Neural Activity Regression Framework (NARF) Documentation

Ivar Thorson

June 27, 2013

Contents

Chapter 1

Basic Functionality and Data Structures

This document was written to clarify and document design decisions made regarding the Neural Activity Regression Framework (NARF) implemented in the Brain, Hearing and Behavior Laboratory of the Oregon Health and Science University (OHSU), under the supervision of Prof. Stephen David. As such, it is intended for developers rather and is very much a work in progress.

1.1 Problem and Approach

1.1.1 The Problem

There are many possible mathematical models which could be used describe the spiking activity of a neuron given some sensory stimulus. In our case, we will be considering mostly models of neurons in the auditory cortex of a mammal, usually a ferret, and usually near the A1 region. Neurons in this region of the brain often respond preferentially to a particular auditory frequency. For example, some neurons may spike continously whenever any sound has accoustic energy near 1.2KHz. Another neuron may spike frequently only on the onset of a 5KHz tone hidden in a ferret vocalization played to the ferret's left ear while it is thinking about dinner. Our goal is to play recordings of various sounds, measure neural activity, and attempt to discover as much as we can about the neuron(s) we are studying.

Mathematically, the problem is that of developing a mathematical model which can accurately predict how a neuron will fire when a new, as-yet-unheard sound is played. This is a classic inference problem, in which we seek to infer which system produced the neural response that we measured. If the model's prediction closely matches the measured response of the neuron, then we have learned something significant about the neuron. The better our ability to predict the neuron's activity, the better our understanding of it, and the more we may learn through experiment about how neural activity is affected by various factors. Our ability to learn about the dynamics of neurons through experiment is thus closely related to our ability to model each neuron.

Fortunately or unfortunately, while the neuroscientist community has developed a plethora of mathematical models which can describe this behavior, there remain few guidelines on which model should be used or how model parameters should be found. The difficulty for most scientists is in the implementation of existing models; it is often a significant amount of work to adapt another person's model to your needs, and significantly more work to develop one yourself. It's hard to compare your work to other scientists'. Ideally, we would be able to easily fitting several different models to the measured activity of the neuron, so that we might quickly identify the characteristics of the neuron under study, as well as give us the ability to compare our work with other researchers'. It would also be advantageous to be able to recombine parts of the model which were particularly useful in describing neural activity such that variations on successfully predictive models could be quickly examined.

1.1.2 NARF's Approach: Pure Function Composition

Most modeling approaches are highly domain-specific; modeling the a biological system often requires strong assumptions about the nature of the signals being processed and correlated. What works well for modeling the auditory cortex may not work well for modeling activity in the motor cortex. About the only thing we can be sure of is that we will be using mathematical functions, so we begin there. If we consider each aspect of a mathematical model to be a **pure function** f_1 which accepts an argument x_1 and returns a value x_2, we can write it as

$$f_1(x_1) \rightarrow x_2 \tag{1.1}$$

The notion of a 'pure' function is simply that it will always produce the same result when given the same input. This is familiar to most mathematicians, but the idea of a function is usually corrupted in the domain of computer science to mean just a "subroutine" of instructions. To be more specific, a pure function is on in which there are no side effects; no print statements, no changing of global variables, no writing to disk, and no use of global variables. A pure function is a complete black box whose outputs only depend on its explicit arguments and inputs. If a function's computation is based in any way on a global variable, then it is possible for the same function, given the same arguments, to produce different results depending on the value of the global variable at the time it was run. This should be avoided to maintain mathematical elegance. Indeed, it is often an explicit goal of a seasoned functional programmer to keep the number of non-pure functions to a minumum; but we digress and must return to the operation of NARF.

