NuWro architecture - Programmers Guide

Cezary Juszczak

27 Nov 2017

Basis assumptions

1. C++ Language
2. Few external dependencies
3. Plain text human readable input
4. output in root tree format (one branch of events)
5. output events are objects (data + methods like `q2()`, `n()` facilitating analysis).
6. Assume only moderate programing skills from Physicists
7. No particular programming style forced
8. Use classes and operator overloading to make coding pleasant
9. Natural system of units $c = \hbar = 1$ (and MeV$= 1$) so c and $\hbar$ never appear in the code.
10. PDG particle numbering scheme
11. Based mainly on theoretical models *with very few fits*
12. Modular design, parts of the code should not depend on each other if it is not necessary

The input (parameters and data files)

The input is specified by means of a text file (default `params.txt`).

```
nuwro -i myparams.txt
```

With simple one parameter per line syntax:

```
parameter1 = value_of_parameter1
paramater2 = value of multi word parameter2
```

Multi-line parameters need `+=` operator to append more than one line:

```
parameter3 =  first line of text
parameter3 += second line of text
parameter3 += third line of text
```

The `@` sign includes other files.

```
@target/my_target_parameters.txt
@beam/my_beam_parameters.txt
```

The comment lines begin with the `#`

```
# This is a comment
```

Command line parameters

- Any number of input parameters can be given in command line:

```
nuwro -p "par1=val1" -p  "par2=val2" ...
```

- For multi-line parameters each line is entered separately

```
nuwro -p "par=line1" -p  "par+=line2" -p  "par+=line3"
```

- Values given in the command line override those from input file.
- One can also specify the input and output file names:

```
nuwro -i "myparams.txt" -o "output-file-name.root"
```

- Some other minor options are only by the online interface.

The `params` class and the `par` object

All the input parameters are collected in one structure.

```
class params
{ public:
int random_seed;
int number_of_events;
...
} par; // all the parameters
```

- Most of the NuWro objects are initialized based on `par`.
- Copy of `par` is included in each output `event`.
- The `params` class is generated from `params.xml` file.
- `params.xml` contains all the information about parameters:
  - type
  - name
  - default value
  - list of possible values (where applicable)
  - description
  - html input type (used by `nuwro-online`)

The output

- The output is a `root` file (default name `eventsout.root`).
- The file contains a `TTree` named `eventsout`.
- The `eventsout` tree has only one branch named `e`.
- The `e` branch contains objects of class `event`

```
class event {
...
    params par;    // copy of input parameters
    flags  flag;   // bool cc, nc, qel, dis, res, coh, mec
    int dyn;                // code of dynamical channel
    vector<particle> in;    // incoming particle
    vector<particle> temp; // DIS temporary particles (by Py
    vector<particle> out;  // particles before FSI
    vector<particle> post; // particles after leaving nuclue
    double weight;    // weight = cross section per nucleon
    ....
};
```

The `event` structure

The `event` object contains the following information

- `par` - copy of input parameters
- `flag` - boolean flags useful for output filtering:
  - `cc`, `nc` – charged/neutral current events
  - `qel`, `res`, `dis`, `coh`, `mec` - indicate dynamical channel
- four `vectors` of `particles`:
  - `in` - two incoming particles: `in[0]` is neutrino, `in[1]` is nucleon
  - `temp` - temporaries coming from Pythia in DIS
  - `out` - leaving primary vertex (before FSI); `out[0]` is lepton
  - `post` - leaving the nucleous (after FSI); `post[0]` is lepton
- `dyn` - integer code of the dynamical channel
- `pr`, `nr` - number of protons/neutrons in the residual nucleus
- `r` - position of the event inside the detector
- `weight` - event cross section* (used internally),

The `event` object has also many methods acting on it that are useful in the analysis.

The `event` methods

The `event` object has many methods useful during the output analysis (you can also write your own custom methods if needed):

- `nu()`, `N0()` - initial neutrino/nucleon
- `E()` - initial neutrino energy
- `q()`,`q0()`,`qv()` - fourmomentum/energy/momentum transfer
- `q2()` - momentum transfer squared
- `s()` - s - variable
- `costheta()` - cos theta lab
- `charge(r)` - total charge: 0 - initial, 1 - before fsi, 2 - after fsi
- `W()` - invariant mass (before FSI, all particles except the rescattered lepton)
- `n()`- number of particles after primary vertex (before FSI)
- `f()`- number of particles leaving nucleus
- and much more (for details see `event1.h` header file)

