

Graphical user interface to perform glacier simulations with PISM

M. Urbanč*, M. Depolli*

* Jožef Stefan Institute/E6, Ljubljana, Slovenia
urbancmrko1@gmail.com

Abstract—Computer simulations of glacial dynamics help us understand how glaciers respond to changes in climate, such as warming temperatures and increased precipitation. Simulations are performed with PISM (Parallel Ice Sheet Model). To tackle the complexities of ice dynamics, PISM employs several sub-models for various aspects and thus exposes a plethora of parameters. There are also several options for setting parameter values, i.e. through config files, command line options or input files. To get a good handle on all the provided buttons and knobs, we developed a user interface for setting parameters of our simulations and for executing PISM with the selected set of parameters on a remote workstation.

In this paper we present the GUI developed in Jupyter and deployed in a local JupyterLab installation. The environment contains DEM files, Python scripts and other inputs locally but connects to a 128-core workstation for simulation execution. The simulation inputs and outputs are transferred between the computers over an SSH connection. The GUI is publicly exposed in a single-user environment, only for use, while development is performed on a git repository which is then synchronized to the public through git. This approach is general enough to be used on other software as well.

Keywords—PISM, glacier simulations, user interface, jupyter, distributed execution, remote management

I. INTRODUCTION

A. Motivation

PISM (Parallel Ice Sheet Model) [1], [2] is a complex simulation tool that can be used in wide range of applications that require simulation of ice sheets or glaciers. Performing glacier dynamics simulations is far from trivial, PISM is designed to be executed primarily through scripts; even the tutorial available in the official documentation presents it through scripts. A scripting approach makes in silico experiments highly reproducible, since the whole experiment can be defined by a script, and the script can be archived. However, handling multiple geographical areas of interest and multiple modelling choices eventually leads the relatively simple execution scripts to evolve into a complex ecosystem of scripts. Such an ecosystem is difficult to manage and presents a formidable barrier for new researchers that are assigned to such an experimentation task.

Within our research, we found that our typical workflow for conducting in silico experiments on glaciers can be defined as follows.

- Select the geographical area to perform the experiment on from a predefined set of options. The

geographical area uniquely defines a set of variables: the DEM (Digital Elevation Map), climatological data and expected glaciation extent.

- Select other PISM modelling parameters and parameters of the custom climatological models to fully define the experiments.
- Format all the inputs appropriately for the use in PISM.
- Execute PISM, either locally or on a remote workstation. In the latter case, communicate the input and output files between the local and remote machines.
- Analyze PISM outputs to evaluate the experiment. Some of the analysis can be fully automated while some is done interactively or on demand.

To aid in executing the workflow swiftly, correctly, and with repeatable results, we avoided scripts and rather designed a graphical user interface (GUI) for the experimenter. It is written in a combination of Python, Jupyter notebooks, and helper bash scripts. The presented approach could be viewed as an extension of the scripting approach with the majority of functionality migrated from bash to Python and from command line arguments to graphical elements. Python has been chosen because of its flexibility and capability of implementing complementary models to those included in the PISM itself. The visualization is implemented in Jupyter notebooks [3], [4] for ease of use and beauty of presentation. The workflow based on these two parts of GUI is presented in Figure 1. The presented approach was very successful within the context it was created for and it seems to be general enough to be applied more broadly. Therefore we present it as a concept that could be copied by researchers facing a different complex simulation software.

In this paper we focus only on the *main* GUI part/Jupyter notebook which is responsible for parameter setup, execution with execution monitoring, and some automated result analysis.

B. Abbreviations and Acronyms

| Abbreviations | Definitions |
|---------------|----------------------------|
| PISM | Parallel Ice Sheet Model |
| DEM | Digital Elevation Map |
| JSON | JavaScript Object Notation |
| MPI | Message Passing Interface |

