Webnucleo Technical Report: Views in libnucnet

Bradley S. Meyer

April 21, 2012

This technical report describes how to work with views in libnucnet.

1 Views

In libnucnet, a Libnucnet__Nuc structure stores data for a collection of nuclides while a Libnucnet__Reac structure stores data for a collection of reactions. A network, which is stored as a Libnucnet__Net structure, is then a combination of the nuclide collection and the reaction collection. Valid reactions in the network are nucleon-number-conserving, lepton-number-conserving, and charge-conserving reactions in the Libnucnet__Reac structure among nuclides stored in the network's Libnucnet__Nuc structure. A reaction in the network is invalid if it does not conserve nucleon number, lepton number, or charge, or if any of the reactants or products are not included among the network's nuclide collection.

Libnucnet__Nuc and Libnucnet__Reac structures store a considerable amount of information about their nuclides and reactions. In many cases it is desirable to have a subset of these nuclides or reactions. The libnucnet API provides the routines Libnucnet__Nuc__extractSubset() and Libnucnet__Reac__extractSubset(), which, respectively, create new nuclide and reaction collections from the original collections through the use of XPath expressions. Importantly, these new collections "own" the nuclide or reaction data in the sense that the extractSubset() routines make copies of those data from the original collections. Any modification of the data in the extracted subset does not affect the data in the original collection.

While these new collections have their uses, it is often preferable to have a "view" of the collection. A view is a subset of collection that does not own its own data. Rather, the data in the view are simply pointers to the data in the original collection. An advantage of a view over an extracted subset, then, is that it requires a considerably smaller amount of memory than the original collection does. Also, since the view does not own its data, modifying the data for a nuclide or reaction in a view modifies the nuclide's or reaction's data in the original collection. This means, for example, that one can modify the data in a nuclide collection for neon isotopes by getting a view of the collection that only includes the neon isotopes and iterating over the species in the view and

modifying their data. This automatically modifies the data for the neon isotopes in the original collection.

The possible views a user can create are a Libnucnet__NucView, a view of a nuclide collection, a Libnucnet__ReacView, a view of a reaction collection, and a Libnucnet__NetView, a view of a network. It is important to note that the only reactions included in a Libnucnet__NetView are ones that are valid for that view.

2 Libnucnet__NucView

A Libnucnet__NucView is a view of a nuclide collection. It is created with the Libnucnet__NucView__new() routine that takes as arguments the original Libnucnet__Nuc pointer and an XPath expression to select the species to include from the original collection in the view. The view collection may be accessed via the API routine Libnucnet__NucView__getNuc(), and the pointer returned from this routine may be passed into any routine that takes a Libnucnet__Nuc structure. The user then frees the view with Libnucnet__Nuc__free().

For example, to count the number of neon isotopes in an existing nuclide collection p_nuc, one could create a view of neon isotopes and get the number of species in it:

```
p_view = Libnucnet__NucView__new( p_nuc, "[a = 10]" );

printf(
  "The number of neon isotopes is %lu.\n",
  Libnucnet__Nuc__getNumberOfSpecies(
    Libnucnet__NucView__getNuc( p_view )
  )
);

Libnucnet__NucView__free( p_view );
```

It is important to note that p_nuc and all its data still exist after these operations since p_view never owned p_nuc's data.

3 Libnucnet__ReacView

A Libnucnet__ReacView structure is a view of a reaction collection. It is exactly analogous to a Libnucnet__NucView structure in that it is created with Libnucnet__ReacView__new(), which takes as arguments an existing reaction collection and an XPath expression to select the reactions to include. The view collection is accessed with Libnucnet__ReacView__getReac(), and the view is freed with Libnucnet__ReacView__free().

4 Libnucnet__NetView

A Libnucnet__NetView structure is a view of a Libnucnet__Net structure containing a subset of species of the original structure and valid reactions among the view's species. It is created with Libnucnet__NetView__new(), which takes the original network and two XPath expressions as arguments. The first argument is the XPath expression that selects the nuclides from the original network to include in the view. The second XPath expression selects the reactions to include in the view. The routine returns a view containing a subset of the species in the original network and the valid reactions among those species that satisfy the reaction XPath constraint.

