Toward Interactive Supercomputing at NERSC with Jupyter

Rollin Thomas, Shane Canon, Shreyas Cholia, Lisa Gerhardt, and Evan Racah
National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory

***Abstract*—Extracting scientific insights from data increasingly demands a richer, more interactive experience than traditional high-performance computing systems historically have provided. We present our efforts to date to leverage Jupyter for interactive data-intensive supercomputing on the Cray XC40 Cori system at the National Energy Research Scientific Computing Center (NERSC). Jupyter is a flexible, popular literate-computing web application for creating "notebooks" containing code, equations, visualization, and text. We explain the motivation for interactive supercomputing, describe our implementation strategy, and outline lessons learned along the way. Our deployment will allow users access to software packages and specialized kernels for scalable analytics with Spark, real-time data visualization with yt, complex analytics workflows with Dask and IPyParallel, and much more. We anticipate that many users may come to rely exclusively on Jupyter at NERSC, leaving behind the traditional login shell.**

1. Introduction

Across nearly all scientific disciplines, researchers face an explosion in data volume and velocity. This growth is largely driven by technological advances that allow for faster sampling rates and higher resolution in scientific detector instrumentation, higher densities of detectors, reliable high-capacity storage, and high-bandwidth networking for data transfer.

Modern large-scale scientific experiments and simulation codes produce vast amounts of data that far exceeds the capacity of clusters and desktops typically close at hand to the user-scientist. Just moving such data sets to compute resources at a user's home institution from the lab or experiment site may be impractical, necessitating remote-controlled data analytics and visualization capabilities. These capabilities are centralized at high performance computing (HPC) centers like the National Energy Research Scientific Computing Center (NERSC)[1].

HPC systems and supercomputers are usually shared among a large number of users and this necessitates asynchronous batch scheduling to manage access to compute resources. Yet scientific insights often spring from interactive, iterative exploration and analysis. A bridge between these two modes of scientific computing is needed, enabling "human in the loop" interactive exploration at Big Data scales (thousands of cores and tens to hundreds of terabytes).

In this paper we outline our efforts to bridge the gap between exploratory data analysis and HPC. Specifically, we are working to leverage the Jupyter project[2] to enable interactive data-intensive supercomputing on NERSC's Cray XC40 "Cori" system. We explain the motivation behind interactive supercomputing, describe our implementation strategy and the process behind the development of that strategy, and outline lessons learned along the way. These findings should generalize to other supercomputing centers and serve to further socialize interactive supercomputing among users, support staff, and vendors.

Our work is part of a much larger effort to make interactive supercomputing a reality at a number of supercomputing sites. Experiences shared with us by staff at those sites and the cooperation of Jupyter developers have been instrumental in our progress. We hope that our contribution provides valuable insight and experiences in kind.

The outline of this paper is as follows. In Section 2 we provide a brief background on Jupyter to orient the reader. Section 3 describes our initial deployment of Jupyter as a science gateway application. The subsequent expansion of Jupyter to a dedicated node on Cori is described in Section 4. Work in progress to support Jupyter on Cori compute nodes appears in Section 5. In Section 6 we discuss the strategy we developed over the course of our experiments with Jupyter, lessons learned, and perspectives on serving the full range of Jupyter user needs at NERSC. Section 7 concludes the paper, describing our vision of Jupyter as an integral part of how users interact with HPC at NERSC.

2. Background: Jupyter

Project Jupyter [2] evolved from the IPython project. IPython itself began as an enhanced interactive Python shell including advanced features like language introspection, syntax highlighting, history, tab-completion, access to documentation and much more [3]. Over time, alternatives to the IPython terminal application were developed, including a graphical user interface based on Qt, and finally a web application framework. These interfaces are centered around the concept of a *notebook* document.