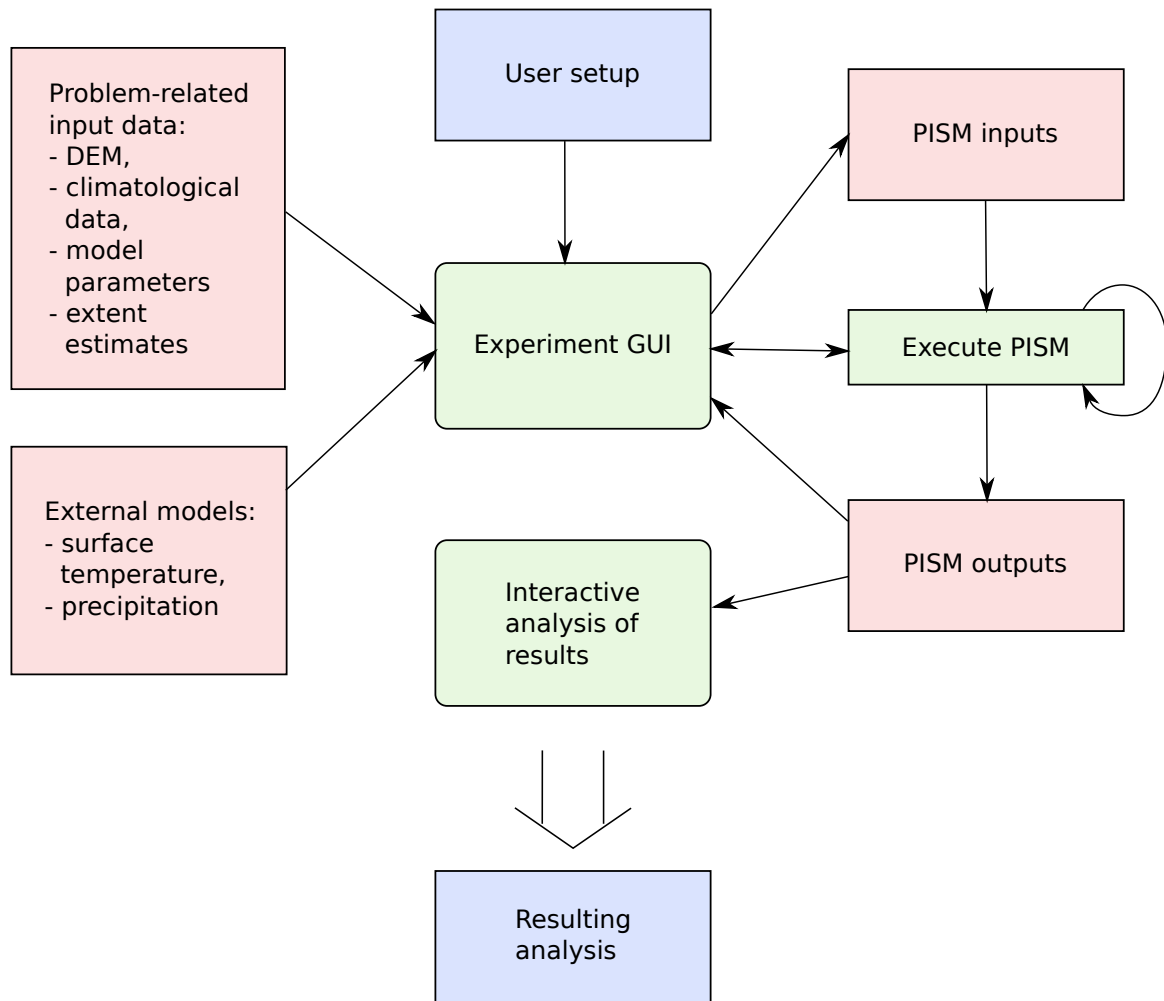


Fig. 1: The workflow of experimentation using the presented GUI. The user is aided by a selection of supported experimental parameters – the relevant subset of the PISM functionality. Several models are also implemented as Python modules and are seamlessly integrated in PISM execution.

II. METHODS

A. Hardware system

There are three different computers involved in the workflow. One is the experimenter's *personal computer*, on which they physically work and on which they interact with the GUI.

Second is the *Jupyter server* that serves the web presentation of Jupyter notebooks and executes their code locally. Note that the Python processing required for the custom (external to PISM) models are executed here. Communication with the user is done through the web server service that runs on the same computer.

Third is what we shall call the *PISM execution server*, but could in reality be a whole cluster of computers. PISM execution server is a number crunching system that only executes PISM as per the input arguments given. It communicates with the web server through the SSH protocol. Internal communication between the multiple workstations of this server is handled by MPI – a platform for distributed computing often used in scientific contexts.

