodynamic quantity which takes a value for each node) with respect to nodes and time. By default, the output file is `neurofield.output` , which can be changed by launching the program with the `-o` switch.

- ```
  Output:
  ```

  Begin with the `Output` declaration.

- ```
  Node: 1 2
  ```

  Enumerate all nodes to be outputted. If outputting all nodes, use shorthand `All` . If no nodes are specified, no nodes will be outputted.

- ```
  Start: 0 Interval: 1e-4
  ```

  Optional parameters for the time to start output, and optional parameter for time interval between outputs.

  If undefined, defaults, to 0 and `Deltat` , respectively.
  ```

- ```
  Population: 4.V
  Dendrite: 5
  Propag: 1 4.phi
  Couple: 3.nu
  ```

  `NeuroField` allows the user to specify which objects to output, by entering the appropriate object indices after the labels. For each object, it has some intrinsic fields that will be outputted; for example, `Couple` outputs `nu` , whereas `CaDP` outputs `nu` and `Ca` .

  For each entry, a field name may be appended after the index with a dot, so that only that field of the object is outputted. If no field name is specified for that entry, then all fields of that object is outputted.

  If a specific field of an object is specified, but that field does not exist, `NeuroField` checks and returns an error.

3 Analysis

`NeuroField` produces 3 files:

`neurofield.conf`	When using the launcher script, this file is created to store the running configuration file.
`neurofield.output`	The result of the simulation is stored here for postprocessing.
`neurofield.pbs`	If `NeuroField` is run in `yossarian` , then this file stores the output of the queueing system.

When the launcher script runs with only one set of parameters, all output files are also in the present working directory. However, if the launcher script sweeps over parameters, each parameter set has its own subdirectory inside `Output/` , and each set of `neurofield.*` files are stored in its subdirectory.

The output file starts with a copy of the input file, to enable the output file to serve as a complete representation of the simulation. The simulation results follow a series of `=` characters. Example content in `neurofield.output` is

```
================================================
                  Time              Propag.2.phi
                                                0
1.00000000000000e-04     8.87145000000000e+00
```

Each column is a time series with its name indicated in the first line. The first column is always time, and in this example, the second column is `Propag.2.phi` , indicating that it is ϕ_{ee} (when checked against connection matrix). The node number is indicated in the second line.
```

It is also worth noting that traces will be written in the order that they are specified. For example, if you write `Population: 3 1` then the columns in the output file will be arranged in this order.

3.1 Matlab

A number of `MatLab` functions are provided to make it easy to manipulate `NeuroField` data from within `MatLab` . The functions are generally self-documenting with comments at the start of the file.

Essentially, an output file from `NeuroField` is read into a `nf` struct object in `MatLab` which simply contains all of the output from `NeuroField` in memory for easy access. Here is an example of a `nf` object:

```
     fields: {'Propag.1.phi'  'Propag.3.phi'}
      nodes: {[1]  [1]}
       data: {[300000x1 double]  [300000x1 double]}
       time: [300000x1 double]
     deltat: 1.0000e-04
    npoints: 300000
```

- `fields` stores a record of which traces from `NeuroField` are present in the output file
- `nodes` is a cell with the same size as `fields` , and records for each field present, the number of the node in the output.
- `data` is a matrix storing the actual values of the traces
- `time` is a vector of time values, so that you can plot any of the data traces directly again `nf.time`
- `deltat` stores the temporal sampling rate
- `npoints` stores the total number of points in the output. The total duration is `nf.deltat*nf.npoints` (or `nf.time(end)`)

There are two ways to create the `nf` object. You can read the output file directly after executing `NeuroField` elsewhere

```
nf = nf_read('neurofield.output')
```

or you can use the `nf_run` helper script to run the config file using `NeuroField` and automatically parse the output

```
nf = nf_run('neurofield.conf')
```