Notebooks are documents that contain both computer code and rich text elements (paragraphs, equations, figures, widgets, links). They are human-readable documents containing analysis descriptions and results but are also executable documents for data analysis. Notebooks can be

1. http://www.nersc.gov/

2. http://jupyter.org/

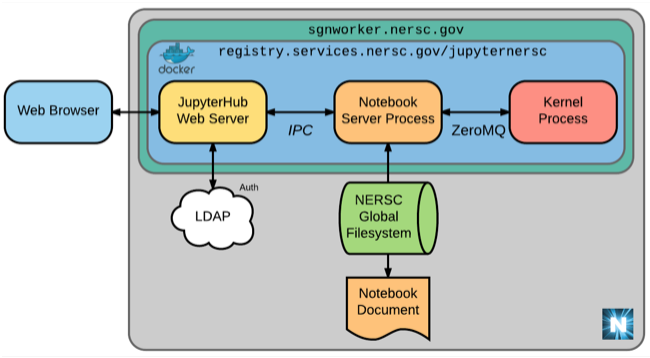

Figure 1. Science gateway application deployment for Jupyter at NERSC. JupyterHub, spawned notebook server processes and kernels all run in a Docker container. The Docker container is hosted on a science gateway "edge service" node at NERSC. Authentication to JupyterHub is handled via LDAP. Part of the NERSC Global File system is exposed to users via the notebook server. Notebook documents themselves are hosted by default in the user's global home directory.

shared between researchers, or even converted into static HTML documents. As such they are a powerful tool for reproducible research and teaching.

A notebook is associated with one or more computational engines called *kernels*. These kernels execute code passed to them from a notebook process. When the Jupyter project was spun off from IPython, IPython itself became one of many kernels. A large number of kernels for a number of languages and programming environments have been developed to work with Jupyter.[3]

On a laptop or single-user workstation, a user typically starts up the Jupyter notebook server application from the command line and users a web browser on the same system to author notebooks. JupyterHub [4] is a web application that enables a multi-user "hub" for spawning, managing, and proxying multiple instances of single-user Jupyter notebook servers.

At NERSC, JupyterHub itself is run as a science gateway application. The hub itself runs as a web server process inside a Docker container. This Docker container runs on an edge services node (a node that provides external services but lives within the NERSC network, and has high-bandwidth connectivity to the HPC system). Users authenticate to JupyterHub using their NERSC credentials. Depending on the exact configuration, notebook and kernel processes may be spawned within the Docker container, or on an external server. Much of our work has centered on mechanisms for spawning notebook and kernel processes remotely without needing to invoke escalated privileges (e.g. running processes as the super-user).

3. Jupyter as Science Gateway Application

Jupyter was first deployed in late 2015 and announced to users for general usage in early 2016. Our initial deployment, leveraging existing NERSC science gateway infrastructure (edge services hosting web portals, web applications, REST APIs and data for science users) is depicted in Figure 1.

In this configuration, the user authenticates via the JupyterHub web application running in a Docker[4] container on a science gateway node. Docker is an elegant solution for hosting Jupyter, as it provides portability and scalability. In addition, we have come to manage our Jupyter images using Dockerfiles, which provide self-documentation and versioning. The Rancher[5] system for container management and deployment has proved very useful for monitoring and orchestrating container resources at NERSC in general, and we plan to control all Jupyter image deployments through Rancher going forward.

NERSC user credentials are verified via LDAP. Upon successful authentication the notebook server process is launched and this process communicates with the user's web browser via websocket connections. The user selects a kernel to launch from a pre-defined list. The notebook and kernel interact via a ZeroMQ[6] message queue. JupyterHub, per-user notebook servers, and spawned kernels are all limited to run in the Docker container. In this mode, we were able to run JupyterHub without any major code modifications or extensions. Local customizations included the ability to traverse the entire file system (while automatically defaulting to the user's home directory), and support for multiple kernels (Python 2, Python 3, R, and deep learning libraries). We also provided users with a mechanism to run their own custom kernels.

This configuration allowed us to run an initial Jupyter service at NERSC with limited access to center resources. Users access data on the NERSC Global File system, in particular their home directories and the shared `/global/project` file systems. Notebook documents by default are created in the user's home directory.

Within six months of activating the service around 100 NERSC users had already tried Jupyter. Several projects came to rely on Jupyter very quickly for data analysis. These include projects like OpenMSI [5] and the LUX experiment [1]. But as expected, users began to push the boundaries of what was feasible for them to do with Jupyter as a science gateway application, and they began to ask for tighter integration with NERSC's HPC and massive storage resources.

One issue is that while Jupyter, as a science gateway node, mounted the GPFS file systems, it had no access to Lustre scratch file systems on the Cray systems. This is problematic for users who may, for instance, write large simulation data sets to Lustre scratch where they have a much bigger quota than on the GPFS file systems. Users would have to move data across file systems just to access it from Jupyter. This was a time consuming, inefficient, and frustrating process.