If we make a neural model out of more than one pure function (e.g. $f_1, f_2, ..., f_n$), then we could potentially chain the output of one function to the input of the next:

$$\begin{aligned} f_1(x_1) &\rightarrow x_2 \\ f_2(x_2) &\rightarrow x_3 \\ &\vdots \\ f_n(x_n) &\rightarrow x_{n+1} \end{aligned}$$

And we will always get the same output (x_{n+1}) at the end of the computational chain. In a perhaps more familiar notation we could have written

$$x_4 = f_3(f_2(f_1(x_1))) \tag{1.2}$$

All of the above has assumed that every function has no parameters. If we had a function which added 23 to it's input argument and then divided by three ($f_1(x_1) = (x_1 + 23)/3$), then we would be required to define a completely new function f_1 if we wanted to try adding 25 or dividing by a different number. If some part of the function's behavior will be changing regularly and many values must be tried, then it makes sense to bundle these changing parts into a model parameter vector $\phi_1 = [\phi_{11}\phi_{12}] = 233$ and define our function as $f_1(x_1, \phi_1) = (x_1 + \phi_{11})/\phi_{12}$.

Therefore, NARF assumes that every function has two arguments

$$f_1(x_1, \phi_1) \rightarrow x_2 \tag{1.3}$$

and that they can be composed together to create chains of pure functions.

$$\begin{aligned} f_1(x_1, \phi_1) &\rightarrow x_2 \\ f_2(x_2, \phi_2) &\rightarrow x_3 \\ &\vdots \\ f_n(x_n, \phi_n) &\rightarrow x_{n+1} \end{aligned}$$

Now we can describe nearly any neural model by specifying some initial conditions x_1, a list of functions $f_1, f_2, ..., f_n$, and a list of parameter vectors $\phi_1, \phi_2, ..., \phi_n$. By executing the functions one after another, our prediction about the neural activity will finally be held in x_{n+1}. If we want, we can observe intermediate values of x to better understand how the model is functioning as a whole.

1.2 Module Technical Details

MATLAB's data types prohibit us from being completely general with regard to how we represent our functions f and our data x. Data structures are never perfectly general, but some are more flexible than others. The approach used in NARF was to use MATLAB's cell arrays as much as possible because of their generality and loose ordering of data along different dimensions.

The structure which holds the list of functions $f_1, f_2, ...$ and parameters ϕ called `STACK` because it represents a big fat stack of transformations on the input data. The structure which holds all the $x_1, x_2, ..., x_{n+1}$ is called `XXX` due to the fact that each x is dependent upon the previous and so it resembles a chain of x's. Each cell of `STACK` and `XXX` is a structure containing named fields. You may put any fields you like in each cell, except for the fields defined in the next two sections, which have special meaning to NARF and are used for special purposes.

1.2.1 Terminology

`STACK` was described as a chain of modules, but we should first define what exactly a module is. Because so often it is convenient to keep a function f and its associated parameters ϕ together, we will call them a **module**, for lack of a better way to describe each block of modeling functionality. Each module's subfields are called **parameters** because they represent named values of ϕ.

The data values $x_1, x_2, ...$ are represented as structs, and we name the fields of each struct a **signals**, because typically they are long vectors of neural or electrical signals that were sampled at a regular rate.

1.2.2 Module Discovery and the `MODULES` Global Data Structure

Whenever a NARF GUI starts running, it searchers for modules that in the `modules/` directory upon startup and attempts to load them into a structure called `MODULES`. MATLAB is pretty good about allowing you to edit modules dynamically, but if you are adding new modules after starting a GUI, you may need to call the function `scan_directory_for_modules.m` with the appropriate to re-scan them.