B. Software system

Jupyter notebooks are documents produced by the *Jupyter Notebook App*. They contain both computer code (in our case Python) and rich text elements, organised in so called cells. Notebook documents are meant to be human-readable as well as executable. Each cell has a type - executable or rich text and can be executed or rendered on its own. A cell is executed by its *kernel* which is the “computational engine“ that executes code contained within a notebook document. For example the *ipython kernel* [5] executes Python code and *markdown kernel* [6] renders rich text cells. Kernels for many other languages also exist. Cells typically contain analysis descriptions, analysis code and results (figures, tables, multimedia). The Jupyter Notebook App is a server-client application which allows editing and running notebook documents via a web browser. It can be executed on a local desktop where it is hosted locally or it can be installed on a remote server and accessed through the internet.

Several Jupyter notebooks are used to present the GUI to the user, with the central one designed to perform the

majority of tasks: setting of the parameters, selection of the host workstation for PISM execution, submission of a job for execution, monitoring of execution, performance of automated analysis and presentation of the results to the user. A further set of Jupyter notebooks is used for development and testing of additional models, and for interactive or specialist analysis of simulation results.

To keep the interface of Jupyter notebooks clean, the majority of its source code is organised into Python source files. In addition, several Python scripts are provided that are able to bypass the GUI and allow the execution of pre-prepared experiments from the console. The main use case of such scripts is the execution of parameter sweeps for parameter studies, which require simulation of multiple experiments and an automated analysis of results. For these scripts, the experiments are also designed in the aforementioned GUI, however, their setups are stored in a JSON (JavaScript Object Notation) [7], [8] files. Python scripts read the experiment setup from JSON files, but are also able to modify several parameter values on their own, e.g. a parameter sweep script generates experiments from an experiment setup, a set of parameters and a set of values for each of the parameters.

For the GUI we leverage the high level interface elements from *Jupyter Widgets* [9] – interactive browser controls for Jupyter notebooks, which can be added to Jupyter as a Python package. The framework has two components; a package in the kernel which provides an interface for the widgets and an extension for the browser Jupyter frontend to manage Jupyter Widgets. The *ipywidgets* package provides Jupyter Widgets for the *ipython kernel*. Installing *ipywidgets* automatically installs extensions for JupyterLab and Jupyter Notebook (the *jupyterlab-widgets* and *widgetsnextension* packages). The package provides a basic and lightweight set of core form controls that use this framework. These include but are not limited to: text area, select/multiselect controls, sliders etc.

Lastly, several bash scripts are used for file synchronization and PISM execution. These are generated from templates by the Python code when PISM execution is requested, either from GUI or from scripts. When executing PISM, bash scripts redirect PISM standard outputs to temporary files and to the Python monitoring functions, providing the option of limiting PISM execution time and handling external signals to terminate execution if so requested by the user.

C. Remote PISM execution

The first panel (Execution host settings, shown in Fig 2) of the GUI is responsible for configuring the remote compute node's parameters. For ease of use and convenience the user can save to, and load the parameters from JSON files. The interface has built-in tests which check for a working SSH connection, a working PISM executable on the remote host and the existence of the specified working directory.

Fig. 2: Execution host settings presented on the GUI.

Skipped values for ['w_delta_P', 'w_delta_P_reference', 'w_delta_T', 'w_delta_T_reference', 'w_mass_balance'], since

Fig. 3: Experiment settings presented on the GUI.

After passing the built-in tests the user moves onto the second panel (Experiment settings) of the GUI. Here lie the meat and potatoes of the interface. As before, the user can conveniently load or save configuration settings from JSON files. There are a plethora of parameters that can be set here, an example is shown in Fig 3.

Once the experiment parameters have been set, the interface creates the necessary input files for PISM. To verify the set parameters, some of the more important model inputs are plotted; see Fig 4. Then a cell can be executed that causes the interface to connect to the configured compute node via SSH. It first copies input files to the compute node, then iteratively copies and executes the bash scripts (as the simulation is often comprised of

several consecutive PISM executions). Finally a script is executed to copy the results from the execution server back to the experimenter's personal computer (where the Jupyter notebook is open).

After the output files are available on the experimenter's personal computer, they can be analysed. The Notebook with GUI provides some automated analysis that can be modified according to one's needs. An example of the visualization that follows the analysis is shown in Fig 5. Researchers can quickly overview the plotted results of simulations and decide whether the experimentation is progressing well and plan further experiments accordingly.

Each PISM execution requires the use of two scripts to enable all the code required for setting up PISM, including a timeout and handles for user-requested termination. The timeout is responsible for gracefully stopping the process if it were to take longer than allowed by the user. Note that the results of the incomplete simulation are nevertheless stored and can be useful to the user. After their operation, bash scripts are deleted. During the simulation the user is presented with progress indicators, as shown in Fig 6.