After a view has been created, it is possible to add or remove reactions from the view with Libnucnet__NetView__addReaction() or Libnucnet__NetView__removeReaction(). It is worth noting that since adding a reaction requires a check that the reaction is valid for the view, this operation is slower than removing the reaction, which simply deletes the reaction pointer from the underlying hash. A network view can be accessed with Libnucnet__NetView__getNet(). A user can copy a network view with Libnucnet__NetView__copy(), which returns a new network view that is a copy of the input one. The user frees a view with Libnucnet__NetView__free().

While network views can be used on their own, it is also possible to store them in Libnucnet__Zones. This is convenient because the user can simply lookup a view rather than create it, an operation that requires numerous checks on reaction validity. An existing network view can be added to an existing zone with the command Libnucnet__Zone__updateNetView(), which adds the view to the zone if it did not previously exist or replaces the existing view with the new one. This routine takes as arguments the zone, three labels for the view, and the view. The user subsequently looks up the view from the zone with the three labels using Libnucnet__Zone__getNet().

It is frequently the case that the logical choices for two of the labels for a view in a zone are the XPath expressions that created the view, especially if no reactions have been added to or removed from the view since it was created. In this case, the third label can simply be NULL. The labels, however, need not be XPath expressions. For example, the network evolution (change of abundances with time) is computed from an evolution network view, which has labels (EVOLUTION_NETWORK, NULL, NULL). To change the evolution network, then, the user would create a view and then update the evolution view in p_zone, the zone of interest. To limit the evolution network to (n, γ) reactions on nuclei with $Z \leq 50$, the user would write:

```
p_view =
  Libnucnet__NetView__new(
    "[z <= 50]",
    "[reactant = 'n' and product = 'gamma']"
  );
```

```
Libnucnet__Zone__updateNetView(
  p_zone,
  EVOLUTION_NETWORK,
  NULL,
  NULL,
  p_view
);
```

libnucnet routines would then use this network to evolve abundances until the evolution view was updated again.

Because a network view is created from a parent network, it is conceivable that the parent network might have changed since the view was generated. For example, suppose a user generates Libnucnet__NetView * p_view from Libnucnet__Net * p_net. Now suppose the user adds a new species to the nuclide collection in p_net. p_view will not include that species. At this point the user will want to delete p_view and generate a new view.

A user can check whether the parent network of a view has been updated since the view was generated with the API routine Libnucnet__NetView__wasNetUpdated(). This routine returns 1 (true) if the parent network has been updated since the view was generated or 0 (false) if not. Checking for an update will allow a user to decide whether to regenerate a view or not.

A user can iterate over the network views stored in a zone with Libnucnet__Zone__iterateNetViews() and apply a user defined Libnucnet__NetView__iterateFunction to them. To do so, the user writes a routine with prototype

```
int
my_net_view_iterator(
  Libnucnet__NetView * p_view,
  const char * s_label1,
  const char * s_label2,
  const char * s_label3,
  void * p_data
);
```

In this prototype, p_data is a pointer to a user-defined data structure carrying extra data for the routine. The routine must return 1 (true) for iteration to continue or 0 (false) for iteration to stop.

The user then iterates over the network views in p_zone and applies my_net_view_iterator using p_data with

```
Libnucnet__Zone__iterateNetViews(
  p_zone,
  s_1,
  s_2,
  s_3,
  (Libnucnet__NetView__iterateFunction) my_net_view_iterator,
```

```
  p_data
);
```

This iterates over all network views in p_zone that have labels that match s_1, s_2, and s_3 and applies my_net_view_iterator to each view. If s_1, s_2, or s_3 is NULL, any view label is a match; thus, supplying NULL for s_1, s_2, and s_3 will iterate over all network views in p_zone.

Once a network view is added to a zone with Libnucnet__Zone__updateNetView(), the zone owns the view. This means that the memory for the view will be freed when the zone is freed. If a network view has not been added to a zone, it is the user's responsibility to free the memory with Libnucnet__NetView__free().