Dataflow Notebooks: Encoding and Tracking Dependencies of Cells

David Koop Jay Patel
University of Massachusetts Dartmouth
{dkoop,jpatel7}@umassd.edu

Abstract

Computational notebooks have seen widespread adoption among scientists in many fields, and allow users to view interactive graphical results inline, to embed text and code together, to organize code into cells, and to selectively edit and re-execute cells. Because they allow quick and recordable analyses, they play an important role in documenting experiments. However, the reproducibility of notebooks can vary significantly due to the ordering of cells or changes in global state that affect the re-execution of those cells. In addition, in many popular notebook environments, cells are tagged with transient identifiers that change when a cell is re-executed so it is impossible to robustly reference other cells. We introduce dataflow notebooks as a method to allow users to explicitly encode dependencies between cells by adding a unique, persistent identifier to each cell and expanding in-code references to results in other cells. In these notebooks, we can both pose and answer provenance queries about dependencies between cells. This permits new notebook operations like downstream updates which, given a change to one cell, allow users to update all cells that may be impacted by the change while leaving all other cells alone. At the same time, dataflow notebooks increase reproducibility and enable greater reuse by making dependencies clear.

Keywords notebook, dataflow, reproducibility

1. Introduction

As computational notebooks have become a tool for many scientists, they have also evolved from a medium used only for scratch work to one also used for published results (e.g. (Styron and Hetland 2014; LIGO 2016)). Not only do notebooks encapsulate code, but they also may contain text (documenting hypotheses or identifying the type of analysis being run) and results, including graphical results like tables or plots. Thus, results in notebooks can be viewed without re-running the code that generated them. However, being able to re-execute that code offers the opportunity to step through past analyses, verify published results, and update a notebook to examine a different dataset. As notebooks are used to compute and document important scientific discoveries, it becomes increasingly important to be able to reproduce and reuse the code encapsulated by notebooks.

A notebook is a sequence of code and text blocks called cells. In traditional notebooks, a user executes individual code cells one at a time, going back to edit and re-execute cells as desired. This is in contrast to scripts where all code is executed at once. Currently in most notebooks, cells are not identified by a persistent, unique identifier so it is difficult to reference one cell from another. Therefore, most users use global variables to store results that will be used in other cells. However, because a variable is mutable, modifying the order of cell execution can produce different results. In addition, new cells may be later inserted between existing, already-computed cells, so it possible that the semantics of a variable change. More concretely, if all the cells of a notebook are executed in sequential order, they may produce different results than those stored from past executions (see Figure 1(i) and (ii)). Our goal is to enhance the notebook interface to encode robust dependencies between cells, enabling execution that respects these dependencies regardless of their order in the notebook.

In this paper, we introduce the dataflow notebook as an extension to the computational notebook that has persistent, unique identifiers for cells and robust methods for referencing cells. This framework provides significant benefits for computation and provenance. Specifically, in a dataflow notebook, we can dynamically execute all dependencies (other cells) of a cell while leaving unrelated cells unexecuted. At the same time, a user can ask to see and update downstream dependencies that may be affected by a change in a cell. With this structure, we gain intuitive provenance links between cells. As with dataflow workflows, the provenance of execution mirrors the encoded dependencies. Fur-

TaPP'17 June 22–23, 2017, Seattle, WA, USA

thermore, in the case of conditionally-used dependencies, we can track, at runtime, only the cells that were actually used.

2. Related Work

Some of the research in provenance paralleled the emergence of scientific workflows, and many of those systems were rooted in dataflow principles (e.g. (Freire et al. 2006; Ludäscher et al. 2006; Oinn et al. 2004)). A major reason for this pairing is that the encapsulation and abstraction provided by workflows allows a high-level representation of complex scientific tasks that led to a meaningful granularity of provenance data. Dataflows also provide significant benefits for related deterministic computations, allowing intermediate results to be cached and reused in the future (Freire et al. 2006). Furthermore, dataflows often produce provenance graphs that have strong correspondences to the dataflow graphs, mapping dependencies between steps and data items. The computational modules in dataflows mirror the cells in notebooks except they are usually predefined and harder to modify.