3. https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

4. https://www.docker.com/
5. http://rancher.com/
6. http://zeromq.org/

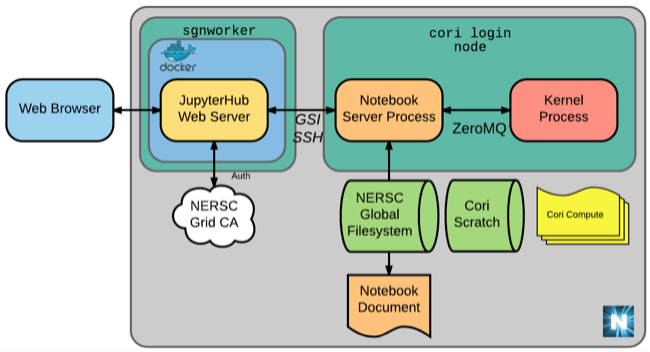

Figure 2. Cori data analytics node deployment for Jupyter at NERSC. In contrast to Figure 1 only the multi-user hub application runs inside a Docker container on a science gateway node. After authenticating to JupyterHub, users' Grid Certificates are retrieved and used to establish a GSI-enabled ssh tunnel to the Cori data analytics node, essentially a repurposed "login" node. Notebooks and kernels spawn on the Cori node. In addition to the global file system, Lustre scratch, batch queues and Cori's software environment are made accessible from Jupyter.

We also found that users often expected to have a software environment matching what Cori provides at the shell login, and they were often astonished to find that Python packages they could easily use on Cori were not present in Jupyter. This is because the Python stack being accessed by Jupyter is installed into the Docker container. Part of the problem is that users did not appreciate the distinction, but it also meant that other software components they could access (e.g. applications they had built on Cori) would not run due to dynamic linking problems.

Given these barriers, our desire to expand the service, and offer better hardware for running Jupyter, we began to investigate using Cori for running Jupyter.

4. Jupyter as Cori Data Analytics Application

Migrating or extending the existing Jupyter service from a smaller-scale science gateway node onto the Cray system provides tighter integration between Jupyter, Cori's various data-friendly features, and storage systems. Cori has been designed with an eye toward data-friendly policies like real-time, interactive, serial, and shared queues; enhanced external network connectivity for compute nodes; accelerated I/O via a burst buffer; and visualization, analytics, and complex work-flow orchestration through a pool of large memory repurposed login nodes. One such specialized node is a "data analytics" node set aside for running interactive data analysis and visualization tools just like Jupyter. This node has more than 500 GB of RAM (a significant upgrade over the 64 GB science gateway node) as well as direct access to Cori resources including networking and storage.

Figure 2 depicts how we run Jupyter on the Cori data analytics node. In Section 3 we described a setup where all Jupyter components were confined to the same Docker container. Here, only the multi-user hub application is running in a Docker container. The hub still manages user authentication via LDAP. The key difference is that single-user notebooks are spawned remotely on the data analytics node.

To enable remote spawning, we have developed a custom authenticator for JupyterHub called **`GSIAuthenticator`**[7] (GSI: Grid Security Infrastructure) that allows users to acquire a grid certificate upon login. A custom spawner that we also developed called **`SSHSpawner`**[8] spins up a Jupyter notebook on Cori via GSISSH[9] (a modified SSH that adds the ability to perform X.509 proxy credential authentication and delegation). Once launched, the Jupyter notebook connects back to the hub over a websocket. The hub then proxies all future user requests to the Cori node via this websocket connection. Users interact with their notebooks running on Cori, launching pre-installed or custom kernels to analyze and visualize their data.

This configuration addresses many of the shortcomings of the preceding iteration. Cori's Lustre scratch file system is available to Jupyter naturally. The Cori software environment available at the shell login and from Jupyter are the same. In fact, Jupyter on Cori is installed as part of the Anaconda Python distribution available to users from the login shell. This makes management of custom kernels and user-managed software much more straightforward.

To expose some level of batch queue functionality through Jupyter, we developed a set of "magic commands" that allow users to interact with the Slurm workload manager on Cori. Magic commands are a construct in Jupyter/IPython that enable extra-language functionality. In the case of Slurm magic[10] we provide a thin wrapper around the Slurm command-line API.

For instance, to query for a list of all running jobs, a user types **`%squeue`** just as they would at the command line. The difference here is that what is returned is a first class Python object (a Pandas dataframe) that they can manipulate. Batch jobs can even be submitted via the **`%sbatch`** command (taking the path to a batch script as argument) or via the **`%%sbatch`** "cell magic" — where the contents of the batch script are provided in the the Jupyter code cell itself.