Several helper files are provided to manipulate the `nf` object. The two most important helpers are `nf_extract` and `nf_grid`. Often you want to extract a particular field from the `nf` object, for example, to examine the output from `Propag.3.phi`. To do this directly with the `nf` object, you would need to check the `fields` variable to find the index of the trace you wanted, and then extract it from the `data` field. In the previous example, `Propag.3.phi` is the second trace. These expressions are identical:

```
trace = nf.data{2};
trace = nf_extract(nf,'Propag.3.phi')
```

`nf_extract` quickly becomes useful when there are many different fields. It is not case sensitive (so `propag.3.phi` works as well). You can also specify using additional arguments to extract only a portion of the time series, and also to select a subset of nodes. Finally, you can also provide multiple traces to concatenate them into a single matrix. For example,

```
trace = nf_extract(nf,'propag.1.phi,propag.3.phi');
```

will create a 300000×2 matrix with both of the traces.

Finally, if you run `NeuroField` with multiple nodes, it typically solves the system of equations on a square grid. Therefore, if you have output for notes 1-400, this corresponds to a 20×20 grid. `nf_grid` allows you to request a trace from the `nf` object and have it reshaped into a square grid. This lets you easily make surface plots of the data, or perform tasks that are spatially dependent.

One important task is computing the power spectrum as predicted by the linearized analytic equations. This can be achieved using `nf_spatial_spectrum` which takes in an `nf` object and computes the power spectrum integrated over k taking into account volume conduction.

4 Development

This section provides programming information about `NeuroField` for development extension and core logic. This section assumes a basic knowledge of `ANSI C++`, including object oriented programming, usage of template, and standard template library (STL). As our development takes place within a Git repository, a working knowledge of Git will significantly help.

A solid command of object-oriented programming is an essential prerequisite in developing `NeuroField`. Given the inevitable complexity arising from many developers and class relationships, good code structure and object-oriented programming principles MUST be adhered to whenever reasonable. One of the primary aims of object-oriented program as used by `NeuroField` is code reuse. Duplicated code, with subtle variations between them, is one of the best ways to introduce undetected numerical errors. The ultimate consequence is erroneous publication and refactoring of the code. Code reuse is one of the highest priorities. Sec. 4.1 contains methods for code reuse. When in doubt, read existing class implementations for

examples, or contact the developers for assistance (`braindynamics@physics.usyd.edu.au`).

Here is a list of training resources:

- Cpp resource
- Git resource
- OOP resource

4.0.1 Development workflow

We have adopted the *fork and pull* development model using GitHub. If you would like to contribute to `NeuroField`, set up your repository as follows:

1. Make a private fork of the main repository.
2. Clone your fork.
3. Set an upstream remote pointing at the main `NeuroField` repository.
4. Use `git pull upstream master` to merge changes from the main repository into the master branch of your working copy.
5. We recommend using a branch in your fork for developing modifications
6. When your feature is ready for integration into the main repository, submit a pull request on GitHub. Developers within the Brain Dynamics Group will then be able to review your code, check compatibility, and perform the merge.
7. Note that you do not have write access to the main repository, so all changes must be proposed via pull requests.

As discussed at the start of manual, `NeuroField` encapsulates each component of the model with an object: `Couple`, `Dendrite`, `QResponse`, `Propag`. The `Dendrite` and `QResponse` (and also `Timeseries`) are contained within `Population`.

Each object class may be overloaded for more sophisticated behaviour; for example, `Propag` may be overloaded to perform wave propagation, or `Couple` may be overloaded for synaptic plasticity.