The particle structure

Each particle contains the following information

- t, x, y, z - energy and momentum coefficients
- _mass - particle mass
- pdg - particle pdg code
- r - position inside the nucleus
- travelled - distance from creation
- mother - index of mother particle in the all vector
- endproc - id of process that destroyed particle

Particle methods make coding/analysis much easier:

- E(), Ek() total/kinetic energy of the particle
- m(), mass(), mass2(), charge() - mass/mass squared/charge
- p(), p4()- the momentum as a 3-vector/ the fourmomentum
- v(), v2() - velocity as a 3-vector/ velocity squared
- momentum(), momentum2() - momentum value/square
- set_mass(m) - set particle mass and adjust energy
- set_energy(E)- set energy and adjust momentum
- set_momentum(p) - set momentum vector and adjust energy
- decay (& p1, & p2) - decay particle to p1, p2
- Ek_in_frame(v) - kinetic energy in frame moving with velocity v

Three- and four- vectors

- Three vectors are representad by class `vec`:
  - `x`, `y`, `z` - coordinates
  - `+ - * / = += -= *= /=` - operators
  - `sqr`, `length`, `dir`, `norm`, `normalize` - methods
  - `vecprod(a,b)`, `angle(a,b)`, `cos(a,b)` - helper functions
  - many others
- Four-vectors are represented by class `vect`:
  - `x`, `y`, `z`, `t` - coordinates
  - `+ - * = += -=` - operators
  - `boost(v)` - boost to a frame moving with velocity v
  - `length()` - length of the spatial part
  - `v()` - velocity of particle with momentum `(x,y,z)` and energy `t`
- `particle` is a subclass of `vect` so you can `boost` particles too.
- You can also perform four-momentum calculations directly on particles:

```
particle nu,lepton;
vect sum=nu+lepton, q=nu-lepton;
double t=q*q, s=sum*sum;
```

What it means?

- All of the goodies are exported and available during the analysis of the output. Many plots with just one command:

```
treeout->Draw("out.mass()"); //masses of outgoing partiles.
treeout->Draw("n()"); // number of outgoing partiles
```

- number of π^0 produced in deep inelastic charged current events:

```
treeout->Draw("nof(111)","flag.dis*flag.cc");
```

- momenta of outgoing π^0s:

```
treeout->Draw("out.momentum()","out.pdg==111");
```

- places where the interactions took place in the cascade code:

```
treeout->Draw("all.r.x:all.r.y:all.r.z");
```

- shape of the $d\sigma/dq^2$ differential cross section for DIS

```
treeout->Draw("q2()","flag.dis");
```

- More complicated scripts are also much shorter and robust due this object oriented approach.

Inside the black box

The generator must follow the following path:

- Initialize itself based on the input parameters
  - Initialize the beam (read data files if necessary)
  - Initialize the target (read detector geometry if needed)
  - Initialize some data tables (e.g. for pion scattering, or Spectral Function)
- Run the test events repeating the following
  - Create new event named `e`
  - Create the beam particle and save into `e.in[0]`
  - Create the target **`nucleus`**
    - if(detector geometry) Get random place on neutrino path and accept based on density times track length inside detector. If not accepted replace neutrino with another one and try again. After accepting a place, choose isotope based on material composition.
    - Create nucleus of the chosen isotope
  - Obtain the target nucleon and put into `e.in[1]`
  - Choose the dynamical channel
  - Run the corresponding code that fills the rest of the event and calculates the event weight
  - add the weight to the sum of channel weights.
  - destroy event `e`

Inside the black box (continued)

- For each channel: cross section = the average event weight.
- Report channel cross sections, and calculate number of events to be generated for each channel.
- Generate real events in temporary files – for each channel repeat until desired number of real events is saved in a file.
  - Create new event named `e`
  - Create the beam particle and save into `e.in[0]`
  - Create the target **`nucleus`** (the same way as for test events)
  - Obtain the target nucleon and put into `e.in[1]`
  - Run the code for current dynamical channel that fills the `temp` and `out` vectors calculates the event weight.
  - if weight exceeds the so far maximum, “delete” appropriate number of events from current file and increase the maximum
  - accept event based on its weight and maximal weight so far for that channel.
    - if accepted run the FSI code and save event to file
  - destroy event `e`
- Copy events from temporary files to output file in random order, skip “deleted” events from the beginning of each temporary file.

The code for dynamical Channels

The code for dynamical channels is/was usually written by physicists not programmers. It is up to a channel developer which programming style they use.