A number of notebooks used in user training events, including some kindly contributed by users, are published via GitHub.[11] User-contributed notebooks include one used to calibrate the LUX detector, and another demonstrating the use of a database at NERSC to render photometry simulations of Large Synoptic Survey Telescope[12] observations.

Jupyter on the Cori data analytics node is again a very popular service, almost too popular. As with Jupyter on the science gateway we periodically have to remind users that they are sharing a single node — even though the amount of memory and CPU power is much greater. The next step is expanding the service to the compute resources available on Cori.

7. https://github.com/NERSC/gsiauthenticator
8. https://github.com/NERSC/sshspawner
9. http://toolkit.globus.org/toolkit/docs/5.0/5.0.4/security/openssh/pi/
10. https://github.com/NERSC/slurm-magic
11. https://github.com/NERSC/new-user-training-notebooks
12. https://www.lsst.org/

5. Toward Jupyter on Cori Compute Nodes

Because it provides some form of interaction with HPC resources, a unified software environment, and exposes the Cori scratch file system, the data analytics node solution provides users with a bit more "power" than the science gateway node can. However, it is still just one node, and as the number of users grows we expect it to be unable to keep up with demand. The obvious next step is to identify and implement methods for launching Jupyter processes on Cori compute nodes. But launching Jupyter processes on compute nodes presents a number of challenges.

One issue is that access to the compute nodes is managed through a workload manager or batch scheduler. Certainly users can make reservation requests that set aside nodes specifically for their exclusive use ahead of time. This is a useful approach for teaching settings where Jupyter notebooks are used for education and training. To be able to run Jupyter on demand, it is clear the easiest solution would be a set of dedicated nodes or an overhead of floating idle nodes that could be called upon to run Jupyter at user request. The issue with this approach is that full system utilization is a very high priority for most HPC systems. On Cori we are able to offer reservation-based requests for running Jupyter, but also have 192 Haswell and 192 KNL nodes dedicated for interactive use cases enabled by Slurm's quality of service (QOS) feature. For reservation-based requests, we are considering a streamlined self-service reservation interface that would allow users to schedule reservations without NERSC staff intervention within reasonable resource limits.

Our effort is not the first to try running Jupyter notebooks and kernels on batch systems. Previous work has been consolidated and released in the form of JupyterHub-affiliated projects that help address some of our needs. These include the **`BatchSpawner`**[13] and **`WrapSpawner`**[14] packages. **`BatchSpawner`** extends Jupyter's **`Spawner`** for batch queue systems and implements spawners for Torque, Slurm, SGE, and HTCondor. To launch Jupyter on Cori compute, a **`SlurmSpawner`** spawner instance is used to map requests for resources into a batch script to run the Jupyter single-user server. After ths script is submitted Slurm is polled to detect when the job has started. The node address is returned to JupyterHub and forwarded to the user who then automatically connects to the server running on the compute node.

To request resources for jobs, users need an interface built into Jupyter. For our prototype implementation we used the **`ProfilesSpawner`** class (part of the **`WrapSpawner`** package) to present a list of pre-defined sets of resources for jobs, defined in a configuration file. For instance, a user could request a single-node Cori job with a time limit of 30 minutes, or a three-node job with a notebook and two kernel processes for an hour. The concept behind **`WrapSpawner`** and **`ProfilesSpawner`** is that they "wrap" **`Spawner`** interfaces so they can be chosen at runtime. This behavior allowed us to present not just **`SlurmSpawner`** configurations for running Jupyter on Cori compute nodes, but also our **`SSHSpawner`** configuration for running Jupyter on the data analytics node, and even the science gateway configuration, under the same interface.

A final, major issue for running Jupyter on Cori compute nodes is that network access to those nodes from outside Cori is restricted. The current prototype implementation, sufficient for concept testing and early demonstrations, relies on a choreographed sequence of SSH tunnels, authentication forwarding, and scripts. In terms of performance and reliability it is highly suboptimal. A much better solution is available through software defined networking (SDN) [6].

Under the SDN scenario, we will be able to launch the job and then request a public IP that is routed to the head compute node. This address can then be communicated back to JupyterHub so it can appropriately route the connection the notebook. This capability is under development but an early prototype exists. Directly connecting to the head node will eliminate much of the latency of brokering connections through GSISSH and SSH tunnels. Initial tests using SDN to launch Jupyter on the Cray testbed at NERSC are promising. We expect to be able to move this to production on Cori by the end of summer 2017.