D. Test deployment

The presented GUI was developed iteratively. In the first iteration, the GUI was designed, developed and used only locally on a the main developer's personal computer. This was sufficient and also very efficient since only one person was working on it. In the second iteration, as of the time of writing this article, the GUI was made available for demonstration to other researchers from the areas of computer science and geography. At this stage, in a form of a demonstration of functionality, it is best served as a single user environment to avoid the overhead of user registration. Note that multiple users can nevertheless work with the notebooks at the same time, but cannot store their changes which would be a cause of race conditions. In its final stage, it will be migrated to a multi-user Jupyter environment, assisted with version control.

The test deployment consists of two computer workstations: a Jupyter node and a compute node. Ubuntu Server 22.04 was used on both machines. Installation on the compute node includes Spack [10] and PISM. Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms. It was designed with large super computing centers in mind. It makes PISM's install a whole lot easier at the cost that it contains an outdated version of PISM in their repositories. This could of course change in the future. For our purposes this older version was fine.

On the Jupyter node, JupyterLab is installed within Docker. This gives us the advantages of *repeatability* and *ease of deployment* as Docker containers are easy to work with while they provide a predictable environment and a user-defined level of isolation between applications. Since the presented GUI has package dependencies we've created a custom Docker image using a Dockerfile based on the Jupyter Datascience Notebook. The required

Ubuntu packages to run the GUI are: `libgdal-dev`, `gdal-bin`, `python3-dev`, `build-essential` and `rsync`.

Python dependencies are also installed via the Dockerfile so that they are permanent and persistent even if the Docker container is restarted (e.g. through `docker-compose down`). Installing additional packages is possible with `pip install ...` from inside the container but the installations will not persist. The required Python dependencies are: `ipywidgets`, `netCDF4`, `numpy`, `matplotlib`, `scipy`, `shapely`, `pyproj` and `gdal`. Some of these are already provided by the Jupyter Datascience Notebook source image but are listed here for the sake of completeness. It is crucial that `gdal` from `pip` and `apt` are of the same version.

All the required code for the GUI is stored on the Jupyter node in the form of a `git` repository. This repository is setup as read-only for the Docker container because it is not meant to be modified inside it. This test deployment is configured to be synchronized with a central code repository through `git`. The repository is mounted into the Jupyter Docker container as a Docker volume at path `/home/jovyan/work/`, which is hard-coded into the official Jupyter Datascience Notebook Docker image.

Additionally we also mount the `ssh_config` and key files to their appropriate locations to enable Jupyter node connect to the compute node with non-interactive SSH sessions. In our environment an additional hop through a middle-man node was required due to our network configuration.

E. Development

The intention is that the GUI maintainer develops the GUI locally on his computer. The maintainer runs the same Jupyter notebook along with some helper notebooks that are development oriented. Changes are then synchronized across to the Jupyter node via `git` and the GUI is updated.

III. RESULTS

In this section, we shall list several features that we found invaluable during the development and use of the proposed GUI. Then we list the positive and negative aspects of the developed system.

A useful feature of PISM is its ability to stop the simulation at any time and force its unfinished result to be stored for inspection. We used this feature to implement a time limit for simulations, which we found to be very useful for experimenting with parameters. On a related note, the ability to monitor the ongoing simulation is important to experimenters and while we did implement it through Jupyter supported widgets, there is room for improvement over our approach. Using the PyQt framework could be one betterment of our approach as PyQt gives the GUI the flexibility to take on what ever form is needed, all the while not being limited by the selection of widgets in Jupyter. This all comes at a great cost though

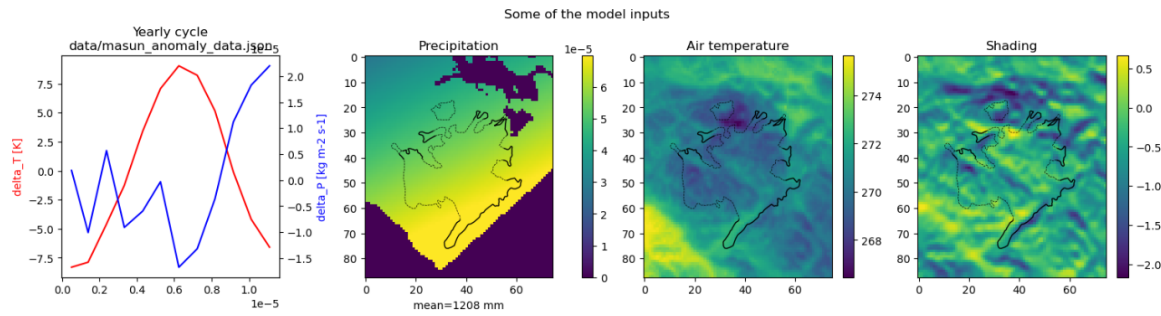


Fig. 4: An example of the overview of experiment settings, which the user can verify to match their desired settings before confirming to execute the experiment.

