*Grammar parser-based solution for the description of the computational graph within GNA framework[a]

Tsegelnik Nikita[b], Gonchar Maxim, Treskov Konstantin

Joint Institute for Nuclear Research

July 6, 2021

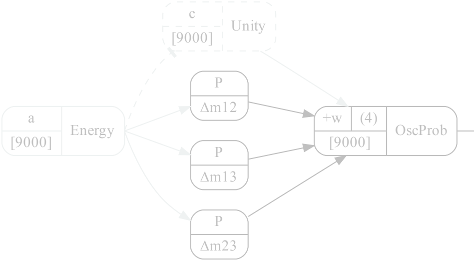
[a] *The research is supported by the Russian Science Foundation grant 21-42-00023*

[b] tsegelnik@jinr.ru

GNA overview

- GNA — **G**lobal **N**eutrino **A**nalysis — high performance fitter based on data flow scheme:
  - Laziness
  - Caching

- Workflow — compile once and build at runtime:
  - Frontend: Python
  - Backend: GNA core (C++)
  - *Potential* support for other backends: *TensorFlow*, ...



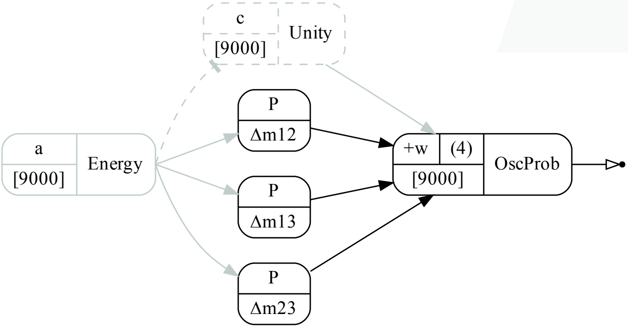
- Representation — everything is accessible and annotated:
  - Print variables
  - Plot graph structure
  - Plot graph data

https://astronu.jinr.ru/wiki/index.php/GNA

Example: neutrino oscillation probability

```python
E = C.Points(np.arange(1.0, 10.0, 0.001))
with ns:
  oscprob = C.OscProb3(from_nu, to_nu)
  unity = C.FillLike(1)
  ws = C.WeightedSum(weights, labels)

  E >> unity.fill
  E >> oscprob.comp12
  E >> oscprob.comp13
  E >> oscprob.comp23
  unity          >> ws.sum.comp0
  oscprob.comp12 >> ws.sum.item12
  oscprob.comp13 >> ws.sum.item13
  oscprob.comp23 >> ws.sum.item23
```



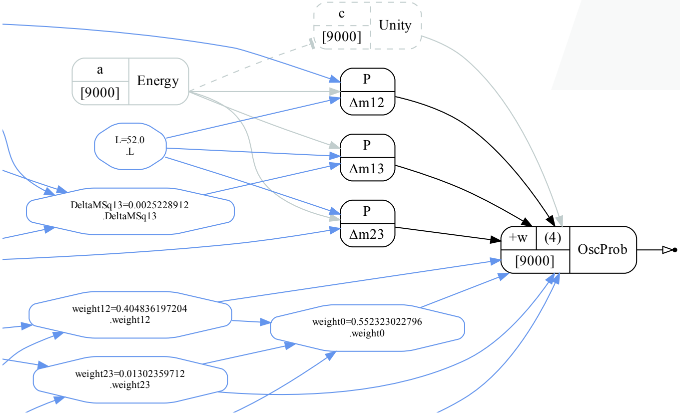
$$P_{\text{sur}} = P_0 + \sum_i \omega_i(\theta_{12}, \theta_{13}, \theta_{23}) \cos\left(C \frac{L \Delta m_i^2}{E}\right)$$

- Nodes are only evaluated if they have changed
- Each nodes result is cached and reused

Example: neutrino oscillation probability

```
E = C.Points(np.arange(1.0, 10.0, 0.001))
with ns:
  oscprob = C.OscProb3(from_nu, to_nu)
  unity = C.FillLike(1)
  ws  = C.WeightedSum(weights, labels)

  E >> unity.fill
  E >> oscprob.comp12
  E >> oscprob.comp13
  E >> oscprob.comp23
  unity          >> ws.sum.comp0
  oscprob.comp12 >> ws.sum.item12
  oscprob.comp13 >> ws.sum.item13
  oscprob.comp23 >> ws.sum.item23
```



- Variables are maintained in recursive namespace
- Nodes do not own variables they depend upon

How to scale?