4.0.2 Coding style

To maintain consistency within the `NeuroField` code base, we strongly encourage you to adhere to these programming conventions used throughout the project:

Tabs:	two spaces.
Braces:	the K&R style.
Class names:	UpperCamelCase
Function names:	lowerCamelCase
Variable names:	lowercase

When using the `C++` standard library, use the `using` directive to explicitly indicate the components you are using. For example,

```
#include<vector>
using std::vector;
```

4.1 Extending NeuroField via inheritance

Most new functionalities may be introduced by inheriting existing classes and overloading appropriate functions, where the core classes are:

Class	is Responsible for	Field
`Timeseries`	A function of time, predominantly used as stimulus.	
`Dendrite`	Dendritic response.	V_{ab}
`QResponse`	Firing response of population.	V_a
`Population`	Contains `Timeseries` , `Dendrite` , and `QResponse` .	Q_a
`Propag`	Axonal firing propagation	ϕ_{ab}
`Tau`	Axonal propagation time latency	τ_{ab}
`Couple`	Synaptic coupling	ν_{ab}

Examples where these classes are inherited to provided new functionalities include:

Derived class	Base class	Extension
`Wave`	`Propag`	Wave propagation.
`CaDP`	`Couple`	Plastic synapse.
`LongCouple`	`Couple`	Nonlocal synapse.

4.1.1 Class hierarchy

The diagram in Figure 4.1.1 shows the inheritance hierarchy for the base NeuroField objects. Note that the `Array` class is a container object, and it is not necessary to interact with it directly. See Sec. 4.6.1 for details.

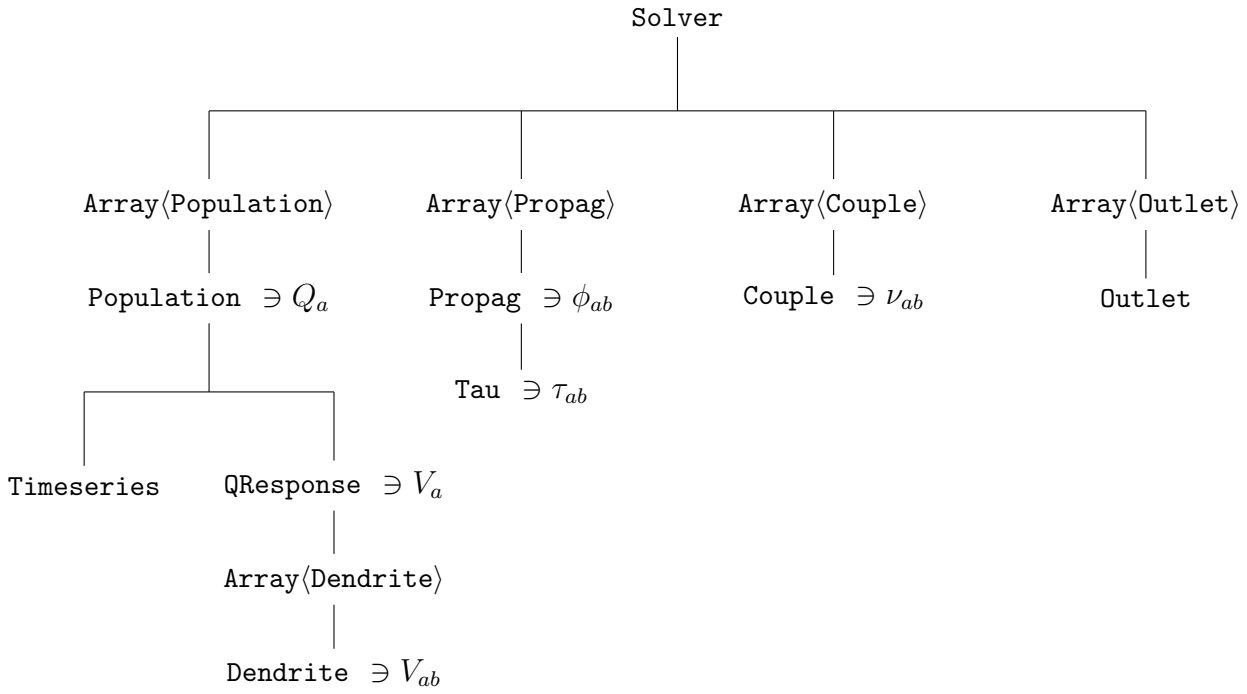


Figure 3: Schematic of the main class structures in `NeuroField`. Each line indicates that the bottom class is a member of the top class. The `a` $\ni b$ symbol indicates that the dynamical field $b(\mathbf{r}, t)$ is a member of the class `a`. Inheritance structures are NOT illustrated.