More recently, many computational scientists have embraced high-level programming as a part of their analysis workflows. To flexibly integrate their own experiments, it is often useful for scientists to build custom pieces of code that interact with the many libraries such languages provide. Tools like noWorkflow seek to capture provenance in scripts by tracking data dependencies and using program slicing (Murta et al. 2014). YesWorkflow recognized that such tools can still suffer from a mismatch between the abstraction desired by the a user and that encoded in the code, and enabled dataflow-like markup (McPhillips et al. 2015). At the same time, development of tools like Jupyter Notebook allowed users to integrate text, code, and graphical results into a single browser window (Jupyter). The cells in notebooks play a similar role to the blocks YesWorkflow delineates but use the cell boundaries to implicitly define these blocks.

There exist a variety of different computational notebook environments (Wolfram Research, Inc. a; Stein et al. 2017; apache; beaker; North et al. 2015), all of which use text and code cells with computational results shown inline. Generally, these environments serve to mimic paper notebooks that document a scientist's work and include text, computations, and visualizations. When persisted, they provide a record of an analysis and any discoveries. Some environments have machinery to enable interoperation of different programming languages, allowing users to share data between execution kernels. In addition, Mathematica has a Dynamic function that updates expressions in cells when a user reassigns a variable's value (Wolfram Research, Inc. b). Zeppelin also allows cells to watch variable changes other cells, updating results via the Angular environment. However, these solutions tend to be focused on variables not the cells themselves. Dataflow notebooks establish cells, not variables, as the focus.

3. Jupyter Notebook

A *computational notebook* is a collection of ordered cells where each *cell* encapsulates text, input code, and resulting outputs. While the ability to mix writing and code is an important feature, we will focus on the cells that encapsulate code. A *code cell* is a cell that contains both input–code to be executed–and optional output–the result of executing that code. The scope of a cell may be local or global: in most implementations, variables created in a cell are accessible in another cell, provided the former cell has already been executed. Cells may be grouped (e.g. in Mathematica) and reordered (e.g. by moving one cell ahead of another). The user chooses both which cells are executed and the order of their execution. Furthermore, a cell may be edited and rerun, if, for example, a bug is found during inspection of a cell's output.

Often, the notebook is an interface to a separate computational engine; the notebook is not itself performing the computations defined in the code cells but rather sends them to a computational kernel which returns results to be displayed in the interface. Generally, cells are executed one at a time, although parallelization may be used to schedule multiple cell runs provided they are independent. Each cell is assigned an identifier when it is created or executed; most notebooks use an incremented integer.

3.1 Environment

While the ideas outlined in this paper apply to other notebook environments, we focus on the Jupyter environment as it is open-source and has seen widespread adoption among scientists in many fields. The Jupyter environment supports different computational kernels including Python, R, and Julia, in a consistent interface across those languages. Each code cell is identified by an integer which is (re-)assigned whenever a cell is (re-)executed. The notebook serves as an interface to a computational kernel which executes all code and returns the results. Jupyter's notebook interface provides some helpful mechanisms including tab-completion of variables or function names as well as interactive widgets that can help users explore a particular result. In Jupyter, the notebook stores text, input code, and output results in JavaScript Object Notation (JSON).

We will further focus on the IPython kernel which was the first kernel for Jupyter Notebook. IPython is an extension of Python that provides an environment for rich interactive computing (Pérez and Granger 2007). The kernel provides a number of features that are also available in the terminal or console interfaces, including the ability to reference results from previous computations and view the history of past executions. Both the notebook and console environments show a prompt indicating how one references a

(i)

```
In [1]: 7 + 11
Out[1]: 18

In [2]: Out[1] - 4
Out[2]: 14

In [3]: Out[1] + Out[2]
Out[3]: 32
```

(ii)

```
In [1]: 7 + 11
Out[1]: 18

In [2]: _ - 4
Out[2]: 14

In [3]: __ + _
Out[3]: 32
```