Once discovered, the global data structure `MODULES` is updated so that under each field is one of the objects returned by the modules. You could manually create such a global data structure like this:

```
>> global MODULES;
>> MODULES = scan_directory_for_modules()

MODULES =

              add_nth_order_terms: [1x1 struct]
              bayesian_likelihood: [1x1 struct]
             concatenate_channels: [1x1 struct]
                      correlation: [1x1 struct]
           depression_filter_bank: [1x1 struct]
                downsample_signal: [1x1 struct]
    elliptic_bandpass_filter_bank: [1x1 struct]
                       error_norm: [1x1 struct]
                       fir_filter: [1x1 struct]
            gammatone_filter_bank: [1x1 struct]
                 gmm_nonlinearity: [1x1 struct]
                    infer_respavg: [1x1 struct]
                  int_fire_neuron: [1x1 struct]
            inter_spike_intervals: [1x1 struct]
       load_stim_resps_from_baphy: [1x1 struct]
             load_stim_resps_wehr: [1x1 struct]
               mean_squared_error: [1x1 struct]
                     nonlinearity: [1x1 struct]
     nonparm_filter_nonlinearity: [1x1 struct]
             nonparm_nonlinearity: [1x1 struct]
          nonparm_nonlinearity_2d: [1x1 struct]
               normalize_channels: [1x1 struct]
                         passthru: [1x1 struct]
                   smooth_respavg: [1x1 struct]
    sparse_empirical_nonlinearity: [1x1 struct]
                       sum_fields: [1x1 struct]
              sum_vector_elements: [1x1 struct]
```

As you can see, there are already quite a few modules available for neural analysis.

1.2.3 A Minimal Module

Let's take a look at how to make a minimalist module that adds 11 to every number it gets as an input.

```
function m = my_module(args)
% A minimal NARF module.
% Returns a function module 'm' which implements the MODULE interface.

% Module fields that must ALWAYS be defined
m = [];
m.mdl = @my_module;
m.name = 'my_module';
m.fn = @do_my_module;
m.pretty_name = 'My Adder Module';
m.editable_fields = {'input', 'time', 'output', 'add_parameter'};
m.isready_pred = @isready_always;

% Module fields that are specific to THIS MODULE
m.input  = 'stim';
m.time   = 'stim_time';
m.output = 'prediction';
m.add_parameter = 11;

% Optional things which may or may not be present
m.is_perf_metric = false;
m.fit_fields = {'add_amount'};

% Overwrite the default module fields with arguments
if nargin > 0
    m = merge_structs(m, args);
end

% Optional plotting fields
m.plot_fns = {};
m.plot_fns{1}.fn = @do_plot_inputs;
m.plot_fns{1}.pretty_name = 'Plot Output vs Time';

% Finally, define any functions specific to this module
function x = do_my_module(mdl, x, stack, xxx)
    % Add some amount to every single
    x.(mdl.output) = x.(mdl.input) + mdl.add_parameter;
end

function do_plot_inputs_and_pnorm(sel, stack, xxx)
    [mdls, xins, xouts] = calc_paramsets(stack, xxx(1:end-1));
    do_plot(xouts, mdls{1}.time, {mdls{1}.input1, mdls{1}.input2}, ...
            sel, 'Time [s]', 'Output [-]');
end

end
```

It should be apparent to most programmers that a module is simply a function that returns a struct named `m` that has some field/value pairs. `m` is a default object which will be created even when `my_module` is not given any arguments. What may be less obvious, confusing, or even mind-blowing is that the `args` variable is also intended to be a struct and that `m.mdl` stores a reference to the `my_module` function itself. This means that a NARF module `m` can create a mutated copy of itself by passing a struct to the function stored under `m.mdl`. Consider the following example, which assumes that you have already scanned for modules and placed them in the global `MODULES` structure.

```
>> global MODULES;
>> m1 = MODULES.my_module;
>> m1.input

ans =

stim

>> m2 = m1.mdl(struct('input', 'foo'));
>> m2.input

ans =

foo

>> m1

m1 =

                mdl: @my_module
               name: 'my_module'
                 fn: @my_module/do_my_module
        pretty_name: 'My Adder Module'
    editable_fields: {'input'  'time'  'output'  'add_parameter'}
       isready_pred: @isready_always
              input: 'stim'
               time: 'stim_time'
             output: 'prediction'
      add_parameter: 11
     is_perf_metric: 0
         fit_fields: {'add_amount'}
           plot_fns: {[1x1 struct]}

>> m2

m2 =

                mdl: @my_module
               name: 'my_module'
                 fn: @my_module/do_my_module
        pretty_name: 'My Adder Module'
    editable_fields: {'input'  'time'  'output'  'add_parameter'}
       isready_pred: @isready_always
              input: 'foo'
               time: 'stim_time'
             output: 'prediction'
      add_parameter: 11
     is_perf_metric: 0
         fit_fields: {'add_amount'}
           plot_fns: {[1x1 struct]}
```

As you can see, both module instances `m1` and `m2` are identical except for the `input` field. This ability to quickly copy, modify, and perturb module instances is a useful feature to have. If desired, modules can be written to ensure that changing one parameter will necessarily update other, related parameters. For example, if `add_parameter` were not a scalar but a vector, you might want it to modify another parameter called `average` which is based upon the average of that vector.