4.1.2 Procedure

Implementing a new class may be done using the following procedure:

1. Identifying the core class to inherit (see above tables). Then look up the documentation of the appropriate base class from Sec. 4.2.2–4.2.7.

2. Decide on the name of the class. Generally, it may be advantageous for the new name to refer to the base name, plus a terse description. For example, `LongCouple` refers to its base class `Couple`, with the additional description indicating that the new class has long range coupling. The file names should be the same as the class name, but all in lower case, for consistency.

3. Overload appropriately the `init()` (see Sec. 4.3), `output()` (see Sec. 4.4), and `step()` functions. **Do not** copy and paste code. Rather, use

   ```
   BaseClass::function();
   ```

   which will utilize the existing function in the base class. This will ensure that your derived class is updated if the base class changes.

4. It is likely that differential equations are solved in the new class. `NeuroField` provides

two classes, `DE` and `Stencil` , that solves spatially homogeneous differential equations, and spatially inhomogeneous equations, respectively. See Sec. 4.5.

5. Register the new class as documented in Sec. 4.2.2–4.2.7.

6. Write a configuration file that uses the new class. Or if the object may exhibit different types of behaviour under different parameter values, having one configuration file for each type of behaviour may be advantageous. Make sure that the configuration file has an appropriate comment.

4.2 NeuroField classes

This section contains an overview of all of the base classes in NeuroField.

4.2.1 Class NF

All core classes are derived from the `NF` object. This abstract base class contains 3 member variables, and 3 interface methods:

Variable	
`nodes`	The number of nodes as specified from the configuration file.
`deltat`	The time increment per timestep in units of seconds.
`index`	The index associated with the object.
Methods	
`init(Configf& configf)`	Initializes the object with the config file.
`step(void)`	At each timestep, this function is called.
`output(Output& output) const`	Specifies which fields to output.

All `NF` classes automatically handles the `ofstream::<<` and `ifstream::>>` operators.

When appropriate, the default constructor, copy constructor, and `operator=` should be made inaccessible by declaring them to be private.

4.2.2 Population

Models a neural population, which may be either a stimulus or normal population. If it has any `Dendrites` , i.e. it has presynaptic connections, then it is a normal population, and it is a stimulus if it does not have `Dendrites` .

In the former case, it contains the `QResponse` class and have a soma potential; in the latter case it contains the `Timeseries` class, and does not have a soma potential.

In either cases, the `Population` class has a keyring storing the firing rate history, coded as a 2D array plus an integer key.

Functions `sheetlength()` and `glu()` provides access to the physical length and glutamate concentration in synaptic cleft, respectively.

A population is "settled" after `Population::init()` is called, after which no dendrites can be added, and the firing rate history cannot grow.

4.2.3 Propag

The `Propag` class implements the axonal propagation as an identity map, i.e.

$$\mathcal{D}_{ab} = 1.$$

To introduce more sophisticated axonal propagation, this class is inherited and overloaded.

The `Propag` class provides a constant references to both presynaptic and postsynaptic populations.

Object `Tau` provides the axonal delay latency of propagator.

For spatially inhomogeneous propagators, class `Stencil` provides a Moore grid, as documented in Sec. 4.5.2.

To "register" your propagator, look for the

```
// PUT YOUR PROPAGATORS HERE
```

subsection in `solver.cpp` .

4.2.4 Couple

The `Couple` class manages ν_{ab}, which is constant in space and time.

To introduce synaptic plasticity, derive from this class.

The `Couple` class provides a constant references to both presynaptic and postsynaptic populations. Glutamate concentration is also provided.

Function `excite()` indicates whether this is an excitatory coupling. Protected variable `pos` is $+1$ or -1, depending on the sign of ν_{ab}.

To "register" your couple, look for the

```
// PUT YOUR COUPLES HERE
```

subsection in `solver.cpp` .