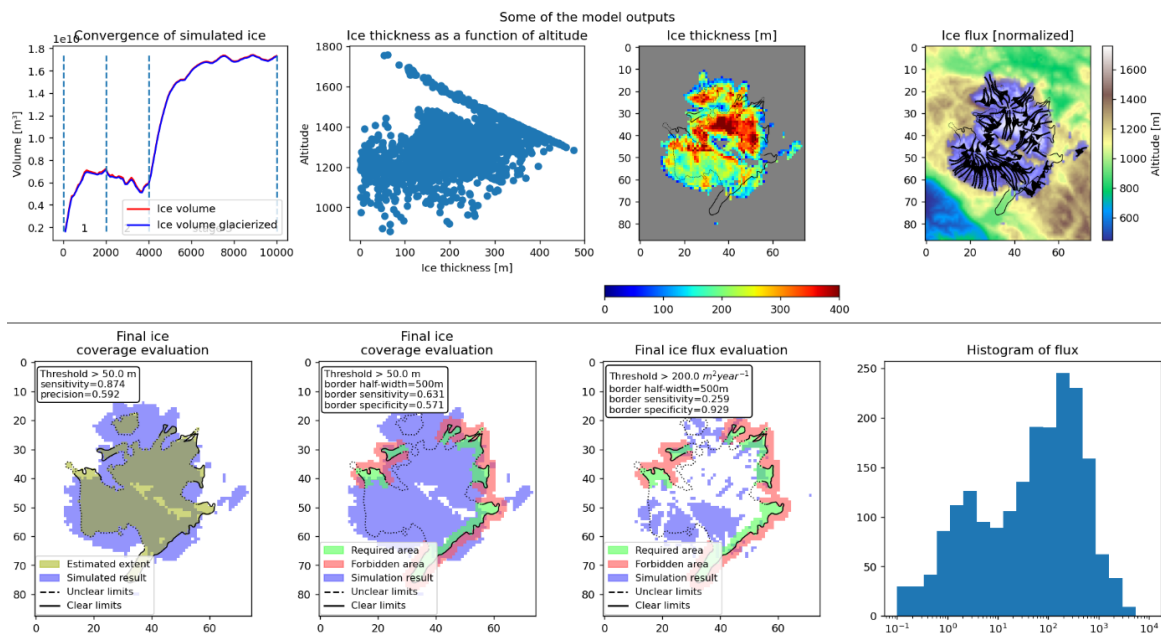


Fig. 5: An example of the visualization presented to the user as an experiment completes.

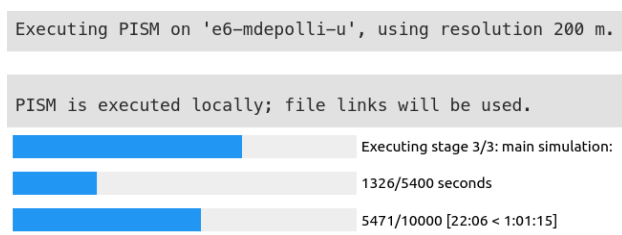


Fig. 6: While a simulation is in progress, GUI visualizes its progress to the user.

as development of a GUI in PyQt is much more complex and time consuming than using Jupyter widgets.

The most notable positive aspects of providing a custom interface towards PISM are listed below.

- Experiments are reproducible from a minimal set of input parameters.
- Experiment can comprise of an arbitrary number of consecutive PISM executions, e.g. to improve execution speed, multigrid simulations are often used,

which are presented as multiple PISM calls with careful handling of input/output files and modifying command line arguments between the consecutive calls.

- Experiments do not need to be executed locally, workstations with ample computational power can be used instead of user's laptops or PCs. This in turn enables one to design and execute experiments that run for several days due to their computational complexity.
- The presented environment is easy to learn for newcomers.
- The presented environment is fully extensible, since it primarily comprises of Python scripts and Python code within Jupyter notebooks.
- GUI is straight-forward to develop since all the required Python widgets are supported by the framework. All the required input arguments of PISM were converted into widget-based inputs, by a fast and efficient development process, facilitated by the simplicity of widgets. Note that the simplicity of

widgets can be a two-edged sword if a more complex interface is required.