Bundles

- Bundles were created in order to facilitate making small computational graphs and scale them based on a simple configuration

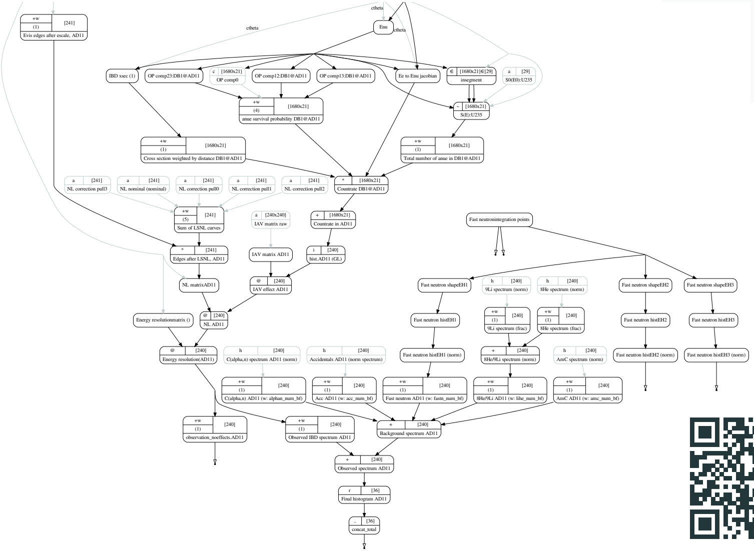
- **Bundles are able**:
  - Read a configuration dictionary, which contain:
    - Bundle name and version number, telling GNA which bundle to load to read the configuration
    - Multidimensional index, defining how the bundle should replicate the parameters and/or nodes
    - Other configuration, specific to a particular bundle
  - Access environment and define a set of variables, required by the transformations chain it builds
  - Create and bind a set of nodes. Register the open inputs and outputs so other bundles may access them
  - Require inputs and outputs, provided by the other transformations and bind them

Simplified view

- Single detector
- Single reactor
- Single isotope
- ∼ 1/10th of the full graph

Full graph

- Current configuration:
  - ∼ 500 nodes
  - ∼ 1000 edges
- Depending on structure may be 2500 nodes



http://gna.pages.jinr.ru/gna/gallery.html

How to set?

GNA expressions

- **Goal:**
  - Math-like expression $\longrightarrow$ graph
- **Solution:**
  - DSL — **D**omain-**S**pecific **L**anguage
- **Advantages:**
  - One expression for *various* backends
  - Some natural *scaling* rule (*indices*)
  - Graph preprocessing

- **GNA DSL objects:**
  - Transformations $\longleftrightarrow$ *vector* objects
  - Variables $\longleftrightarrow$ *scalar* objects
  - Indices, arithmetic operations $(+, -, *, ...)$, brackets, ...

Experiment with reactor antineutrinos: formula

Partial expression

```
eres[d]|
 lsnl[d]|
  iav[d]|
   integral2d|
    sum[r]|
     baselineweight[r,d]*
     ibd_xsec(enu(), ctheta())*
     jacobian(enu(), ee(), ctheta())
     sum[i](
      power_livetime_factor[d,r,i])*
      anuspec[i](enu() )*
     sum[c]|
      pmns[c]*oscprob[c,d,r](enu())
```

Energy resolution

Partial equation

$$\vec{N}_d = C_{\text{eres}} \times C_{\text{lsnl}} \times C_{\text{IAV}} \times \int d\cos\theta \int dE_{\text{vis}} \sum_r 1/(4\pi L_{dr}^2)\, d\sigma(E_\nu, \cos\theta)/d\cos\theta\, dE_\nu/dE_{\text{vis}} \sum_i (P_{dri} S_i(E_\nu)) \sum_c \omega_c P_c(E_\nu, L_{dr})$$

$a|\,b|\,c*d = a(b(c*d)))$

Parser implementation

Current GNA implementation:

- Pure Python:
  - Cumbersome
  - Excessiveness
  - Difficulty in improving
  - Lack of abstractness (made *in* and *for* GNA framework *only*)

New implementation:

- Designed as a completely stand-alone module:
  - Parser based on *Lark-Parser*:
    - Fast and strict parser algorithm LALR(1)
    - Small grammar file **(in this talk $\sim 50 - 100$ lines)**
    - Easy to modify
    - Caching
  - Graph features based on *PyGraphviz*
  - *Suitable for different backends*
  - Separation of roles: parser, builders, matchers, data classes, tests, ...
  - Representation: trees, graphs, text view, ...