4.2.5 QResponse

To implement new firing response dynamics, inherit from class `QResponse` , where `init()` and `fire()` should be overloaded.

Object `dendrites` is an array of presynaptic `Dendrite` .

Objects `glu_m` and `glu_rk4` calculates glutamate concentration in the synaptic cleft, which is accessed via `glu()` .

To "register" your firing response, look for the

```
// PUT YOUR QRESPONSE HERE
```

subsection in `population.cpp` .

4.2.6 Stimulus

To implement new stimulus pattern, inherit from class `Timeseries` , where `init()` and `fire()` should be overloaded.

The time is provided via the variable `t` .

To "register" your stimulus, look for the

```
// PUT YOUR TIMEFUNCTION HERE
```

subsection in `timeseries.cpp` .

4.2.7 Dendrite

To implement new dendritic responses, inherit from class `Dendrite` .

References are provided for the presynaptic coupling and propagator.

To "register" your dendritic response, look for the

```
// PUT YOUR DENDRITE HERE
```

subsection in `qresponse.cpp` .

4.3 Adding variables to the configuration file

Your new class might require additional parameters in the configuration file. Input via the configuration file is implemented in the `init()` function, via `Configf` , which provides the following functions:

1. `param()` : go to the next keyword and reads in a variable. If the keyword is not found, barks and exits.
2. `optional()` : same as param, but does not bark nor exit. The use of this function is discouraged, since it may introduce subtle bugs or human errors.
3. `numbers()` : reads an arbitrary number of white-space separated numbers, returned in a `vector` .

Sometimes the config file may search for either one parameter, OR another one. For example, `Wave` propagator accepts parameter `gamma` or `velocity` such that `gamma` = `velocity` / `range` , so that we may have either of these:

```
Wave - Range: 80e-3 gamma: 116
```

```
Wave - Range: 80e-3 velocity: 9.28
```

Then, we may use this pattern with `optional()` to achieve the desired effect:

```
configf.param("Range",range);
if( !configf.optional("gamma",gamma) ) {
  double temp; configf.param("velocity",temp);
  gamma = temp/range;
}
```

4.4 Adding variables to the output file

`NeuroField` outputs field variables at every timestep, where the user chooses which object to output via the configuration file, and the object chooses which fields to output in the `output()` function.

To output field solutions, overload `NF::output()` to write

```
output.prefix("Object Name",index+1);
output("field1",field1);
output("field2",field2);
subobject1.output(output);
subobject2.output(output);
BaseClass::output(output);
```

or for a single output field,

```
output("Object Name",index+1,"field1",field1);
```

where `field1` , and `field2` are `vector<double>` with size equal to the number of spatial nodes.

To output a single number, as opposed to a spatial field, use the function

```
output.singleNode("Object Name",index+1,"field1",field1);
```

or

```
output.singleNode("field1",field1);
```

where `field1` is a `vector<double>` of size 1.

4.5 Tools for solving differential equations

Classes `DE` and `Integrator` (currently RK4 is implemented) are used to solve generic systems of ODEs, where the dynamical variables are homogeneous fields. For inhomogeneous DEs and spatial dependency, a 9-point stencil is provided in `Stencil` .

4.5.1 Class DE

Class `DE` and `RK4` together solves ODEs of homogeneous fields.

To define your differential equation, declare a new class that inherits `DE` . Overload the `rhs()` function to define the differential equation. For example, the differential equation

$$F = m\frac{d^2x}{dt^2},$$

is redefined as

$$F = y_0,\ x = y_1,\ \frac{dx}{dt} = y_2,$$

so that it can be formulated as a system of 1st order differential equations

$$\frac{dy_0}{dt} = 0,\ \frac{dy_1}{dt} = y_2,\ \frac{dy_2}{dt} = my_0,$$

with

$$y_0 = F,$$

being an algebraic equation.

