CoreNEURON

Michael Hines and the BlueBrain Project

Jan 12, 2021

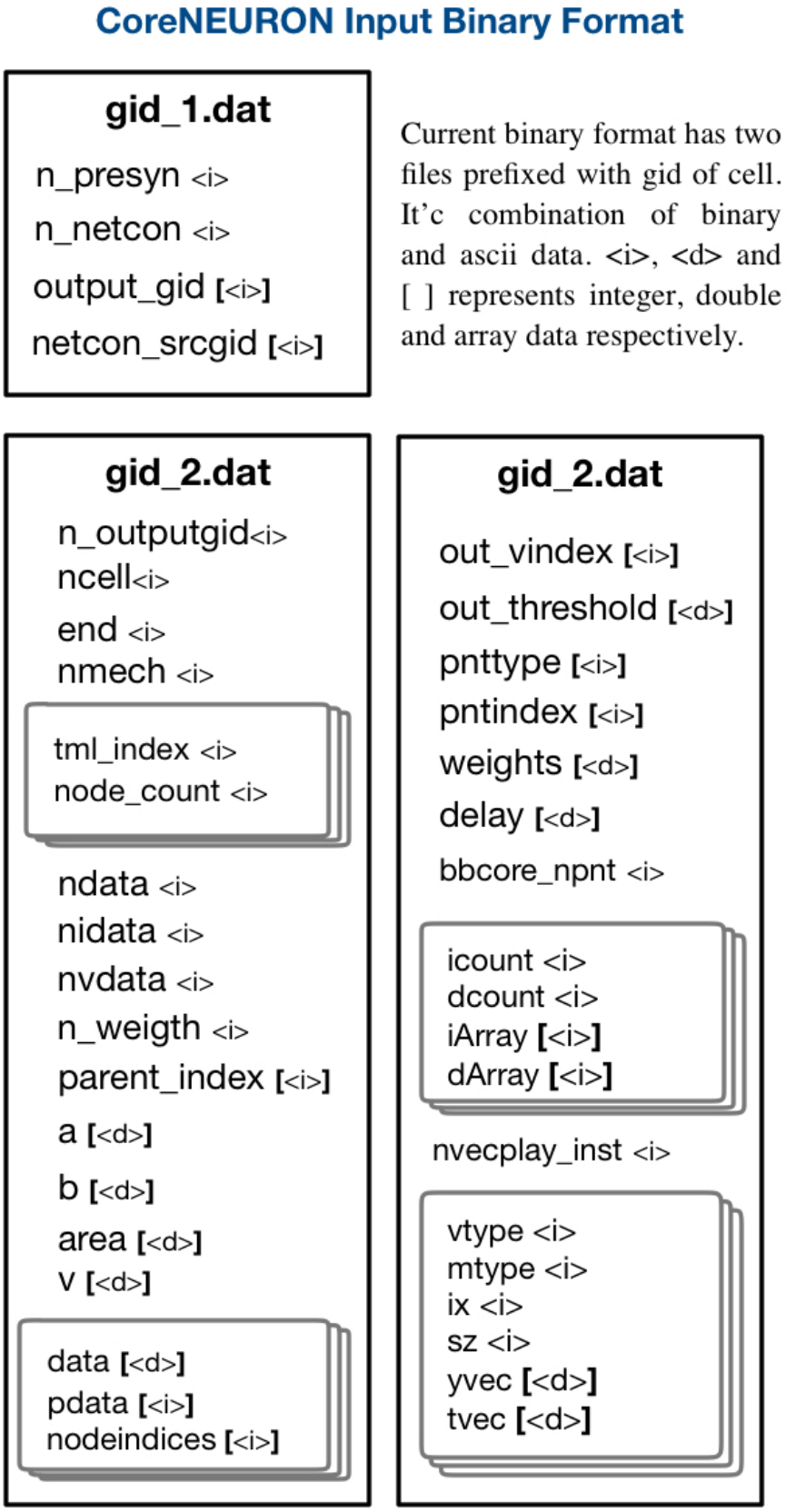
USER DOCUMENTATION:

CHAPTER

ONE

CORENEURON INPUT BINARY FILE FORMAT

NEURON is used for building in-memory model of the network. The in-memory representation of model is then dumped to binary files and read by CoreNEURON. The abstract structure of these binary files is shown :



Binary File Format.

