Declarative analysis in "Troitsk nu-mass" experiment

Alexander Nozik

Institute for Nuclear Research RAS, prospekt 60-letiya Oktyabrya 7a, Moscow 117312
Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russian Federation

Abstract. The modern scientific data processing is not only a collection of powerful algorithms but also a whole infrastructure of facilities for data reading, processing, and results output. As the amount of data grows, so grows a need for automation of the process. The proper automation requires not only improvement of existing frameworks, but also a search of new ways to organize data processing in a way it could be automated and parallelized. DataForge experimental framework solves some problems by making the analysis configuration declarative instead of imperative. In this article, we present a limited demonstration of the idea applied to "Troitsk nu-mass" experiment in the search for sterile neutrino.

1. Introduction

The DataForge is an experimental data processing framework designed to leverage declarative configuration, lazy computations, and automatic task parallelization and output management. In the previous article [1], we discussed major concepts behind the design of the DataForge framework prototype. While the scientific community was quite responsive to ideas, presented in the article and in project documentation, it soon became obvious that it is very hard to discuss pros and cons of the approach without actual example. The DataForge prototype was developed and implemented at "Troitsk nu-mass" experiment ([2–4]), so we will describe current analysis layout of DataForge-based analysis for this experiment.

2. Basic concepts

In order to achieve the required automation level, DataForge requires for data, computation environment and task definition to be defined in a declarative way, which allows simple verification and comparison. It is possible to use framework tools for the classic scripting approach, but in this case, it loses its advantage compared to script-based frameworks. The paradigm could be summarized in a few basic concepts:

- **Data**: A data from experiment or simulation. According to DataForge paradigm, data is always immutable, meaning it could be neither created, nor mutated. Contrary to the scripting approach, intermediate analysis results are automatically managed and cached by the framework but in general not accessible by a user. This restriction solves the major problem of an invalid intermediate state which plagues the analysis everywhere. An additional benefit is that intermediate state, which is not required for the final result is not calculated, automatically saving computation time. The state which is reused could be automatically cached by the system because the results are uniquely determined by meta-data.

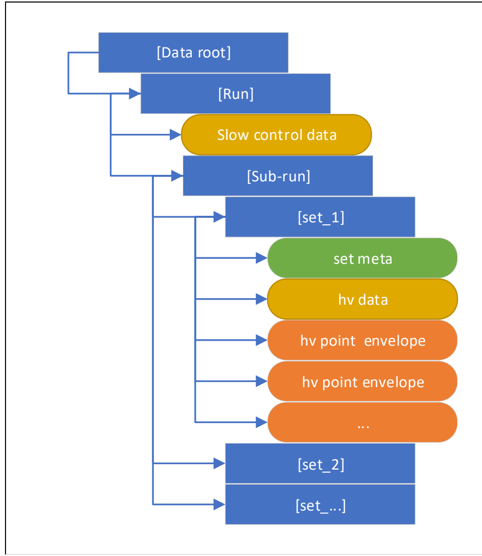

Figure 1. Raw data tree structure for "Troitsk nu-mass" experiment.

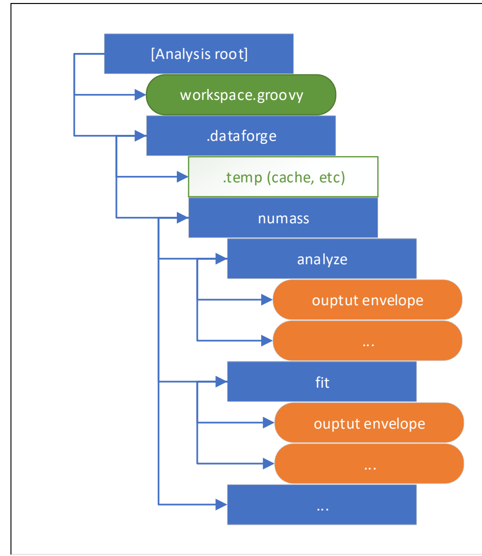

Figure 2. Analysis and result directory structure.

