NeurOn Neural Network Design Language and Compiler

1.0.0 Alpha



User Manual

Contents

* About neurOn
* Command Line Arguments
* Compilation Chain
* Source and Object files
* Syntax
  + Expressions
  + Groups
  + Lists
  + Assignment
* Operators
* Variables
* Macros
* Expressions
* Functions
* Inline API
* Dependencies
* Makefile compilation

About neurOn

Biological neural systems are extraordinarily complex. In a SAPA system, the intention is to create a sort of hybrid logical system that behaves similarly to a biological model but built to function on modern hardware. The intent is to create programs capable of complex learning.

With rapid development in mind, it can be extremely tedious and time consuming to constantly write, re-write and edit lower level code. Thus, neurOn offers a higher level way of designing systems without worry about the underlying overhead.

The principle is similar to a hardware design language such as VHDL or Verilog. Circuits may be designed via written expressions and compiled into a final product. However, there are several major differences.

First of all, SAPA systems are dynamic. The program written and compiled will only yield an optimized program and a starting point. From there, a system is free to evolve and learn as time progresses. Secondly, the systems intended to be created are significantly more complex than an average electrical circuit. Thus, expressions are designed with extreme complexity in mind and make heavy use of loops and regional generations.

The intent of this part of the SAPA toolset makes the development of learning machines quick and (relatively) simple. The remainder of this manual contains definitions, explanations and examples to begin creating unique neural systems.

Command Line Arguments

The table below lists all accepted command line arguments. Note that options must be set in the form of -[n] or --[setting]=*value* with no spaces in an expression.

|  |  |
| --- | --- |
| -v | Print version |
| -h | Print Help Dialogue |
| -o | Activate optimization |
| -d | Debug mode |
| -e | Enable evolution |
| -w | Suppress warnings |
| -s | Preserve C source code |
| -c | Compile to object and C files, do not compile to binary |
|  |  |
| --bitmode=*32/64* | Set to 32 or 64 bit compilation |
| --output=*executable* *output directory* | Directory to output build |
| --title=*name* | Set build name |
| --language=*output* *language* | Set compiled language |
|  |  |
|  |  |
|  |  |
|  |  |

Compilation Chain

Once all source files are complete, the compilation process begins. Although the design information is structural an entry point is still required. This entry point is the inputs. The neurOn compiler will accept a single neurOn file (*.nrn*). This may link to other source files. However, multiple definition files (*.stem*) may be passed directly to the compiler, although they may also be referenced by the source files as well.

Compilation begins with source and definition files passed into the compiler. The compiler will first look for definitions and diagram instructions. Object declarations, such as cell types and region definitions, are added to a symbol reference table to be drawn upon at a later time. Instructions on how to assemble the circuitry is converted to bytecode. An object file is then produced with the extension *.cnctm*, which stands for “connectome”, or the connection data of a neural network.

This object file harbors the entirety of the project. An object contains all cell type references and the hand-designed circuits. Intentionally, this is left fairly vague, and is only intended to describe a single particular system without additional unnecessary information. From here, conversion to a programming language is possible.

When the neurOn compiler is given an object file, or if the *–language* flag is set during initial compilation, the connectome file passed will begin being converted into a common language. This release intends to support the Java and C languages. The output is placed in a folder within the project directory, and a makefile that can be used to compile the project into an executable form.

While this may seem excessive, with regards to source files being mixed into an object file, converted into another source file and then finally into a functioning project, there is a reason for this.

A SAPA executable comes with the ability to write its own connectome to an object file. As these systems are autonomous and polymorphic, they will re-design themselves over time. The actual executable exists merely as a way to optimize how data is processed. For instance, one fully compiled system of any language or platform will be capable of simulating any connectome with the same dependencies. A system saves itself by writing its connectome as an object file. In fact, only the definition file is truly necessary in creating a system! The executable will ensure that it contains all data types as required by a connectome during load-time, and reject the file if not.

Source and Object Files

Syntax

Writing a neurOn design script consists of three general tasks: configuring compilation settings, declaring objects, and designing the neural circuitry.

**Configuring compilation settings**

When compiling a system, there are a number of adjustments that can be made. These primarily pertain to environmental information that influences the operation of the virtual cells, as well settings that adjust the compilation process.

These settings may be defined in two ways. First is through the *settings operator* followed by the value to adjust and its value. This may be written as

Where ‘!’ denotes a system configuration variable, followed by a variable name with no spaces in between. The ‘&’ operator denotes an attribute to be assigned to a variable. Multiple or no attributes may be assigned to a variable.

Below is a table that describes all available system settings.

|  |  |
| --- | --- |
| ECSC | Extra cellular space concentration; This denotes the system equilibrium concentration of ions. Cells will use this value as their baseline. A higher ECSC will provide a faster partial-refractory rehabilitation stage, and will dim the effects of ionic influx of the opposite charge. The default value is 0; |
| TDF | Time dilation factor; This sets the maximum system update cycles per second. If this value is 0, it will not be compiled into C. The default value is 0; |
|  |  |
|  |  |
|  |  |
|  |  |
| BITM | Bit Mode; Sets the compiled bitmode. This value may be 32 or 64. The default value is the bitmode of the compiler binary itself. |
|  |  |

In addition to system settings, certain compilation attributes may be defined. These set special rules on system variables. An attribute follows the ‘@’ operator, and multiples may be defined separated by whitespace.

|  |  |
| --- | --- |
| volatile | This states that a re-definition of a set variable will override the current definition. This is on by default for all files except the master file. |
| absolute | Opposite of the volatile attribute, this defines that the current definition of a variable cannot be modified. This will throw a warning if another attempt to change it is made and a fatal error if another absolute modification is attempted. This is on by default for the master file. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Dependencies and Imports**

A dependency is an external piece of code that defines an object used somewhere else. For instance, if a file FileA defines a type of neuron that FileB requires, FileB may make a reference to FileA, which is then scanned and added to the definition database. When a file is imported, it is added to a list with its absolute directory so that it ignored when referenced in the future to prevent circular dependencies.

To import an external file, the *import* keyword may be invoked. For example,

imports both FileA and FileC.