`https://lark-parser.readthedocs.io/en/latest/index.html`

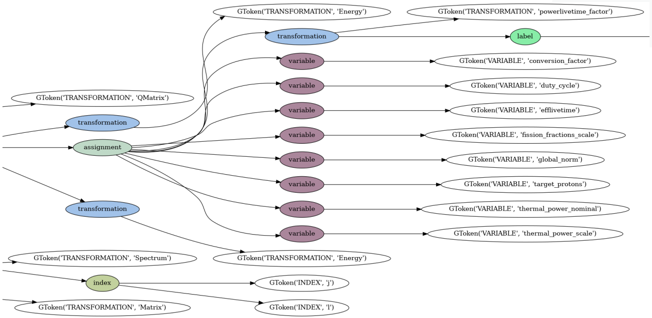
Parsing tree and pattern matching

Data to parse

```
Object = ns1.ns2.norms[k] *
 # comment lines with '#'
 # alphan_rate*alphan_rate_norm*
  Transf_1 + Transf_2 | QMatrix @
  Energy * (conversion_factor *
  duty_cycle * efflivetime *
  fission_fractions_scale *
  global_norm * target_protons *
  thermal_power_nominal *
  thermal_power_scale) +
  Spectrum[j,l] @ Matrix #comments
```

- This is a *pattern matching* feature!
- The formula has no physical meaning!

Parsing tree for the red sequence



Part of the library with patterns

```
powerlivetime_factor:
  expr: 'conversion_factor*duty_cycle*efflivetime*
    fission_fractions_scale*global_norm*target_protons*
    thermal_power_nominal*thermal_power_scale'
    label: 'Power/Livetime/Mass factor, nominal'
```

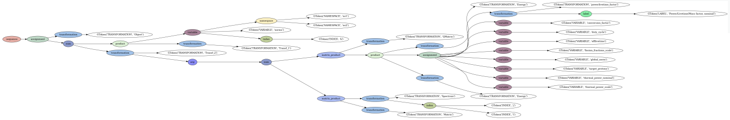
Building graph

Data to parse

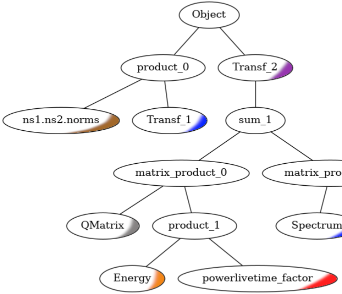
```
Object = ns1.ns2.norms[k] *
 # comment lines with '#'
 # alphan_rate*alphan_rate_norm*
  Transf_1 + Transf_2 | QMatrix @
  Energy * (conversion_factor *
  duty_cycle * efflivetime *
  fission_fractions_scale *
  global_norm * target_protons *
  thermal_power_nominal *
  thermal_power_scale) +
  Spectrum[j,l] @ Matrix #comments
```

- *Simple graph supporting scaling!*
- *product_0*, *sum_0*, … — automatically generated nodes for operations '+', '*', …

Full parsing tree



Final graph



Conclusions

- **Conclusions:**
  - The GNA DSL has been developed that is *suitable for various* (**potential**!) *backends* and naturally enables a *scalability* of physical models
  - A similar solution is applicable for *any* data flow scheme
  - Can be developed completely *separately* from the main project (*this was done in the GNA*)

- **Current status:**
  - Merging the Parser module with GNA framework
  - Testing on the real physical model

Thank you for your attention!

Grammar parser-based solution for the description of the computational graph within GNA framework[a]

Tsegelnik Nikita[b], Gonchar Maxim, Treskov Konstantin

Joint Institute for Nuclear Research

July 6, 2021

[a] *The research is supported by the Russian Science Foundation grant 21-42-00023*

[b] tsegelnik@jinr.ru