Figure 3 illustrates our vision for Jupyter on Cori. By extending the spawner infrastructure and using SDN we will be able to power Jupyter notebooks on the data analytics node and Jupyter notebooks on Cori compute nodes through a single platform. Within compute nodes many configurations for notebook-based workflows are possible. Users will no longer need to share compute resources to run Jupyter if they need all the cores or memory available on a node for their analysis tasks. Multi-node jobs involving schedulers and workers (for Spark and Dask) are possible.

The prototype setup has been used to run multi-node Spark jobs and visualization of N-body data sets using **`yt`**.[15] With **`yt`** we have even been able to leverage the use of IPyParallel with the Python **`mpi4py`** back-end. We are not yet able to scale these jobs reliably past about the 10-node level without SDN.

6. Discussion

In this section we reflect on the strategy developed over the course of our experiments with Jupyter on Cori and at NERSC. We also discuss best practices we have learned along the way. We believe this practical information should be useful to staff at other institutions who may be interested in running Jupyter on HPC systems.

Over time we have developed a strategy consisting of the following components:

- Using Docker to containerize components, Dockerfiles to manage construction of images, and Rancher for orchestration. While not all configurations of Jupyter at NERSC fit entirely into Docker containers, the benefits that come with containerization

13. https://github.com/jupyterhub/batchspawner
14. https://github.com/jupyterhub/wrapspawner
15. http://yt-project.org/

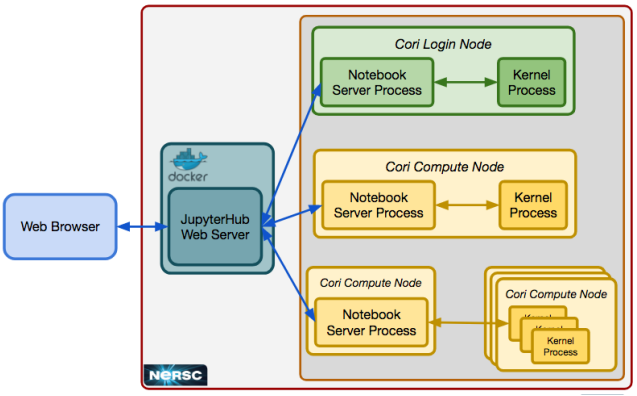

Figure 3. Plan for Jupyter on Cori at NERSC. As in Figure 2, the multi-user hub application runs in a Docker container on a science gateway node. The data analytics (repurposed login) node configuration currently in production appears at the top of the figure in green. The gold boxes below represent two potential configurations of Jupyter processes running on Cori compute nodes. In the middle box, a user simply requests a compute node to run a Jupyter single-user server and kernel process with exclusive use of a Cori compute node for some period of time. The boxes at the bottom depict a more involved workflow where a user requests multiple nodes, running a notebook server process on the "head" node of the job, and possibly multiple kernels running on the other nodes in the job allocation. Spark, Dask, and IPyParallel jobs are typical examples of workflows using this job configuration.

(self-documentation via Dockerfiles, standardization, flexible deployment, etc.) will make it possible to scale components of the service given limited staff resources.

- Capitalizing on the Jupyter ecosystem's modular design to customize our deployment to address security requirements, address performance issues, and provide a NERSC-centric user experience. Where possible we prefer to extend existing components and contribute them back to the community.

- Communicating and engaging with the Jupyter developer community through personal contact, workshops, email lists and chat. Jupyter is a large and rapidly evolving system which makes accurate up-to-date documentation a challenge. Often the best advice comes directly from interacting with the developers.

- Proceeding with an incremental deployment strategy. This has been essential in building up expertise within the staff regarding Jupyter as we move toward bigger challenges. As demonstrated in Sections 3 and 4, components from previous deployment phases can sometimes be re-used or adapted.

- Leveraging SDN for orchestration of remote processes on the HPC systems. Experimental efforts used a series of SSH tunnels to get the system running on Cori compute as proof-of-principle. More direct networking connections enabled by SDN will provide a more elegant and performant approach.

