On the design of script languages for neural simulation

Romain Brette
romain.brette@ens.fr

(1) Laboratoire Psychologie de la Perception, CNRS, Université Paris Descartes, Paris, France
(2) Equipe Audition, Département d'Etudes Cognitives, Ecole Normale Supérieure, Paris, France

Running title: On the design of script languages

Abstract

In neural network simulators, models are specified according to a language, either specific or based on a general programming language (e.g. Python). There are also ongoing efforts to develop standardized languages, for example NeuroML. When designing these languages, efforts are often focused on expressivity, that is, on maximizing the number of model types than can be described and simulated. I argue that a complementary goal should be to minimize the cognitive effort required on the part of the user to use the language. I try to formalize this notion with the concept of "language entropy", and I propose a few practical guidelines to minimize the entropy of languages for neural simulation.

Introduction

In neural network simulators, models are defined using a specific language, or with a set of objects and functions defined in a general programming language, e.g. Python. For example, NEST initially used a specific stack-based language named SLI, based on reverse Polish notation. It now uses a Python interface, in which neuron models are specified by predefined strings (Eppler et al., 2009). For example, `neuron = nest.Create("iaf_neuron")` creates an integrate-and-fire neuron with default parameter values, which can be accessed by their predefined names. Neuron uses a high-level script language named Hoc, as well as a lower-level language called NMODL to describe channel dynamics (Hines and Carnevale, 2000). It also has a Python interface that can replace the Hoc language (Hines et al., 2009). Recently, a common Python interface to several simulators was developed. In this language, PyNN (Davison et al., 2009a), the following instruction creates a group of 20 integrate-and-fire neurons with alpha synaptic currents and specified values for the membrane time constant and capacitance:

```
p= Population(20, IF_curr_alpha, cellparams={'tau_m': 15.0, 'cm': 0.9})
```

This example is representative of the approach taken by many simulation languages: there are a number of predefined models, with parameter values accessed by their predefined names. The units are also implicit (e.g. `tau_m` is in ms). An exception to this approach is the Brian simulator (Goodman and Brette, 2009): models are also defined in the Python language, but the general strategy is to minimize the number of predefined objects. This means for example that neuron models are defined by differential equations, written in mathematical form with explicit units, a threshold condition and reset operations. A recent standardization initiative, NineML, uses a similar approach (Raikov et al., 2011).

What principles underlie these design choices? It appears that the main concern is generally expressivity. That is, a language is designed so as to express the models that the simulator is designed to run. In general, it is fair to observe that little attention is devoted to the particular choice of names (e.g. "`IF_curr_alpha`") or to the syntax. The goal of this note is to draw the attention of developers on this underestimated issue.

Language entropy

Often, we tend to think that names and syntax of functions are somewhat arbitrary. Indeed, once the user knows it, or reads it in the documentation, any particular choice is arbitrary. But this point of view cannot be right. As a thought experiment, imagine that your scripting language, which I shall call Zübkl, entirely consists of commands with seemingly random names. All these names have 6 letters, among the 26 letters of the English alphabet. A typical Zübkl script would be:

```
lbhyev f(n):
      peknnx n == 0:
            kkpden 1
      kkpden n*f(n-1)
xghtui f(5)
```

You might have guessed that this script defines the factorial function and prints the factorial of 5. But you will certainly agree that Zübkl is a rather difficult language to learn, to read and to use.

You will probably find yourself constantly consulting the documentation when writing a script. Why?

Take the example the print command, which is called `xghtui` in Zübkl. When I designed the Zübkl language, I chose 6 letters at random for each command. This means that for any given command that I want to execute, I need to remember one particular combination out of all possible combinations of 6 letters. Each letter represents 26 possibilities or $\log_2(26)=4.7$ bits, so each word represents 28 bits of information or "entropy". Thus, we may consider that memorizing the name of the print command has a cognitive cost of 28 bits. In contrast, consider the command name `print`. This command name, in many languages, is used to display something on the screen. There are of course many variations, but they are generally very similar. Naming it `print` rather than `xghtui` has several advantages:

- It is an English word. Given that there are about a million words in the English language, restricting command names to English words reduces the entropy of command names to about 20 bits – with the conservative assumption that all words are equally likely.
- The meaning of the name `print` corresponds to what the command does. If we assume that only 30 verbs (synonyms) satisfy this constraint, then this principle reduces the entropy to about 5 bits.
- The command could also be written `Print` or `PRINT`. Enforcing lower case for all commands reduces the uncertainty, for a benefit of $\log_2(3)=1.5$ bit (assuming three possibilities for the case).
- Finally, this command is named `print` in almost all languages. Thus, choosing to name it `print` rather than, for example, `display`, reduces the entropy to near 0 bit.