(iii)

```
In [3c51a9]: 7 + 11
Out[3c51a9]: 18

In [eb7c2c]: Out[3c51a9] - 4
Out[eb7c2c]: 14

In [747895]: Out[3c51a9] + Out[eb7c2c]
Out[747895]: 32
```

(a) Initial notebooks

```
In [1]: 7 + 11
Out[1]: 18

In [4]: Out[1] - 14
Out[4]: 4

In [5]: Out[1] + Out[2]
Out[5]: 32
```

```
In [1]: 7 + 11
Out[1]: 18

In [4]: _ - 14
Out[4]: 18

In [5]: __ + _
Out[5]: 50
```

```
In [3c51a9]: 7 + 11
Out[3c51a9]: 18

In [eb7c2c]: Out[3c51a9] - 14
Out[eb7c2c]: 4

In [747895]: Out[3c51a9] + Out[eb7c2c]
Out[747895]: 22
```

(b) The notebooks from (a) after changing 4 to 14 in the middle cell and re-executing only the bottom two cells

```
In [1]: 7 + 11
Out[1]: 18

In [2]: Out[1] - 14
Out[2]: 4

In [3]: Out[1] + Out[2]
Out[3]: 22
```

```
In [1]: 7 + 11
Out[1]: 18

In [2]: _ - 14
Out[2]: 4

In [3]: __ + _
Out[3]: 22
```

```
In [3c51a9]: 7 + 11
Out[3c51a9]: 18

In [eb7c2c]: Out[3c51a9] - 14
Out[eb7c2c]: 4

In [747895]: Out[3c51a9] + Out[eb7c2c]
Out[747895]: 22
```

(c) The same notebooks as (b) but in a new session with the cells executed sequentially

Figure 1. Jupyter Notebook produces inconsistent results ((b) vs. (c)) for the same notebook when using cell identifiers (i) or relative references (ii). Dataflow notebooks use persistent identifiers to produce consistent results (iii).

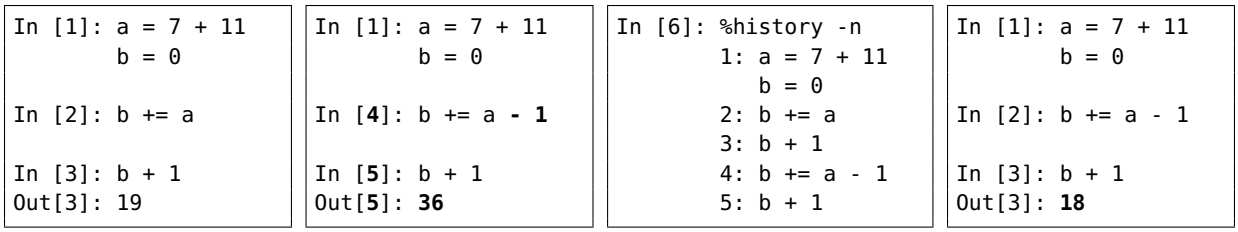
computation's output: `Out[`N`]` where N is the integer identifier of the cell. However, note that identifier for each cell is assigned after execution, and thus, it may change if a cell is re-executed. Furthermore, whenever a user restarts the kernel (e.g. in a new session), the identifier counter is reset to one. IPython also grants access to the most recently generated results via the special underscore variables `_`, `__`, and `___`. Finally, IPython provides *magic* commands that add functionality outside of the normal Python interpreter. For example, `%history` lists, in order, all of the code that has been executed (see Figure 2(c)).

3.2 Current Practices

In order to understand how cells are currently referenced in Jupyter Notebook and the errors that may result, we analyzed existing notebooks made available via a notebook gallery at `nb.bianp.net`. There, we searched for uses of the `Out` dictionary and underscore references. There were very few uses of these references with most users opting to use global variables to reference results computed in other cells. In addition, in a manual examination of some of the notebooks, we found that many were designed to be executed top-to-bottom as they were published tutorials that may not necessarily reflect raw analyses. However, when reviewing a recent Jupyter user study (Santilli et al. 2016), we found a number of users concerned with the *order* of cells. Specific comments included responses to the question "What aspects of Jupyter Notebook make it difficult to use in your workflow?" that stated "Out-of-order execution" and "Easy to get cells out of place in workflow order". In addition, there were feature and enhancement requests that asked for an "Easier way to see execution order of cells which have run out of order sometimes" and an "Easier way to reorder cells...according to how I executed them".