Note : additional datasets are being added for additional functionality (e.g. Gap Junctions). This dcoumentation / format will be updated in the future.

CHAPTER

TWO

TRANSFERRING DYNAMICALLY ALLOCATED DATA BETWEEN NEURON AND CORENEURON

User-allocated data can be managed in NMODL using the `POINTER` type. It allows the programmer to reference data that has been allocated in HOC or in VERBATIM blocks. This allows for more advanced data-structures that are not natively supported in NMODL.

Since NEURON itself has no knowledge of the layout and size of this data it cannot transfer `POINTER` data automatically to CoreNEURON. Furtheremore, in many cases there is no need to transfer the data between the two instances. In some cases, however, the programmer would like to transfer certain user-defined data into CoreNEURON. The most prominent example are random123 RNG stream parameters used in synapse mechanisms. To support this use-case the `BBCOREPOINTER` type was introduced. Variables that are declared as `BBCOREPOINTER` behave exactly the same as `POINTER` but are additionally taken into account when NEURON is serializing mechanism data (for file writing or direct-memory transfer). For NEURON to be able to write (and indeed CoreNEURON to be able to read) `BBCOREPOINTER` data, the programmer has to additionally provide two C functions that are called as part of the serialization/deserialization.

```
static void bbcore_write(double* x, int* d, int* d_offset, int* x_offset, _
↪threadargsproto_);

static void bbcore_read(double* x, int* d, int* d_offset, int* x_offset, _
↪threadargsproto_);
```

The implementation of `bbcore_write` and `bbcore_read` determines the serialization and deserialization of the per-instance mechanism data referenced through the various `BBCOREPOINTER`s.

NEURON will call `bbcore_write` twice per mechanism instance. In a first sweep, the call is used to determine the required memory to be allocated on the serialization arrays. In the second sweep the call is used to fill in the data per mechanism instance.

The functions take following arguments

- `x`: A `double` type array that will be allocated by NEURON to fill with real-valued data. In the first call, `x` is NULL as it has not been allocated yet.
- `d`: An `int` type array that will be allocated by NEURON to fill with integer-valued data. In the first call, `d` is NULL as it has not been allocated yet.
- `x_offset`: The offset in `x` at which the mechanism instance should write its real-valued `BBCOREPOINTER` data. In the first call this is an output argument that is expected to be updated by the per-instance size to be allocated.
- `d_offset`: The offset in `x` at which the mechanism instance should write its integer-valued `BBCOREPOINTER` data. In the first call this is an output argument that is expected to be updated by the per-instance size to be allocated.

- `_threadargsproto_`: a macro placeholder for NEURON/CoreNEURON data-structure parameters. They are typically only used through generated defines and not by the programmer. The macro is defined as follows:

```
#define _threadargsproto_                                                    ␣
↪             \
    int _iml, int _cntml_padded, double *_p, Datum *_ppvar, ThreadDatum *_thread,␣
↪NrnThread *_nt, \
    double _v
```

Putting all of this together, the following is a minimal MOD using BBCOREPOINTER:

```
TITLE A BBCOREPOINTER Example

NEURON {
    BBCOREPOINTER my_data
}

ASSIGNED {
    my_data
}

: Do something interesting with my_data ...

VERBATIM
static void bbcore_write(double* x, int* d, int* x_offset, int* d_offset, _
↪threadargsproto_) {
    if (x) {
        double* x_i = x + *x_offset;
        x_i[0] = _p_my_data[0];
        x_i[1] = _p_my_data[1];
    }
    *x_offset += 2; // reserve 2 doubles on serialization buffer x
}

static void bbcore_read(double* x, int* d, int* x_offset, int* d_offset, _
↪threadargsproto_) {
    assert(!_p_my_data);
    double* x_i = x + *x_offset;
    // my_data needs to be allocated somehow
    _p_my_data = (double*)malloc(sizeof(double)*2);
    _p_my_data[0] = x_i[0];
    _p_my_data[1] = x_i[1];
    *x_offset += 2;
}
ENDVERBATIM
```

CHAPTER

THREE

C++ API

Link to doxygen C++ API

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search