1.2.4 Required Module Fields

Modules are just structs; what makes them more than just a struct is that they implement the module **interface**. Every module *must* have these required fields which are used by NARF to interface with the functionality of the module:

`m.pretty_name` An alphanumeric string describing the function that is displayed to the user of the NARF GUIs.

`m.name` The name of the function. It should be the same as the module function file name.

`m.mdl` A function handle of the module. This will be the same as `name`, but with an `@` symbol prefixed to get the function handle.

`m.fn` A handle to a function which does the core computation f associated with the module. Function f should accept four arguments `(mdl, x, stack, xxx)`, where `mdl` is the module instance struct itself and `x` is the data input x that it should work on. Although it is rare to use them in practice, for generality a copy of the complete `STACK` variable and the `XXX` variable are also provided to the function so that a function f_n can look at parameters or data of modules that preceeded it (e.g. ϕ_{n-2}, x_{n-5}).

`m.editable_fields` A cell array of strings that list which parameters should be editable in the GUI. Sometimes you don't want to expose all of the module parameters to naive users, and so by default no parameters are gui-editable unless listed in this cell array. Any fields listed in `m.editable_fields` must have values with one of the following types: integer, floating point, 1D or 2D matrix, string, boolean, function, or a cell arary of strings. No other data types are allowed. If you need to add one, please make sure it is compatible with `write_readably.m`.

`m.isready_pred` The "module-is-ready-to-be-appended predicate"; in other words, a function that returns true only when the module has its data dependencies and connectivity met. Not all modules can be appended to an existing `STACK`; sometimes the signals are not ready or the input dimensionality is not correct for the module being used. In current practice, this predicate is not used much because we are mostly just building models whose modules all interact harmoniously, but leaving this hook intact is a good idea because perhaps in the future we will need to auto-generate models from modules that are not always compatible.

Modules may also have these optional fields which enable special behavior. These aren't fully documented yet; this is just an overview.

`m.plot_fns` A cell array of functions two fields: plot names, and functions with three arguments each. The first element in each sub-cell-array is a user-readable string naming the plot function, the second is a function handle to a function which plots on the default plot axes, and the third is a struct which contains parameters that will be passed to the plot function.

`m.plot_gui_create_fn` If defined, during GUI creation this function is called and used to create GUI widgets in the designated widget space for each module. If desired, you can then make the module's plots react to changes in the gui widgets.

`m.auto_plot` If this is defined, when the model has been successfully fit and it is time to generate a summary image of the model, any function placed here will be called.

`m.auto_init` A function that is called which can be used to initialize a module's parameters based on the run-time values of the `XXX` struct. This is used sparingly but usefully in practice, since it can allow modules to adapt themselves to the particular model structure already defined. It is intended to be called by `append_module.m`.

`m.splitter` Used to create parameter sets that act on different parts of the same data.

`m.unifier` Used to join the results of multiple parameter sets that acted on different parts of the same data.

`m.is_perf_metric` When true, this signals to any fitting algorithms that this module contains a performance metric that should be used to fit.

1.2.5 `STACK` Global Data Structure

The `STACK` data structure is a cell array of cell arrays. The reason for this nesting will be described further in section **??**, but for now we need only concern ourselves with the first level of cell arrays, which will be as long as there are modules in the stack.

A cell array containing the functions and their parameters that were applied to reach this point.