This example highlights several principles which I will comment in more details below. In terms of (informal) information theory, the idea is to minimize the conditional entropy of names and syntax in the language, conditioned to the meaning. I shall call this (loose) definition the "language entropy". By enforcing syntax rules (whether explicit or implicit) and carefully chosen names, one can reduce language entropy.

Entropy of single functions or objects

On the benefit of enforcing syntax and naming rules

To give a simple example, suppose we allow composite names to be written as `compositenames` or as `composite_names`. For a given name, this adds two possible choices. Therefore, the cost of allowing this possibility is 1 bit, in terms of entropy. Therefore, there should be only one allowed possibility. We could, however, imagine that the two options are possible, if there is a clear rule for selecting between the two options – in this case the entropy is 0 bit. More generally, naming rules reduce language entropy. Therefore, it is a good idea to enforce naming conventions such as the PEP-8 guidelines in Python, not just from the point of view of "good style" but simply because it reduces language entropy.

Short names or long names?

At first sight, it could seem that shorter names are easier to remember, and therefore reduce cognitive load. But I would argue that the opposite is true, from the point of view of language entropy. The problem with short names is that there are generally several possible abbreviations of the same name. For example, in Brian, a group of neurons is called `NeuronGroup`. If it were shorter, we could have called it `NrnGroup` or `NeuronGrp`. But the sole fact of choosing an abbreviation means the user has to remember which one is right, with a cognitive cost of 1 bit.

Choosing a good name

Explicit naming still leaves many possibilities in general. A good name should correspond to the meaning of the function or class, in the most possible obvious way. There will almost always be several possibilities to name a given concept. But, at least, one simple rule should be that the name should unambiguously evoke the desired concept. That is, meaning should be obvious from the name.

Named keywords

Python has become the interface language of most simulators (Davison et al., 2009b). It gives the possibility to use named keywords, that is, instead of passing arguments as a list with a predefined order (x,y,z), one can pass them explicitly as (x=2,y=3,z=4) or (y=3,x=2,z=4), for example. Which choice minimizes language entropy?

With the list approach, one needs to remember the order of arguments, but not their name. In the named keyword approach, one needs to remember the name, but not the order. The best choice depends on the number of arguments. Indeed, what is the entropy of a ranking of n arguments? The number of possibilities is n!, therefore the entropy is $\log_2(n!)$. This grows as $n\log_2(n)$, that is, faster than n. On the other hand, the cognitive cost of remembering n names grows linearly with n. Therefore, named keywords are better when there are many arguments, while the list is better for few arguments. Of course, in practice, it depends on how obvious the names and the order are. As a rule of thumb, I would suggest that named keywords are better from the third or fourth argument, while the order is generally obvious for two arguments (and one as well, obviously).

Dynamic typing

A great feature of the Python language in terms of language entropy is dynamic typing. This means that the type of arguments of a function is not predefined, type is only checked at run time. Therefore the same function, that is, a single name, can be used with arguments of different types. For example, in the Brian simulator, one can specify the synaptic weights when creating synapses between two groups of neurons, using the `weights` keyword. These three examples below show how dynamic typing is used to specify a uniform weight, a random weight for each synapse, or a weight that depends on pre- and postsynaptic neuron through a user-defined function:

```
weight=2*nS
weight=rand(N,M)
weight=lambda i,j: exp(|i-j|)
```

Combined entropy

In the previous section, I examined language entropy by considering isolated functions. By looking at the entire language, we can set additional rules to minimize language entropy. The idea I want to expose here is an elementary notion of information theory. The entropy of a set of variables $H(X_1, ..., X_n)$ is just the sum of individual entropies $H(X_i)$ if these variables are independent. However, it is potentially much smaller if they are correlated. In the same way, to minimize language entropy, one should introduce correlations between syntax choices for all functions and objects, by way of explicit or implicit rules. The bottom line is that consistency in syntax and naming reduces language entropy.

Consistency can be enforced by naming conventions, such as PEP-8. But it is not strictly necessary that these rules be explicit. A simple rule of thumb when naming a new function, is to find the inspiration in the most similar existing functions. For example, in Brian, all classes that continuously record variables of a simulation are called `Monitor`, for example `StateMonitor` for state variables and `SpikeMonitor` for spike times. Compared to an uncorrelated alternative such as `StateMonitor` and `SpikeRecorder`, this consistency saves 1 bit of language entropy.