This can be defined in `NewDE::rhs()` as

```
void NewDE::rhs( const vector<double>& y, vector<double>& dydt )
{
  // y = { F, x, dxdt }
  dydt[0] = 0;      // F, leave unchanged
  dydt[1] = y[2];   // x
  dydt[2] = m*y_0; // dxdt
}
```

where the comments are strongly recommended for readability.

Declare the number of differential equations (in this example, 3) in the constructor

```
NewDE( int nodes, double deltat ) : DE(nodes,deltat,3) {}
```

To integrate `NewDE` , declare a pair of `DE` and `RK4` objects:

```
NewDE de(nodes,deltat);
RK4 rk4(de);
```

then the differential equation may be solved via (usually done in `Class::step()`)

```
for( int i=0; i<nodes; i++ )
  de[0][i] = F; // algebraic equation
rk4.step();     // integrate differential equation by one step
```

Often, the `NewDE` class is incorporated in a `NewClass` . In such cases, it is advantageous to declare `NewDE` as `struct` , which is a class where all members are public by default, and have this new `NewDE` declared within `NewClass` , so that it is accessible within it. All parameters of the differential equation (in the example above, m) will then belong to `NewDE` .

Since `NewClass` has access to all members of `NewDE` , all initialization of `NewDE` may be done in `NewClass::NewClass()` and `NewClass::init()` .

Remember to redirect the interface and output variables to members of `NewDE` . For example,

```
vector<double>& NewClass::x(void) const
{
```

```
  // original code from base class:
  // return _x;
  // new code in derived class replaces original x variable:
  return (*de)[1];
}
```

```
void NewClass::output( Output& output ) const
{
  output.prefix("NewClass",index+1);
  // original code from base class:
  // output("x",_x);
  // new code in derived class replaces original x variable:
  output( "x",(*de)[1] );
}
```

To extend an existing `DE` class, for example extend `NewDE` to `New2DE`, inherit `New2DE` from `NewDE`. The `extend()` function allows the introduction of new differential equations and field variables.

4.5.2 Stencil

Class `Stencil` provides Moore grid for spatially inhomogeneous calculations.

Given a stencil,

```
Stencil stencil(nodes,longside,"Torus");
```

Use `operator=` to set the spatial field values of a `vector<double>`.

The stencil pointer can be set and get via the `set()` and `get()` functions, and incremented via `operator++`. The Moore grid can be read with

```
stencil(nw); stencil(n); stencil(ne);
stencil( w); stencil(c); stencil( e);
stencil(sw); stencil(s); stencil(se);
```

4.6 Core logic

Normal extension of `NeuroField` *does not involve any modification of the core logic. Proposed to the core logic are unlikely to be accepted. This section is intended to provide context to understand the design of the base objects.*

One integration step of the model implements the following stages:

1. Dendritic response
2. Afferent summation.
3. Firing response/stimulus response.
4. Wave equation integration step which includes Q delay processing
5. Coupling response.

Most of the computational *load* comes from integrating wave equations and harmonic oscillators within the dendritic responses. Most of the execution *time* is probably spent writing the output file.

Wave equations are integrated by explicit finite differences integration. A nine point spatial stencil is used to reduce high frequency spatial instabilities when driven by random noise. Other parts of code are unaffected by spatial geometry so this can be switched to irregular gridding easily.

Harmonic oscillators with dendritic rseponse are integrated using a heavily strength reduced explicit direct integration assuming constant drive. This was more efficient than a constant drive RK4 algorithm which would not be fourth order in any case due to the constant drive. Rennie used a constant drive RK4 for his 1997 code.

4.6.1 Class Array

`Array` is a container array to store objects that supports the `ofstream::<<` and `ifstream::>>` operators, as well as a `step(void)` function. This object typically is, but is not required to be, an `NF` object.

The `step(void)` function is equivalent to a `foreach(element).step()` in pseudocode. This function is encouraged over the use of `empty()` , `size()` , and `operator[]()` , which are discouraged to be used.

4.6.2 Program flow

Essentially, the program flow can be read from Fig. 4.1.1, so that objects take priority from top to bottom, left to right, both in terms of initialization and stepping through each timestep. A more detailed description is given below, and the reader is referred to the source code for complete description.