1.2.6 `META` Global Data Structure

A model is more than just a collection of modules; certain pieces of information don't really work as a parameter of any particular module, yet need to be carried around with the model. The `META` struct is used for this purpose, as it contains all the **metadata** of the model, for example:

- the name of the model
- the filename of the saved model
- the path where the saved model file is located
- the saved images are associated with the model
- when its parameters were fit
- how long it took to fit the model, in seconds
- which performance metric is used to fit the model
- the git hash referring to the version of NARF used to fit the model

It might look something like this:

```
>> global META
>> META

META =

     git_commit: '7c42b9ee3cee2c5f1065ea76157d59179479e139'
          batch: 241
      modelname: 'env100_log2b_firno_initrc_nonl_mse_boost'
      modelfile: '241_por019b-a1_env100_log2b_firno_initrc_nonl_mse_boost.mat'
      modelpath: [1x103 char]
    perf_metric: @pm_mse
       fit_time: 217.1380
```

1.2.7 `NARFGUI` Global Data Structure

This isn't really documented yet, but it contains handles to all of the GUI widgets created for each module. It is mostly used by `narf_modelpane`.m.

1.2.8 `XXX` Global Data Structure

A cell array, with the most recent data being first. The contents of each cell could technically be any MATLAB data structure, but quite a few conventions have developed as NARF has evolved. The following are typical values and data structures being used. Feel free to define your own fields as necessary.

WARNING: THIS DOCUMENTATION IS HORRIBLY OUTDATED SO DON'T TRUST ANYTHING PAST THIS POINT.

SYMBOL	DESCRIPTION	TYPE	SET OR MODIFIABLE BY
`X.dat.().cellid`	Name of the cellid	String	-
`X.dat.().stimfile`	Name of the stimfile	String	-
`X.dat.().include_prestim`	Boolean. 1 prestim was included, 0 otherwise	Boolean	load_stim_resps_from_baphy.m
`X.dat.().raw_stim_fs`	Raw stimulus frequency	Double	load_stim_resps_from_baphy.m
`X.dat.().raw_resp_fs`	Raw response frequency	Double	load_stim_resps_from_baphy.m
`X.dat.().raw_stim`	Raw stimulus	[??]	load_stim_resps_from_baphy.m
`X.dat.().raw_stim_time`	Time vector for stimulus	[??]	load_stim_resps_from_baphy.m
`X.dat.().raw_resp`	Raw spike timings	[??]	load_stim_resps_from_baphy.m
`X.dat.().raw_resp_time`	Time vector for response	[??]	load_stim_resps_from_baphy.m
`X.dat.().raw_isi`	Raw inter-spike intervals		
`X.dat.().pp_stim`	Preprocessed stim		
`X.dat.().ds_stim`	Downsampled, preprocessed stim		
`X.dat.().ds_stim_time`	Time vector for downsampled stimulus		
`X.dat.().lf_stim`	Linear filtered stimulus (FIR or whatever)		
`.lf_preds`	Needs to be RENAMED		
`X.dat.().nl_stim`	Nonlinearly scaled stimulus		
`X.dat.().pred`	Prediction of the model		

In the above, dimensions are indicated with:

T = Time index

S = Stimulus index #

C = Channel index #

R = Repetition index #

Coefficients are the only things that AREN'T with the time index in first place. Why? Well, it's easier to edit them in the gui if they are not?

1.3 Overview of Typical Operations

The following is a rough overview of how typical operations affect the `STACK` and `XXX` structs.

Evaluation: Essentially, there is a chain of function calls, with the output of one function pushed onto the inputs of the next. Mathematically, it's easy to understand: `XXX{i+1} = STACK{i}.fn(XXX{i})`

Invalidation: If any intermediate parameter struct is modified, then it erases all `XXX` cells after it and the computation must recommence from that point.

Multiple models: If you need to do different 'branches' of computation or compare model structures, you can store the current computation `STACK` and save them somewhere, then load them back later.

Module loading: The only functions available are isted in the "modules" directory, which is read ONCE, at startup. They are only available from the popup selection when their ready_pred() function returns a true.