The presented GUI also has its set of problems.

- Execution is slow, for example, rendering the *Experiment setup* cell can take 10 seconds.
- Working with the Jupyter-based GUI is not the most intuitive. Selected parameters in a cell that renders the GUI are not immediately available – another cell with the code for collecting the parameter values must be executed, to make them available to the rest of the notebook. Moreover, while any cell of the notebook is being executed, input widgets from all the cells are not responsive.
- The GUI is limited by the visual elements provided by the *ipywidgets* library. Storing and loading settings from a file, for example, is a user-created widget and is not as responsive as the first-time users might expect from it.
- Multi-user support is only partial on PISM execution server. Multiple users could, for example, run simulations concurrently and inadvertently write over other user's input or output files. Multiple concurrent users could also overload the server, since there is no load-balancing implemented, nor are users aware of the existing load of the PISM execution server.
- Only a subset of PISM options is supported by the GUI – those that were used within our experiments so far.
- The GUI is tightly decoupled to the PISM simulations, therefore, Jupyter notebook must be kept open for the duration of the simulations, or the results will be lost. Therefore, running complex experiments that take hours or days to complete is unrealistic in GUI.

There is room for improvement in the deployment too. We had learned of several quirks while working on the test deployment. Multi-user support could be bettered by running Jupyter notebooks in an instance of JupyterHub instead of just JupyterLab, which will be attempted in the next phase of development.

A. Generalization

Given the flexibility of Python, Jupyter and especially Jupyter widgets our approach is general enough that it could be applied to other software as well. The condition is, that such other software supports either command-line or configuration file interface – there is no sense in creating another graphical interface if one already exists. We also see no hurdles in implementing execution on a remote host over an SSH connection for other software. The interactive part of the proposed GUI implementation is not great though (slow and with limited interface), and the source code that implements it is notably more complex than the

rest. Therefore, we would not recommend our approach where real-time user interaction is required or where user interaction is complex, e.g. selection of a region on a plot, 3D rotation of plots, interaction with sound or animation.

IV. CONCLUSIONS

We present an attempt at making the interface towards a complex simulation software more manageable for experimenters and easier to learn for new users. The presented approach separates user interface from the simulation software and is even able to put it on another computer entirely. Currently, the presented GUI is meant to be only used online with the editing disabled. All the GUI development is performed on personal computers with Jupyter locally installed and is propagated to the web through `git`. There are ongoing efforts to make GUI and the external models also editable from the web interface. Although the presented GUI serves to handle PISM, the approach is general enough to be used on other software as well.

ACKNOWLEDGMENT

This work was funded by the Slovenian Research Agency, research core funding No. P2-0095 and project funding No. J1-2479.

REFERENCES

- [1] E. Bueller and J. Brown, "Shallow shelf approximation as a "sliding law" in a thermodynamically coupled ice sheet model," *J. Geophys. Res.*, vol. 114, p. F03008, 2009. [Online]. Available: <https://doi.org/10.1029/2008JF001179>
- [2] R. Winkelmann, M. A. Martin, M. Haseloff, T. Albrecht, E. Bueller, C. Khroulev, and A. Levermann, "The potsdam parallel ice sheet model (pism-pik) part 1: Model description," *The Cryosphere*, vol. 5, pp. 715–726, 2011. [Online]. Available: <https://doi.org/10.5194/tc-5-715-2011>
- [3] T. Kluyver and B. R.-K. et al., "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.
- [4] J. M. Perkel, "Why jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.
- [5] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.
- [6] J. Allaire, J. Horner, V. Marti, and N. Porte, "The markdown package: Markdown rendering for r," 2014.
- [7] F. Pezosa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th international conference on World Wide Web*, 2016, pp. 263–273.
- [8] T. Bray, "The javascript object notation (json) data interchange format," Tech. Rep., 2014.
- [9] J.-W. maintainers, "Jupyter-widgets/ipywidgets: Interactive widgets for the jupyter notebook." [Online]. Available: <https://github.com/jupyter-widgets/ipywidgets>
- [10] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral, "The Spack Package Manager: Bringing order to HPC software chaos," in *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. [Online]. Available: <http://tgamblin.github.io/pubs/spack-sc15.pdf>