Scientific and High-Performance Computing Practices at AER

At Atmospheric and Environmental Research (AER), we have a long history of scientific computing in fields including, but not limited to, weather forecasting, atmospheric radiation, air quality, space weather, and oceanography. AER develops and tests software internally; on supercomputers at NERSC (`cori`), NASA (`pleiades`), and NOAA (`s4` and `orion`); and on the Amazon Web Services (AW) cloud. We deliver software that models physical and chemical processes, extracts data, and performs analysis, and the software is trusted by agencies such as NASA, NOAA, NSF, the Department of Defense, the Department of Energy, and numerous others. With deliveries to these customers, AER must focus on software development that is resilient and reproducible, and this document describes how we are able to generate these products.

Karen Cady-Pereira and Rick Pernak of the Radiation and Climate (RC) group at AER, in response to the RFI posted by DOE's Advanced Scientific Computing Research (ASCR) program, will address a subset of the nine categories for which information is solicited. The subset includes the categories:

- Software dependencies and requirements for scientific application development and/or research in computer science and applied mathematics relevant to DOE's mission priorities (Category 1)
- Practices related to the security and integrity of software and data (Category 2)
- Infrastructure requirements for software development for scientific and high-performance computing (Category 3)
- Developing and maintaining community software (Category 4)
- Requirements, barriers, and challenges to technology transfer, and building communities around software projects, including forming consortia and other non-profit organizations (Category 6)

Contents

Software Dependencies

Most of the scientific computing in our group is now written in Python, unless high performance computation or operational code (embedded on instruments like GOES-R or installed in data centers) is necessary. In these cases, lower-level programming languages like FORTRAN and C++ are utilized. Some of our RC group's legacy scientific computing still includes higher-level languages like IDL and R, but for better support and consistency with the software development community, Python usage is replacing these languages at our company. Python is more widely used outside of the scientific community and thus development is more mature, so sources like StackOverflow and GitHub typically contain posts that help immensely with development, debugging, and testing, and libraries exist for almost any purpose we need for scientific applications (e.g., reading different types, visualization, heavy computation, machine learning, etc.). No license fees apply, either, making Python a better value.

Going forward, we believe DoE's ASCR should continue to support older, legacy code, but recommend users adopt more modern practices. These include:

- Python should be suggested over Perl, Bash, or IDL
- Institutions should install only the most commonly used non-standard libraries (e.g., `numpy` for Python), but less-common APIs (e.g., `netCDF4` or `xarray` in Python) may be best left to the end users and creating their own environments.
- End users should know how to set up virtual environments
- Applications should be run in containers so end users only spend time running them instead of trying to build them and all of their dependencies.
- Because we recommend the usage of containers, it's vital for supercomputers to host repositories for the associated images.

We will elaborate on these concepts more in this section. Knowledge of these concepts will provide a solid foundation for scientific programmers to produce more operations-ready applications. However, in some fields it will still be beneficial to know how to at least read legacy code like FORTRAN, C, and IDL.

Python

Installing non-standard Python libraries can be problematic, but package managers exist to facilitate the process. One such package manager is miniconda, and once installed in user space (not at an administrator or root level), dependency installation is seamless. Users can create their own virtual environments (VEs) for the programming language, and they can be comprehensive or task-specific. Numerous VEs can be generated, and users can switch between them easily. After configuration of the VE, path issues become less common and, if the VE is set up correctly, perhaps non-existent. Giving end users the flexibility of creating their own, local environments instead of leaving the task up to administrators is far more preferable and efficient.

Currently, the commonly used Python APIs at AER can be organized into different categories:

- Computation: `numpy`, `scipy`, `dask`
- Complex data sets: `netCDF4`, `xarray`
- Visualization: `matplotlib`, `cartopy`
- Data Science and Machine Learning: `pandas`, `scikit-learn`
- Collaboration: Jupyter notebooks

The `numpy`, `scipy`, `netCDF4`, and `matplotlib` are effectively ubiquitous in the scientific community, as they contain vital modules for mathematical data types (e.g., arrays and vectors) and procedures (e.g., regression fitting, matrix multiplication), data organization (`netCDF4` datasets), and visualization, whether they be simple diagnostic and exploratory plots, images, or publication-ready figures.