Based on both user feedback and usage metrics, it is clear that there is substantial demand for interactive services like Jupyter at NERSC. It is also clear from the same sources that users have a spectrum of needs, and we imagine that this holds at other HPC centers as well:

- Those with analysis and visualization workflows that do not expressly require HPC resources but still need to explore massive data sets stored at NERSC may find Jupyter delivered via science gateways to be ideal. Science gateways are almost completely independent of the HPC systems at NERSC and so planned or unplanned outages of the HPC systems would not affect such users. Managing the entire Jupyter stack through Docker and Rancher also allows us to scale up (or down) easily in response to demand.

- Using Jupyter to manage and interact with HPC work-flows appeals to many users. We envision the data analytics node being used for interacting with such work-flows through tools like Slurm magic commands and analysis/visualization of data stored on Cori scratch.

- Spawning Jupyter notebooks and kernels on Cori compute nodes, coupled with interactive QOS, can give each user their own, isolated "Jupyter node." A single data analytics node on Cori is insufficient to service all NERSC Jupyter users, so development is currently focused at making this work in a streamlined way.

- The largest data analytics jobs will require multiple nodes running Jupyter components, in a number of different configurations. Interactive data analytics powered by Spark or Dask via notebook, one or more scheduler processes, and a large number of kernels running will require multi-node allocations. Alternatively it may be possible to run a notebook on the data analytics node and connect to a job running Spark or Dask on the compute nodes. To enable these work-flows, interactive QOS and SDN are both indispensable.

Given this broad spectrum of needs it is clear that one challenge we face will be presenting a sensible and unified Jupyter-based ecosystem to users that exposes the right resources for the task at hand. To address this challenge we are developing a new interface, building on **`WrapSpawner`** but exposing more than a simple list of allowed configurations for a single system. At this point it is not clear how that can generalize to other HPC centers, since each has its own queue policies and different types of resources from what NERSC provides. We suggest that standardized APIs abstracting HPC center resources, for example NEWT,[16]

16. https://newt.nersc.gov/

could have a role in helping to make work like ours more easily transferable.

7. Conclusion

We have outlined our efforts to date to enable interactive, exploratory data analysis on supercomputers at NERSC using Jupyter. We have explained the motivations behind interactive supercomputing, and presented our implementation approach and described lessons learned along the way. Examples of Jupyter notebooks from NERSC user training events and contributed from actual NERSC users are available on GitHub for reference. Our experiences, code, and these materials should be useful in moving forward interactive supercomputing in the HPC community.

At least at NERSC, we can envision a day when a good fraction users rely exclusively on Jupyter, or a framework similar to Jupyter, for large-scale exploratory data analysis. Many of them may seldom or never use a traditional login shell.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank our colleagues in the Security and Networking, Infrastructure Services, and Computational Systems Groups at NERSC for advice and valuable contributions to the infrastructure necessary to run Jupyter. We are grateful to Andrea Zonca (San Diego Supercomputing Center), Michael Gilbert (Northern Arizona University), and Michael Milligan (Minnesota Supercomputing Institute), for their successive advancements of **`BatchSpawner`** and related packages. We also thank the Jupyter developer community for advice, feedback, and collaboration. We thank Evan Pease from the LUX collaboration and Bryce Kalmbach from LSST for sharing their notebooks with us. Last but not least, we thank our users who have helped shape the trajectory for Jupyter at NERSC.

References

[1] Akerib, D. S., et al., *The Large Underground Xenon (LUX) experiment*, Nuclear Instruments and Methods in Physics Research A, Volume 704, p. 111-126, doi:10.1016/j.nima.2012.11.135

[2] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, Jupyter Development Team, *Jupyter Notebooks - a publishing format for reproducible computational workflows*, Positioning and Power in Academic Publishing: Players, Agents and Agendas p87-90, doi:10.3233/978-1-61499-649-1-87

[3] Fernando Pérez, Brian E. Granger, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

[4] Jupyterhub, https://jupyterhub.readthedocs.io/

[5] Oliver Rübel, Annette Greiner, Shreyas Cholia, Katherine Louie, E. Wes Bethel, Trent R. Northen, and Benjamin P. Bowen, "OpenMSI: A High-Performance Web-Based Platform for Mass Spectrometry Imaging" Analytical Chemistry 2013 85 (21), 10354-10361, DOI: 10.1021/ac402540a.

[6] Richard S. Canon, Brent R. Draney, Jason R. Lee, David L. Paul, David E. Skinner, and Tina M. Declerck, "Enabling the Super Facility with Software Defined Networking", Cray Users Group, 2017