- Each dynamical channel code has the form of a function with the following parameters:

  ```
  double f(params &p, event& e, nucleus &t, bool nc);
  ```
- The arguments of this function are basically everything which is needed to generate event of a given kind.
- Some global utilities like form factor library or kinematics generators can be also be used.
- the function should:
  - create outgoing particles for given incoming neutrino `e.in[0]` and nucleon `e.in[1]`, and put them to `e.out`.
  - Calculate the differential cross section and store it to `e.weight`.
  - return `e.weight`.
- The average `weight` must be equal to the total cross section for any selection of the events in this channel.

The FSI code

The code for the FSI is called `kaskada`.

- It takes particles from `e.out` and propagates them through the nuclear matter taking into account NN and πN interactions.
- In fact it can be separated from NuWro to investigate nuclear transparency and nucleus pion interactions
- FSI never changes the `weight` of the event so it has no influence on the inclusive cross section.
- It propagates nucleons and pions in small steps and proportionally to nuclear density at given place and the interaction cross sections performs interactions.
- all intermediate particles are logged in `e.all`
- particles leaving nucleus are placed in `e.post`

Beam interface

1. By definition the beam creates unweighted (equally probable) particles
2. Exception is made for DIS events where biased beam is used:
    - Energy profile is multiplied by energy
    - but probability of accepting event is divided by neutrino energy

Projectiles (neutrinos) are produced by method `shoot(bool biased)` of an abstract class `beam` which has five implementations in descendant classes:

- single flavor beam (particle PDG, direction, energy histogram)
    - plain text histogram originally,
    - root histograms implemented by Luck Pickering
- multi flavor beam (mixed beam) is really a weighted mixture of single flavor beams having the same direction but different particles and energy profiles
- realistic beam: reads a list of weighted neutrinos coming from beam simulation (different flavors, places of origin, momenta) from files. Select neutrinos from the list with probability proportional the weight (times energy if biased).

The `target` class

The role of the `target` class is to produce a `nucleus` which the neutrino is going to scatter on. There are following implementations:

- The original single isotope target. Creates `nucleus` based on: number of protons, and neutrons. Fermi momentum and binding energy can also be given.
- Mixture is really a weighted mixture of the above.
- Full detector geometry. This is read from a root file provided by the GEANT simulation software. Specific box shaped region of interest can be defined.
  - The generated isotope must lie on neutrino trajectory (in case of realistic neutrino flux) and inside the region of interest.
  - If neutrino fails to hit region of interest or the chosen place is not accepted, another neutrino is asked from the beam.
  - Probability of accepting a place grows with density of material and length of the track inside the region of interest.
  - Accepted place yields an isotope based on its contribution to the mass of the material.

The `nucleus` class

The nucleus class provides the `get_nucleon()` method that creates the initial nucleon that is hit by neutrino and is stored in `e.in[1]`. This nucleon is positioned according to nuclear density and has momentum genereted according to Local Fermi Gas or Global Fermi Gas model. For certain isotopes the Spectral function model is also available. Other methods of this class like:

```
double density (double r);
void remove_nucleon (particle);
void insert_nucleon (particle);
pauli_blocking (particle);
```

are heavily used in the FSI code.
There are two nucleus models implemented in NuWro

- Uniform density model (ball).
- Realistic density model: density profiles implemented for most of the existing isotopes. In some cases more than one fit is available.

Main Statistical Assumption

The events for a given dynamical channel should be generated in such a way that

- the mean weight of events after any selection be equal to the total cross section for analogous selection of events.
- Cross section for non biased events (w_i = weight of i-th event):

$$\sigma = \sum_{i=1}^{n} w_i/n$$

- Non biased events are accepted with probability $\sim w_i$.
- For DIS the events are biased with neutrino energy (on creation) therefore we accept them with probability proportional to cross section/energy (thus avoid the high energy tail problem with efficiency)
- For biased events, where b_i = bias = enhancement in probability:

$$\sigma = \sum_{i=1}^{n} \frac{w_i}{k_i} / \sum_{i=1}^{n} \frac{1}{k_i}$$

- Biased events are accepted with probability $\sim w_i/k_i$.

Efficiency considerations

- The efficiency of generating reals events is the percentage of events that get accepted and stored in file. It is equal to average probability of accepting event:

$$\text{Efficiency}_{nonbiased} = \frac{\text{mean}(w_i)}{\max(w_i)}$$

$$\text{Efficiency}_{biased} = \frac{\text{mean}(w_i/k_i)}{\max(w_i/k_i)}$$

- This is obvious that it might be benefitial to scale the bias of the beam with the shape of the total cross section.