Other packages listed are not as widely installed, but they have incredible utility and knowledge of them is just not as proliferated, likely because they are newer packages. `cartopy` is particularly useful for our company, because we frequently need to overplot our data or model output onto a map of the Earth. `cartopy` assists with this process and provides a number of different projections. `dask` is great for scaling from small desktop systems to large, parallelized computing clusters. `xarray` is an extension of `netCDF4` that has more features for user-friendliness and is more efficient (not all data are stored to memory). Jupyter notebooks facilitate collaboration by providing code snippets, documentation, and figures all in one easily-viewed and edited web page. JupyterLab on DoE's `cori` supercomputer at NERSC was utilized by AER scientists and collaborators recently, and running of code and sharing of results could not have been easier.

Lower-level Languages

In lower level languages (C++, C, FORTRAN), we commonly need APIs installed. An example is the `netCD4` library, which has a number of dependencies of its own (`HD5`, `szip`, `zlib`). Since AER's Linux systems -- desktop and servers -- share architectures (e.g., operating systems), we can typically install these kinds of libraries from source in a file system that all users can read and thus linked to when compiling executables.

Docker

A better alternative is to exploit Linux images and containers, typically through a service called Docker. Images are developed with specific applications in mind -- e.g., an air quality or radiative transfer model -- and **only** contains what is necessary for

that application to run. These requirements include a given OS, libraries, and static inputs. Images can be viewed as lightweight virtual machines that exist for a specific purpose. Running a container then entails providing the image with dynamic inputs and options that are necessary for the application to run.

As an example, a radiative transfer model image can contain:

- CentOS 7 OS
- netCDF4 library and dependencies
- absorption coefficient files

Then a container can be run by specifying:

- atmospheric conditions for which to model the radiative transfer
- runtime options including container name and volume mounts so output can be accessed

Security and Integrity of SW and Data

AER works on a number of DoD contracts and thus is subject to federal regulations protecting this kind of technology (e.g., ITAR). Data and code that should not be publicly available are saved internally on encrypted systems. Version control of code and documentation are done through an AER-hosted Gitlab server so we can ensure compliance with ITAR regulations. Only US persons at AER have access to the server, and groups can determine whether their projects are viewable by all US persons at the company or a select few.

For data that can be available to the public, we share them on our own community on Zenodo. This service is provided freely by CERN, and other services like it are recommended for the purposes of open access and reproducibility.

Publicly available code is available on GitHub. For example, our Radiation and Climate Group has its own AER-RC GitHub group with the models that we have validated over decades via contracts that supported data collection and analysis for remote and *in-situ* sensors. Again, visibility of the repositories can be controlled by the group or users, and further opacity can be introduced by developing under specific users until development is more advanced and ready to be viewed by the public. Repositories can be quickly transferred between users or from a single user to a group.

One way we have ensured integrity of our model output is by providing transparency into the aforementioned validation studies, and we do this through our publications and code releases. We have a number of what we call "run examples" that are released with the code that provide end users with "truth" as generated by the AER RC group. This gives users a baseline with which they can verify if their model has been built correctly. Perhaps more closely related to validation, before every model release, we compare our model results to "truth" from satellite measurements across the IR and MW spectrum. Further verification is done by comparing the pre-release and release version of the code and ensuring changes to the model results are consistent with what we expect, given the modications.

Differences in building the models can be bypassed by instead running our models in Docker containers. Our group plans to modernize the validation process by including code and baseline results in a version-controlled Jupyter notebook that contains run examples and figures of expected results. In any case, the verification and validation processes can be codified by including everything in a Continuous Integration Continuous Deployment (CI/CD) pipeline, where every commit into a code repository triggers these kinds of tests for building the code then comparing to truth. CI/CD can be mostly automated, giving developers more time to innovate instead of manually running the tests.

As programs grow, so do their verification and validation (V&V) processes, but if proper CI/CD pipelines are designed from the start, extensions can be trivially added to the CI/CD configuration, and validation with data can be as simple as adding another section of code snippets and comparison plots to a Jupyter notebook.