- **Meta-data**: Configuration data, which partially comes with data and partially represents user analysis configuration. Meta-data has two important features: it could be easily validated and merged. Validation means that one could clearly define analysis by its meta-data configuration and check if a new result is obtained with a different configuration. This could not be done with scripts, because in the script there is an infinite number of ways to define the same process. Merging allows making structured configuration analysis, meaning that some parameters could be defined for the whole data set instead of a single piece of data. Also one could override general analysis rules for specific data pieces.
- **Context**: The environment for the computation. The context is isolated from the system environment and could be defined in a reproducible way inside the framework. The context could be customized by plugins adding new functionality and customizing output for analysis, but context could not be changed during the analysis itself, so it is not possible to introduce a shared mutable state for analysis tasks.

3. Input data

Since DataForge relies heavily on meta-data coming with data, it is convenient to use a container that will contain both data and meta-data. This container is called an envelope and from the programmatical point of view is just a class with two fields: `data` which is a general binary and a `meta` which is a meta-data tree. This container could be serialized in or deserialized from a file using various meta-data text representations (like `JSON`, `XML` or a binary representation). The file contains an additional short binary header to save information about meta length and meta encoding. This allows reading and parsing the meta-data block from the file without loading the whole file into memory. The text format is preferred for meta-data because it allows interpreting the result by just opening a file in a text editor and reading its header.

The "Troitsk nu-mass" run data is represented in a following way (see Fig. 1):

- The data tree root is a regular file system directory.
- The directories under the root represent experimental runs and named after year and month of the data acquisition start like **2017_05**.

- The run could have an inner structure represented by sub-directories containing either so-called tritium data (actual measurements with the tritium in the source) or calibration data.
- The directories with data contain numbered measurement directories called **set** (like **set_1**, **set_2**, etc.), each containing a single high voltage scan as described in [3].
- Each set contains a number of individual measurement points for fixed high-voltage values, a file for high-voltage monitoring measurements and a file with set meta-data containing information about the set as a whole (start time, readout system state, etc.). All files are envelopes, meaning they contain a meta header alongside the binary or textual data. In the case of points, the data part of the envelope is represented by a Protobuf ([5]) encoded message with optional zip compression (the compression is defined by the flag at point meta-data).
- Slow control data for the whole run is usually placed in the run root directory in a DataForge envelope format with a simple ASCII text table.

This structure is obviously very well suited to be used with DataForge meta-data processor paradigm, and in this case has an advantage compared to frequently used data storage systems like ROOT-IO ([6]) or HDF5 ([7]). It allows to copy small portions of the data from one computer to another and easily load individual points into memory, allowing analysis program to keep low memory footprint without sacrificing performance. Also, it delegates problems like parallel data access, data integrity checks, etc to the modern file system instead of implementing them inside the framework. One should note that DataForge itself does not impose any restrictions on binary input or output format. One could use one of main-stream formats like Protobuf, compressed text or even ROOT IO binary. The specific type of data is inferred by the reader based on meta-data.

The analysis configuration is separated from the data directory tree and usually have a single configuration file for the whole data run and `.dataforge` directory for output and caches in the same directory (Fig. 2). The configuration file is basically meta-data itself and could be represented as `JSON`, `XML` or Groovy ([8]) script that generates such configuration in more compact way. One should note, that while Groovy script is imperative itself, it is used only to generate declarative configuration, it does not interact directly with the framework. The configuration defines a context and its plugins, data root reference and one or several target configurations (meta-data for specific data set). A typical target configuration is presented in the following code block.

```
//define a configuration for a single sub-run
target("fill_2"){
  data{   //Configuration for data selection
    include(pattern: "Fill_2.set_2.")
  }
  analyzer{ // Configuration for amplitude spectrum analyzer.
    t0(crFraction: 0.05, min: 1.5e4)
    window(lo:500, up:3000)
  }
  //Merge flag. If this line is commented, all sets are evaluated individually
  // instead of merging them in one table.
  merge(mergeName: "fill_2")
  fit{  // Fit task parameters
    //Define the model to fit data against
    model(modelName: "sterile"){
```

```
      // Parameter for model, including interaction in source and spectrometer
    }
    // Starting point for fit and errors
    params{
      N(value: 8e5, err: 1e4, lower:0)
      bkg(value: 3, err:0.1)
      //other parameters
    }
    // Custom fit stages
    stage(freePars: ["N", "bkg", "E0"])
    stage(freePars: ["N", "bkg", "E0", "U2", "trap"])
  }
  // Configuration for sterile neutrino mass scan
  scan(masses: [0.5, 1.0, 1.5, 2.0, 2.5, 3.0])
}
```