This idea goes a long way: it also applies to argument names and order, expected types and class methods, etc. For example, all functions that process spike times should use the same argument name, for example `spikes`, and the type should always be the same, e.g. a list of floats. A positive side effect of choosing consistent names is that, in a dynamically typed language, it allows *duck typing*. This refers to the idea that the semantics of an object is defined by its sets of methods and properties rather than by its belonging to a specific class.

This suggests a few simple entropy-reduction rules when introducing a new function in the language:

1) If it applies an existing operation to a new type, then it might be better to extend the existing function using dynamic typing, rather than to create the new function.
2) If it is a new function, its name should be similar to (or follow the same logic as) those of related functions.
3) Arguments (or variable and method names for classes) should be named and typed in the same way as arguments with a similar meaning in other functions.

Equation-oriented design

The previous remarks were not highly specific to neural simulations. I will now discuss more specifically the description of neuron models. There are many existing neuron models, and probably, there will be many more in the future. To use a model, one option, used in many simulators, is to choose one from a set of predefined models (for example, an integrate-and-fire (IF) model with exponential synapses). This may be a reasonable option if the simulator is specialized for a limited set of models, but in general this design choice increases language entropy very substantially. First, one needs to remember the exact name of the model. Given the modularity of models (e.g. an IF model could have exponential or alpha synapses), this means either very long names or ambiguous names (high entropy). More importantly, one needs to

remember the names of variables, and what exactly they refer to. When there are many possible models, it is unlikely that the average user can do this without referring to the documentation.

Another strategy, used in the Brian simulator, is to let users define the models. That is, the user provides the mathematical equations of the model (including the condition for spiking and what happens at reset), providing the names of the variables with their units. The equations are written in standard mathematical language, and therefore contribute little additional language entropy. This equation-oriented design considerably reduces language entropy, because there is no more need to remember the names of models or of state variables. The same design applies to spike-timing-dependent plasticity models. In fact, the current version of Brian features a new class called `Synapses`, which unifies the description of synaptic dynamics, including nonlinear synapses, gap junctions, short-term and long-term plasticity. The user defines differential equations on synaptic variables, and operations that are executed on pre- and post-synaptic spikes. For example, synapses with stochastic transmission can be defined as follows:

```
S=Synapses(source,target,model="""w : 1
                                 p : 1""",
                           pre="v+=w*(rand()<p)")
```

where `source` is the presynaptic group of neurons, `target` is the postsynaptic group and `v` is the membrane potential of the postsynaptic group, introduced by the user in the definition of that group. When a presynaptic spike is produced, variable `v` in target neurons is increased by an amount `w` with probability `p`.

Arguably, for complex models (e.g. Hodgkin-Huxley models), this equation-oriented design potentially means very long sets of differential equations. In practice, the user would probably end up copying and pasting previous code for e.g. sodium channels. But is this a great penalty? The alternative is to look up in the documentation for the name and specific syntax of the model. This alternative does not save time, and does not increase readability.

Conclusion

My goal in this note was to convince developers of neural simulators that syntax is an important issue, and also that it is not an arbitrary choice. I tried to conceptualize this choice with the notion of *language entropy*, which corresponds to the cognitive effort required on part of the user to use the language. I suggested a number of rules to reduce language entropy, summarized below:

- use obvious, explicit names, with consistent naming and syntax rules
- exploit dynamic typing
- define models by their mathematical description

It is tempting and natural for developers to increase the functionality of their favorite simulator by adding new objects, new functions, new models. This temptation should be resisted. In fact, I suggest that developers should follow the opposite trend: to increase functionality by generalizing existing mechanisms rather than by adding new special cases.

Acknowledgments

This work was supported by the European Research Council (ERC StG 240132).

Declaration of Interest

The author reports no conflicts of interest.

References

Davison, A.P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009a). PyNN: a common interface for neuronal network simulators. Front. Neuroinform. *2*, 11.

Davison, A.P., Hines, M.L., and Muller, E. (2009b). Trends in programming languages for neuroscience simulations. Front. Neurosci. *3*,.

Eppler, J.M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: A convenient interface to the NEST simulator. Front. Neuroinform. *2*, 12.

Goodman, D.F.M., and Brette, R. (2009). The Brian simulator. Front. Neurosci. *3*,.

Hines, M.L., and Carnevale, N.T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. Neural Comput *12*, 995–1007.

Hines, M.L., Davison, A.P., and Muller, E. (2009). NEURON and Python. Front. Neuroinform. *3*, 1.

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., De Schutter, E., Djurfeldt, M., Gleeson, P., Gorchetchnikov, A., Plesser, H., et al. (2011). NineML: the network interchange for neuroscience modeling language. BMC Neuroscience *12*, P330.