Infrastructure Requirements

Source Code Version Control

To start, version control of code and perhaps static data is a must. In years past, our IT department would be involved with CVS and SVN (internal and external) servers, but most of our repositories are hosted on our aforementioned internal Gitlab server or public GitHub. On NOAA's `s4` supercomputer, a Gitlab server is hosted as well, so collaboration amongst contractors, sub-contractors, and customers can be seamless and code deliveries are easier. Gitlab transfers more control from IT to the end users on specific projects, which also improves efficiency. Group members have control over who sees their code.

Gitlab and GitHub provide avenues for better documentation of code through verison-controlled READMEs, wiki pages, and issue tracking. Its web interface visualizes code versions comparisons, which was non-existent with Git's predecessors.

Image Repositories

Should end users adopt the usage of Docker and containers, it is important to have access to a server that hosts the images so they can be easily pulled and used by end users. Options include DockerHub, GitHub Container Registry, and AWS Elastic Container Registry. These work similarly to version control repositories of code, but they do not exist for comparing different versions -- only for cataloging all of the "tags" (versions) of an image. Limits on the number of times images can be pulled do exist for some services, though, namely DockerHub, so users must be mindful of these quotas in their testing and developing environment.

Artifact Repositories

Source code repositories ideally only contain text, which are easily implemented into `diff` visualization applications so code commits can be compared to each other. Sometimes, binaries need to be managed as well, but they should be in an "artifactory" (artifact repository) that is separate from the source code. We have a server that hosts this kind of repository at AER.

Infrastructure as Code

Infrastructure as Code (IaC) usage is quickly growing at AER, given our adoption of cloud computing best practices. Services such as Terraform and AWS CloudFormation codify resource usage so infrastructure reproducibility are readily achievable. Infrastructure configuration is specified in YAML or JSON format, both of which are ASCII text, so version-controlling of infrastructure setup is also possible and encouraged.

Developing and Maintaining Community SW

If version control and documentation is practiced from the start, very little effort for community availability is needed. However, one or both of these aspects are frequently ignored when maintaining scientific code. Properly documenting in both the code and via Wiki pages in GitHub or Gitlab thus is a large portion of the effort to migratation to the scientific community. Another challenge is portability and flexibility of the software -- was it just run on a test case, an *ad-hoc* sample, or a comprehensive population? Additionally, does it work on other computer architectures (OS, compiler, etc.)? This kind of conversion can also be time-consuming. Inevitably, one or even a few developers cannot reproduce *all* of the possible bugs that might arise when the community starts using the software. Issue tracking ability becomes of even more importance at the this point.

Beyond code usage, community software development can be hindered for a couple of reasons that are immediately apparent to our group. First, the vast majority of funded work is for specific software development and results analysis, which leaves little time to properly document the code and results in the most general sense that might assist others in the scientific community when trying to use the novel software. Another impediment is end user inertia. Scientists might not be accustomed to code that is developed with software engineering best practices in mind. The learning curve and effort needed to adapt may outweigh the perceived benefits.

One example with AER is the aforementioned AER-RC that hosts repositories with publicly available radiative transfer modeling code. All of the models were once hosted in SVN and on a static website that was updated every few months and had to involve our IT department in both instances. Even with the best intentions, model releases would take days or weeks. Development (branching off the main repository, then merging modifications back onto the main branch), releases, and packaging of the code (which currently includes a binary file with run example baseline results that end users can use for validation) now occurs within the AER-RC group and only in the GitHub web interface. All documentation is written in Wiki

pages also hosted by GitHub, README Markdown files that are version controlled in the associated repository, and issues that are tracked by both the AER-RC group and user community when problems occur. Legacy code was already reasonably documented. Very little company time was spent on this migration, and there was considerable resistance within our section of the company, and a proof-of-concept with an extended evaluation period was necessary to convince users accustomed to the older system.

Requirements and Barriers to Technology Transfer

Migrating from "research-grade" code to production-ready requires a *desire* to develop code that is easily maintained -- well-designed, modular, well-documented, portable -- and results that are reproducible. Some do not have this desire, and this was alluded to in the Developing and Maintaining Community SW section. Rather, a set of goals promised when applying for funding becomes the focus. ASCR should consider encouraging code flexibility and documentation, demonstrations on various operating systems or containerization of software (the latter is encouraged), unit testing, infrastructure as code, and allotting time for software design in future solicitations.