The syntax for configuration is following: `node(a: 2, b: "string){<children>}` is transformed into meta-data node with name `node`, leafs `a` and `b` with appropriate values and inside of `{<children>}` block defines child-nodes.

4. Task evaluation

The analysis logic is defined by tasks. A task is a lazy computation which is performed on a data tree and in turn returns a new data tree (it does not modify anything in the initial data). The task could depend on another task or on initial workspace data, but not on external files, which means that the whole analysis is self-contained and could be viewed as a directed graph of computations leading from initial data to the result, the user requested. It is not possible to describe the whole task structure for "Troitsk nu-mass" analysis (about 15 different tasks), so we will limit ourselves with a short explanation of task inner workings.

In order to compute something, use must define or get a task (pre-loaded tasks are accessed by their name) and the meta-data (target). Task evaluation is in general done in three steps:

(i) **Resolve task dependency tree**. Each task can resolve its own dependencies based on the input meta. The dependency is either data or the result of another task (in this case task passes configuration to its dependency, modifying it if needed). At this stage, no computations are done and result meta-data is not generated.

(ii) **Compute lazy result**. Tasks resolve their actual input based on input data and its meta. Then each task calculates its output node (it is not a single element, but in general a tree-like object). As a result, user gets a `DataNode` object which contains all requested results. Ideally, at this stage, no actual computations are done. Inside each element of `DataNode` resides a computation graph that connects initial data elements to the result. Some fractions of this tree could be replaced by cached elements in case the computation with a given meta was done before.

(iii) **Running the computation graph**. The actual computation starts only when the user tries to get the result from `DataNode`. One can select only a fraction of `DataNode` and it will request only the data which is actually used in the result and only those computations which are required. All actions are automatically parallelized so if some branches of the computation graph do not intersect they are automatically computed in parallel, without additional effort from the user. One does not need for all elements in a task to finish in order to start independent elements in the next task. It allows avoiding computation bottlenecks. For example, if you have a parallel computation graph, where one single data element takes

a long time to calculate, the system won't stop everything for this specific element, it will proceed with other tasks, which do not depend on this element.

It is possible that some tasks could not be resolved in a lazy way. In this case, task is called **terminal** and its results are calculated immediately at the second stage.

The DataForge supports basic task templates like

- Pipeline, where all elements of `DataNode` are evaluated independently in the same way (with possible customization for each element based on its meta).
- Join, where one waits for several elements from data or dependent tasks and then combines them.
- Split, where each incoming elements is transformed into results with fixed types.

The only tricky part is to implement tasks so the result meta is calculated without triggering the computation itself. As a compact example of pipe task we can look into `analyze` task, which computes the amplitude spectrum from the raw data (the tasks are written in Kotlin - [9, 10]):

```
val analyzeTask = task("analyze") {
    model { meta ->
        dependsOn(selectTask, meta)
        configure {
            "analyzer" to meta.getMetaOrEmpty("analyzer")
        }
    }
    pipe<NumassSet, Table> { set ->
        val res = SmartAnalyzer().analyzeSet(set, meta.getMeta("analyzer"))
        val outputMeta = meta.builder.putNode("data", set.meta)
        context.output.render(res, stage = "numass.analyze", name = name, meta = outputMeta)
        return@pipe res
    }
}
```

Here we can see two blocks. The model block defines the dependencies of the task and takes all necessary parameters from external configuration (it could also validate those parameters and throw an error during the configuration phase). And the second `pipe` block uses a template to transform the data for a single input element in a type-safe way, without thinking about data structure and parallelization. Output renderer requires additional explanation.

