CASE STUDY 25

Developing an Open, Modular Simulation Framework for Nuclear Fuel Cycle Analysis

KATHRYN HUFF

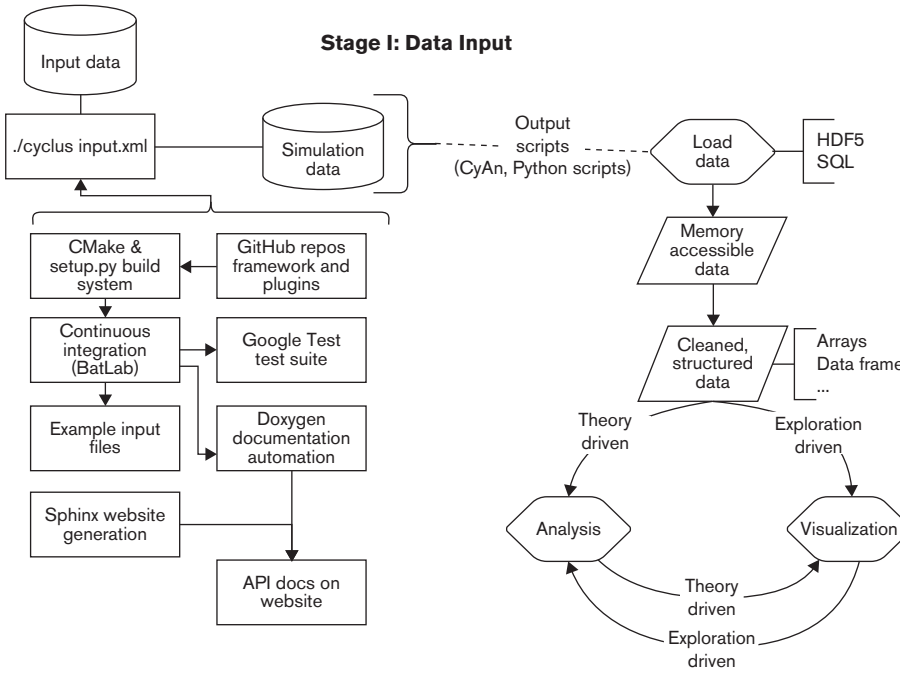
My name is Kathryn (Katy) Huff, and I am a Nuclear Science and Security Consortium postdoctoral scholar in the Nuclear Engineering Department and a Data Science Fellow with the Berkeley Institute for Data Science. My research includes computational nuclear fuel cycle analysis and computational simulation of coupled, transient, nuclear reactor physics.

Improving the safety and sustainability of nuclear power requires improved nuclear reactor designs, fuel cycle strategies, and waste disposal concepts. The systems are sufficiently complex that breakthrough advancements may emerge when modern data methodologies are applied to their simulation. In particular, faithful assessments of potential nuclear fuel cycles require dynamic, discrete facility, discrete material simulations of the mining, milling, transmutation, reprocessing, and disposal of nuclear materials as well as the production of energy and movement of capital.

This case study is an overview of the workflow behind the Cyclus nuclear fuel cycle simulation framework—a tool for exactly that kind of modeling, simulation, and analysis. The workflow described used to create a software tool that other nuclear engineers can use easily, modify quickly, and contribute to when they need to customize behavior or model a different technology.

WORKFLOW

A group of geographically dispersed researchers (graduate students and professors) and I collectively develop and maintain an agent-based



simulation framework called Cyclus. We also develop and maintain plug-in models representing the agents in the simulation. These agents model the mining, milling, fabrication, transmutation, and disposal of nuclear material in the *nuclear fuel cycle*.

Cyclus is a C++ code base. The configuration and build system is created from a combination of Python and CMake (a crossplatform automatic makefile configuration system) and supports both Linux and MacOS operating systems. Our input validation library accepts either xml or json input files. The simulator accordingly conducts a simulation which generates an output database in either SQL or HDF5 format which can be traversed by a separately developed graphical user interface.

As we develop this software, we rely on a number of best practices to ensure reproducibility.

When a large-scale enhancement is needed, a Cyclus Enhancement Proposal (CEP) is proposed and discussed among the development team. Smaller enhancements are discussed as issues in GitHub. Once approved, the enhancement is implemented and a pull request is made in GitHub. Our automated continuous integration server (BatLab) runs

the full suite of unit, integration, and regression tests. Before a proposed change is allowed into Cyclus, it must be covered by a test and all tests must pass.

Unit tests cover code units like functions and are implemented using the GoogleTest framework. Integration and regression tests are performed by running sample simulations and verifying that results match predictions or previous results. A set of standard input files are run, then the output is inspected and compared via Nose, a unit testing framework in Python.

Similarly, API changes must be documented as required by the Cyclus documentation CEP. The documentation for the current stable branch and the development branch are both provided on the Cyclus website using Doxygen and Sphinx, which are both automatic documentation systems that rely on the code comments in the C++ and Python code, respectively.

Finally, we use the Google C++ style guide to make our code as consistently formatted as possible.