We use the semicolon to denote a succession of functions/procedures, and `a()` $\Rightarrow$ `b()` symbol to denote function `b()` as content of function `a()` .

`main()`	⇒	Initialize the config file, dump file and output file;
		`Solver::init()` ; `Solver::solve()` ;
`Solver::init()`	⇒	read in global parameters; Read in `CntMat` ;
		Construct `Population` ; construct `Propag` ; construct `Couple` ; `Population::add2Dendrite()` ;
		Read configurations for `Population` , `Propag` , `Couple` , and `Output` .
`Solver::solve()`	⇒	`for(...)` { `Solver::step()`; `Output::step()` ; }
`Solver::step()`	⇒	`Population::step()` ; `Propag::step()` ; `Couple::step()` ;
`Population::step()`	⇒	`QResponse::step()` if neural population;
		`Timeseries::step()` if stimulus
`QResponse::step()`	⇒	`Dendrite::step()` ; sum over V_{ab}

4.7 Output routine

To accommodate the coding interface for `NF::output` , the output routine of `NeuroField` involves 4 separate classes: `Outlet` , `Output` , `Outputs` and `Dumpf` .

Class	Role
`Outlet`	Stores a reference to field variable (`vector<double>`) and its associated name.
`Output`	Helper class in the parsing of which objects and which fields to output, according to the configuration file.
`Outputs`	Contains `Array<Outlet>` and performs output routine.
`Dumpf`	File handle (maybe to `stdout`) to output.

4.8 About object-oriented programming

Working knowledge of object-oriented programming (and `C++`) is an essential prerequisite for developing `NeuroField` . While object-oriented programming is standard computing knowledge and material and references on the topic is abundant, here we outline the motivation behind object-oriented programming, and in particular the reason for using it on `NeuroField` .

4.8.1 Procedural programming

1. *Procedural programming* consists of *variables* and *functions*. Variables (and structs) are the "nouns"; functions are the "verbs."

2. In procedural programming, there are no inherent *code structure* in the code that is enforced by the language. Rather, code structure is given by, for example, file systems.

4.8.2 Object-oriented programming

1. *Object oriented programming* generalizes variables and structs to *objects*, where an object is a collection of variables and functions.

2. While an object is a noun, it is capable of performing actions and having other objects perform actions on them.

3. The idea of an object performing actions and being the recipient of actions lead to the idea of the *interface* of an object.

4. This separation between implementation (the actual algorithmic code) and interface gives rise to encapsulation, inheritance and polymorphism:

   Encapsulation is the *hiding of the implementation* that is not part of the immediate interface.

   Inheritance is where a class of objects *reuses* and *extends* the implementation and/or interface of a more fundamental class.

   Polymorphism is where objects of the same interface perform their own appropriate actions. This gives rise to dynamic object types and behaviour.

5. To summarize, some features brought forth are code structure and protection, code reuse and extension, dynamic allocation of object type, and (if applicable) coding-level object hierarchy.

4.8.3 NeuroField

1. `NeuroField` is naturally object oriented: firing response, propagators, couplings, dendrites naturally arise and form a network, where each class/object influences other ones.

2. The extension of simple objects to more sophisticated ones (e.g. from static to plastic synaptic coupling) may naturally be done via inheritance. In many cases, the interface is kept while the implementation is extended.

3. The object type is determined during *runtime*, while reading the config file. Thus, *polymorphism* comes into play.

4. All fundamental classes in `NeuroField` has already been inherited. Among the inherited classes, many have fundamental relationships with classes in the inheritance hierarchy, e.g. `Wave` propagator may degenerate to `Harmonic` , `BCM` couple extends `CaDP` .

5. Given these complexities, which will only increase in the future, object-oriented programming principles MUST be adhered to whenever reasonable. Failure to do so will inevitably lead to unmaintainable (and unacceptable) code that may well introduce undetected numerical error.