5. Directory-based output

An important feature of the framework is the customizable asynchronous output. The general way to send something to output is to use `OutputManager` plugin which allows creating output channels using two identifiers: **stage**(designates the process that produces the output) and **name**(designates the specific data pieces for which the output is produced).

For example, in "Troitsk nu-mass" we by default use the so-called directory-based output. The working directory. In this case the **stage** parameter is transformed into a directory name and **name** is used as a file name. For example **analyze** task sends its output to `.dataforge/numass/analyze` directory and file name corresponds to the name of the input set. The directory-based output uses a different output format depending on the type of object which has been passed to it. In this particular case, the table is transformed into a human-readable ASCII text table, wrapped in an envelope with provided meta, so one knows which

meta was used to produce this result without additional analysis journal. The layout of the output directory is demonstrated in Fig. 2.

The output could also be forked. In this case, the same object is sent simultaneously to two different output renderers. During the analysis, we usually fork the output and send the result both to the output directory and to graphical interface to be presented as a plot or table. All output actions are done asynchronously, so they do not affect the primary computation.

6. Conclusion

The DataForge framework presents a way to implement scientific data processing in a declarative way rather than using the imperative (scripting approach). It allows for solving some problems of modern data processing like automatic parallelization and validation of intermediate data. It also allows excluding the calculation of intermediate results that are not required for a specific task and establishes a robust way to treat input and output. The major drawback is that tasks could take only simple meta-data trees as a parameter. Meaning that one needs to create a new task or pre-compiled function for anything else.

The current DataForge prototype code could be found at `https://bitbucket.org/Altavir/dataforge/src/dev`. Limited documentation is available in [11] The modules, developed for "Troitsk nu-mass" experiment are located at `https://bitbucket.org/Altavir/numass/src/dev`.

Currently, there is an ongoing work to rewrite the core functionality of the framework from scratch using Kotlin language: `https://github.com/altavir/dataforge-core`.

References

[1] Alexander Nozik. "DataForge: Modular platform for data storage and analysis". In: *EPJ Web Conf.* 177 (2018), p. 05003. DOI: 10.1051/epjconf/201817705003. URL: https://doi.org/10.1051/epjconf/201817705003.

[2] V. N. Aseev et al. "An upper limit on electron antineutrino mass from Troitsk experiment". In: *Phys. Rev.* D84 (2011), p. 112003. DOI: 10.1103/PhysRevD.84.112003. arXiv: 1108.5034 [hep-ex].

[3] A. I. Belesev et al. "The search for an additional neutrino mass eigenstate in the 2–100 eV region from 'Troitsk nu-mass' data: a detailed analysis". In: *J. Phys.* G41 (2014), p. 015001. DOI: 10.1088/0954-3899/41/1/015001. arXiv: 1307.5687 [hep-ex].

[4] D. N. Abdurashitov et al. "The current status of "Troitsk nu-mass" experiment in search for sterile neutrino". In: *JINST* 10.10 (2015), T10005. DOI: 10.1088/1748-0221/10/10/T10005. arXiv: 1504.00544 [physics.ins-det].

[5] *Protocol Buffers*. https://developers.google.com/protocol-buffers/. [Online; accessed 10-May-2019]. 2019.

[6] *ROOT-IO*. https://root.cern.ch/root-io. [Online; accessed 10-May-2019]. 2019.

[7] *HDF5*. https://www.hdfgroup.org/solutions/hdf5/. [Online; accessed 10-May-2019]. 2019.

[8] *Groovy language*. http://groovy-lang.org/. [Online; accessed 10-May-2019]. 2019.

[9] *Kotlin language*. https://kotlinlang.org/. [Online; accessed 10-May-2019]. 2019.

[10] D. Jemerov and S. Isakova. *Kotlin in Action*. Manning Publications Company, 2016. ISBN: 9781617293290. URL: https://books.google.ru/books?id=qtcIkAEACAAJ.

[11] *DataForge homepage*. http://npm.mipt.ru/dataforge/. [Online; accessed 10-May-2019]. 2019.