When the change is made, a developer begins to conduct a particular analysis by creating an input file. That input file is provided to the Cyclus framework and validated by its input validation framework. According to the input file, a simulation is run. The output database that is produced contains important metadata about the simulation. It contains:

- A complete copy of the input file.
- The commit hash of the current version of the Cyclus code.
- Commit hashes for all necessary plugins retrieved from the Cyclus ecosystem.

That database, containing both data and metadata, can then be analyzed by the user. When analyzing the database, a choice is made by the user about how to interact with the data. The Cyclus development team has provided a GUI (Graphical User Interface) and a Go library (called CyAn) with which the database (in either SQL or HDF5 format) can be accessed and brought into memory for visualization and analysis. Additionally, many user-developers have their own set of Python scripts that can do this stage of tasks. Given the universal nature of these database formats, most common scripting languages can be used to extract the data and metadata efficiently, so many options exist.

In summary, the research workflow in this framework has the following steps:

1. If necessary, a developer proposes a change to support their analysis.
2. The change is made including passing tests and satisfactory documentation.
3. It is reviewed and pulled into the master branch.
4. The software is rebuilt and installed using our build system.
5. A simulation is defined in json or xml.
6. The input file is run and an HDF5 or SQL database results.
7. The database is analyzed with a separate GUI, python scripts, or a Go library.
8. A collaborative paper is created in LaTeX on GitHub.
9. All input files contributing to the analysis are contained in the repository holding the document.

All of these steps are conducted in the context of Git and GitHub.

PAIN POINTS

Build systems are painful. In particular, cross-platform configuration and builds are an enormous time-sink for our research group. There are a number of reasons for this.

First, supporting C++ builds on Windows is sufficiently difficult that we abandoned supporting that platform.

Also, due to the physics-based solvers and optimization calculations in our simulations, external library dependencies are essential to Cyclus. We rely on libraries like Boost and LibXML2 to facilitate development, and we rely on libraries like Blas, Lapack, and COIN for mathematical solvers. For this reason, new developers spend a nontrivial bulk of their spin-up time building and installing the dependencies necessary to install Cyclus on their particular platforms.

Finally, our continuous integration system relies on our ability to create scripts that build, install, and test Cyclus. For this, we use a set of servers at the University of Wisconsin called the BatLab. Unfortunately, BatLab has a few problems. Because of the proprietary nature of MacOSX, it cannot run truly MacOSX instances. It runs, instead, Darwin servers that mimic the behavior of MacOSX. For this reason, idiosyncratic failures apparent in Mavericks and Yosemite but not Darwin cannot be caught before entering the code-base. Additionally, BatLab is

somewhat unpredictable and inflexible. Since the behavior of BatLab undergoes a lot of churn, our continuous integration suite is sometimes rendered completely useless.

KEY BENEFITS

The Cyclus Enhancement Proposal (CEP) strategy was a bright workflow choice that was inspired by the analogous strategy in the Python community (PEPs). I recommend this to any research group that values strategic planning, consensus, and thoughtful development. A discussion of our workflow around these proposals can be found at http://fuelcycle.org/cep/cep1.html.

Fundamentally, a CEP is a design document providing information to the Cyclus community, or describing a new feature or process for Cyclus and related projects in its ecosystem. The CEP should provide a concise technical specification of the feature and a rationale for the feature.

CEPs document major new features, community discussions, and documentation of theory or design not captured by the in-code documentation. Because they are maintained alongside the website source code in a version controlled repository, provenance of the discussion surrounding their acceptance is maintained.

KEY TOOLS

We use CMake to configure and build our software. Much more human readable than the configuration files within the GNU auto-tools suite, CMake makes our lives easier.

The continuous integration system, though difficult to implement due to build issues, has decreased development time. It would not be possible without CMake, GoogleTest, and Nose.

QUESTIONS

What does "reproducibility" mean to you?

A reproducible research product is one that has been sufficiently documented, well-constructed, and preserved for its results to be re-created by some external researcher or group.

Why do you think that reproducibility in your domain is important?

My domain, nuclear engineering, is one where precision and accuracy are both of utmost importance to both human and environmental

outcomes. Any conclusions drawn by science can only make an impact in the real world if they can meet the standards set out by the Nuclear Regulatory Commission. For this, reproducibility is paramount.

How or where did you learn about reproducibility?

I learned these practices primarily from my advisor, Paul P.H. Wilson at the University of Wisconsin, Madison. I also learned from colleagues in The Hacker Within, the Scientific Python community, and Software Carpentry.

What do you see as the major challenges to doing reproducible research in your domain, and do you have any suggestions?

One major problem is export control. Making software and data open source is restricted by the US Department of Energy, in some cases.

What do you view as the major incentives for doing reproducible research?

Fear. The fear of retractions due to faulty software or data can be reduced by enforcing transparent reproducible practices, which tend to reduce the likelihood of being accused of scientific fraud.

Surprise. Six months after a paper is submitted, the surprise of no longer recalling your own thought process is unpleasant. To avoid it, reproducible practices can help you reproduce your present work in the future.

Ruthless Efficiency. The automation inherent in reproducible workflows makes tweaking and rerunning of simulations and analysis very efficient.