NeuroField Developer Guide

Felix Fung

October 30, 2014

This guide documents `NeuroField` for development extension and core logic. The readers are assumed basic knowledge of `ANSI C++`, including object oriented programming, usage of template, and standard template library (STL). For introduction and usage of `NeuroField`, refer to the separate user manual.

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 NeuroField coding principles

`NeuroField` solves each equation within the Robinson et al. model with an object:

$$\begin{aligned} P &= \nu_{ab}\phi_{ab}, && \texttt{Couple} \\ D_{ab}V_{ab} &= P, && \texttt{Dendrite} \\ Q_a &= S_a\Big[\sum_b V_{ab}\Big], && \texttt{QResponse} \\ \mathcal{D}_{ab}\phi_{ab} &= Q_b, && \texttt{Propag} \end{aligned}$$

where `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.

1.1 Object-oriented programming and code reuse

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.

NEVER copy and paste code. EVER.

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. 2 contains methods for code reuse.

When in doubt, read existing class implementations for examples, or contact Felix Fung (`f.fung@physics.usyd.edu.au`).

1.2 Coding style

Within the C++ source files, developers are strongly encouraged to stick to this conventions for consistency:

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

Do not include unused header files, for readability reasons. When using the `C++` standard library use the `using` directive to indicate the reason of the inclusion. For example,

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

1.3 Submission to SVN

To make changes to the program, copy the trunk into a new subfolder within branches

```
cp -r trunk ./branches/my_branch
```

You can now modify the code in the branch, where Sec. 2 provides good documentation and methods.

To submit your changes, first perform make sure your code is numerically correct, by running in `MatLab`

```
run Test/test_eirs_spectrum.m
```

And see that the new spectrum is in agreement with the existing ones.

When you are ready, check the new code into `SVN` with

```
svn ci
```

and provide a descriptive log.

Others can now obtain your code through the branch. If your modifications are accepted for the main program, your branch will then be merged into the trunk subject for appropriate testing.

2 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.

2.1 Procedure

The writing of a new class may be done by following this procedure:

1. Identifying the core class to inherit (see above tables). Then look up the documentation of the appropriate base class from Sec. 2.3–2.8.
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. 2.9), `output()` (see Sec. 2.10), and `step()` functions. NEVER copy and paste code. Rather, use

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

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. 3.
5. Register the new class as documented in Sec. 2.3–2.8.

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.

7. After debugging, check the code in. See Sec. 1.3.

2.2 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.

2.3 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.

2.4 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. 3.2.

To "register" your propagator, look for the

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

subsection in `solver.cpp` .

2.5 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` .

2.6 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` .

2.7 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` .

2.8 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` .

2.9 Reading from 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;
}
```

2.10 Output

`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.

3 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` .

3.1 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.

3.2 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 Core logic

Normal extension of `NeuroField` does NOT involve any modification of the core logic. Do NOT touch the core logic unless you are certain what you are doing. And when you do, contact Felix Fung (`f.fung@physics.usyd.edu.au`).

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 computational time is probably on outputting.

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.

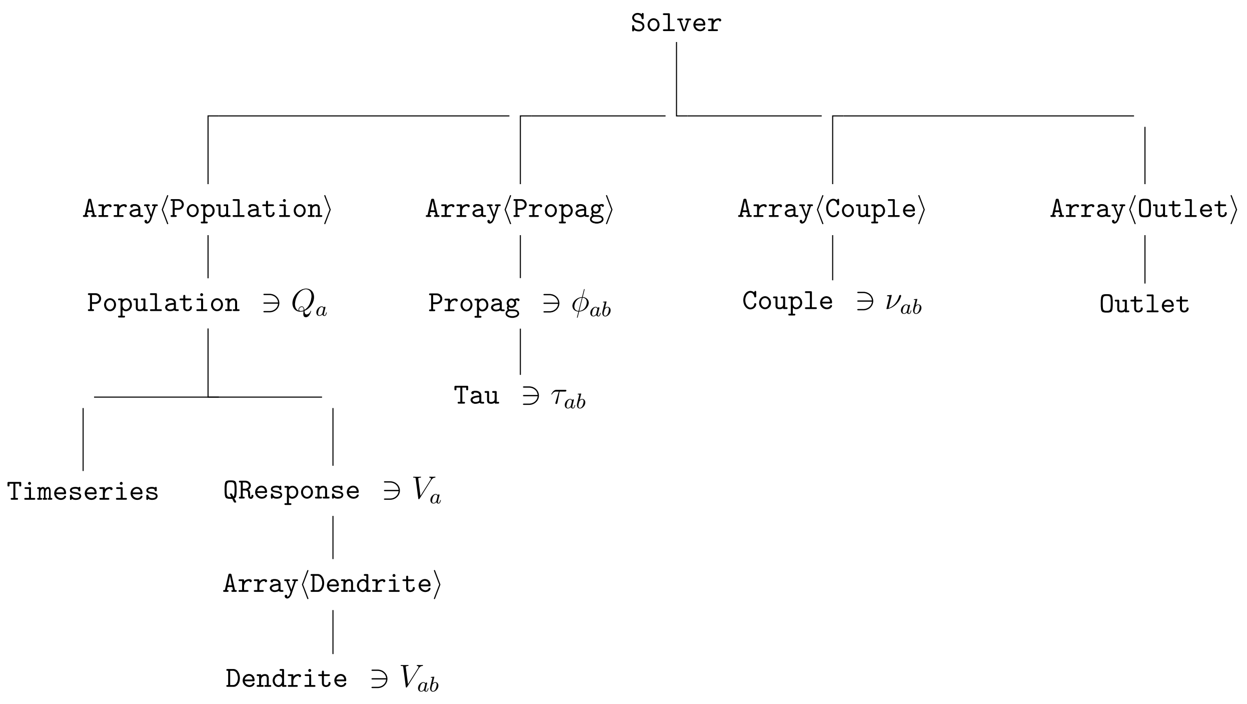

Figure 1: Schematic of the main class structures in `NeuroField` . Each line indicates that the bottom class is a member of the top class. The `a` $\ni b$ symbol indicates that the dynamical field $b(\mathbf{r}, t)$ is a member of the class `a` . Inheritance structures are NOT illustrated.

4.1 Class diagram

4.2 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 not necessarily has 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.

4.3 Program flow

Essentially, the program flow can be read from Fig. 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()`	$\Rightarrow$	Initialize the config file, dump file and output file;
		`Solver::init()` ; `Solver::solve()` ;
`Solver::init()`	$\Rightarrow$	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()`	$\Rightarrow$	`for(...)` { `Solver::step();` `Output::step()` ; }
`Solver::step()`	$\Rightarrow$	`Population::step()` ; `Propag::step()` ; `Couple::step()` ;
`Population::step()`	$\Rightarrow$	`QResponse::step()` if neural population;
		`Timeseries::step()` if stimulus
`QResponse::step()`	$\Rightarrow$	`Dendrite::step()` ; sum over V_{ab}

4.4 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.

5 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` .

5.1 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.

5.2 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.

5.3 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.

6 TODO

1. check numerical entry of connection matrix