Editing: Fields in the `editable_fields` struct are GUI editable, but most other fields should be editable by passing the argument to the user-defined module creation funciton. However this is just a convention and relies on module authors to implement.

Graphing: Each module has (multiple) associated graphing functions which can be seleceted via a dropdown.

Error handling: Whenever you load or run a user-loadable function, a try-catch block around it to catch errors.

Saving and loading: When you want to save a model, just save the `STACK` data structure somewhere along with the GIT hash tag and initial data. Data from that point can always be reconstructed. When you want to load a model, loop through the `STACK` structure, starting from the first data X, and reconstruct the data as you go along.

Optimization pack/unpack: Packing operations go through `STACK` sequentially, pulling out any args found in the `fit_fields` cell array and putting them into a vector. The unpacking operation goes through `STACK` sequentially, pushing in any args with a FIT checkbox (accepts a vector as the input). TODO: During optimization, all controls should probably be disabled to avoid invalidation problems?

Optimization There will need to be three types of modules: a performance metric, termination condition, and a sampling algorithm which picks out the next point to try. These are not part of the model explicitly, and must be reusable amongst different optimization algorithms.

1.4 Allowing Varying Signal Dimensionality

There is an intrinsic problem in science in that we often do not know how many dimensions we will need to test before starting an experiment. For example, when presenting an auditory stimulus during an experiment, there may be multiple dimensions to the stimulus which need to be indexed:

1. The amplitude of the sound as a function of time
2. Which experiment the data came from
3. Which particular sound that was played (i.e., a particular stimuli fragment, usually given an index number)
4. Whether the sound is of one type or another type, such as during discrimination tasks.
5. The index number of the speaker playing the stimuli
6. The L/R index of which ear the animal is listening with
7. The repetition index number, if a stimulus somehow varies in a minor way across each major stimulus pattern.

For visual presentations of images, the problem can become even worse due to the higher dimensionality of image stimuli, although this depends on how the video stimuli is represented as data, and whether or not each pixel has an X component, a Y component, and RGB color dimensions.

There are similar problems with dimensionality in regards to model complexity. It is often the case that there will be extra dimensions internal to the model used to help predict neural response to a stimulus. For example, a model may have:

1. Linear or nonlinear combinations of input data channels intended to model higher order terms.
2. A filter bank applied to a single stimuli, converting a single stimulus into many.
3. Multiple macro channels of FIR filters, acting on different portions of the signal

Whether changing dimensionality results from changes to model structure or from the experiment itself, the problem is the same: it is hard to perfectly predict when extra dimensions will need to be added or removed, especially when many models must be tried.

Allowing arbitrary NARF modules to work with every possible experimental dimensionality is a very difficult problem to solve and would require some sort of clever tensor notation. This turned to be too difficult to implement...so it wasn't. Instead, things were just coded up as 3D or 4D matrices. You may have to tweak your data to make it fit into the expected interface of another module; that's just life.

1.5 Model Auto-Initalization Not Encouraged

One lazy design decision made was not to pass the `STACK` and `XXX` as arguments to each module creation function. Although this could easily be done to allow modules to automatically initialize themselves, most of the time we want to set their values via a script anyway. Initializing a module from more than one place could lead to confusion about who sets the last, final value of the module, so this approach was avoided.

1.6 Example Optimization

Clearly optimization is clearly NOT part of the model, yet we will want to plot various quantities and act in a modular way in a similar fashion to how the model already works. Pluggable sampling, termination conditions and objective functions are needed.

1.7 Further Development

See the TODO.org file in NARF's main directory.

Chapter 2

NARF GUIs

Not much to say here; `narf_browser.m`, `narf_analysis.m` and `narf_modelpane.m` are the big three.

Chapter 3

The Keyword Composition System

Keywords are arbitrary functions. The concept is similar to how the STACK works, but intermediate values are not preserved.

Chapter 4

Optimization: Fitters & Performance Metrics

Fitters go in the `fitters/` directory.

Chapter 5

Distributed Computation

Chapter 6

Advanced Functionality

6.1 Parameter Sets

6.2 Jackknifing

Chapter 7

List of Modules