3.3 Reproducibility

Defining reproducibility for a notebook is complicated because the user controls which cells are executed and in what



(a) Initial Notebook (b) Cell Updates (c) History (d) Re-execution

Figure 2. When global variables are used, Jupyter Notebook may produce inconsistent results as cells are re-executed. A user creates an initial notebook (a), then changes the middle cell and re-executes the bottom two cells (b). If the user saves the notebook and later opens it and re-executes all cells (d), the final result is different. The history (c) of all executed code in (a) and (b) shows how this issue arises as `b` is incremented both times the middle cell is executed.

order. A common convention, supported by Jupyter's "Run All" command, is to assume all cells should be executed in the order they appear (top-down). Then, a notebook is *reproducible* if a re-execution of the cells in sequential order matches the stored results. However, this top-down convention often defies normal usage as a user often returns to a cell, edits it, and re-executes only that cell; re-running all the cells that haven't changed is inefficient. That non-linear history of execution often leads to cases where the results in the saved notebook do not match the reproduced results.

In Figure 1, notebooks (i) and (ii) show the results obtained when the middle cell is updated and the bottom two cells re-executed (b). If a user, satisfied with those results, saved that notebook, opened it in a new session, and re-executed it, that user would obtain *different* results (c). In notebook (i), the `Out[2]` reference in the last cell in (b) recalls the original result in (a) but stores an updated value in (c). In notebook (ii), `_` in the middle cell references `Out[3]` in (b) since it was the last cell executed but references `Out[1]` in (a) and (c). These issues also affect the more widely used method of storing intermediate results in global variables. Figure 2 shows how inconsistent results are generated when a variable (`b`) is mutated as a cell is re-executed.

4. Dataflow Notebooks

Dataflow notebooks introduce a new unique, persistent identifier for cells and use these robust references to enable recursive execution of upstream cells, helping to ensure the consistency of results. The framework encourages users to reference intermediate results directly rather than storing them in global variables. This adds greater significance to cell boundaries, building on a concept that notebook users utilize to organize their analyses. We extend the current method of referencing past results to allow recursive updates if changes have occurred. Finally, we have updated the existing notebook interface to show users what dependencies exist and communicate when those dependencies are executed.

A *dataflow* organizes computation into modules, each of which has a set of inputs and a set of outputs, and connections, which connect outputs of one module to inputs of another. The graph formed by these modules and connections is acyclic. Then, a module A is *upstream* of another B when there exists a directed path from A to B; B is also then *downstream* of A. Then, a module may be executed when all of its upstream modules are up to date. In dataflow notebooks, cells are the modules and the connections are implicitly defined by the references to other cells in a cell.

In a dataflow notebook, an identifier is assigned to a cell when it is created. This identifier is unique and persistent, no matter how the cell changes. Thus, one can fix bugs or update a computation without worrying about any references to that cell becoming stale as with Jupyter notebooks. These identifiers also persist across sessions. We chose the UUID scheme (Leach et al. 2005) to allow cells to be reordered and reinforce the idea that a notebook need not be read or executed in the linear top-down fashion, but other schemes are possible. Referencing a cell's output is similar to the current method in Jupyter Notebook but with the new identifiers (`Out[3c51a9]`) instead of numbers (`Out[1]`). Note that the new identifier may be stored as an integer but represented as a string. Figure 1(c) shows that the dataflow notebook version is reproducible.

With respect to implementation, dataflow notebooks introduce a number of changes to the kernel and the notebook interface. In the kernel, we expand the utility of the `Out` dictionary to allow it to *dynamically* compute its upstream dependencies. To enable this, the kernel stores all of the code (by cell). Thus, the notebook pushes *all* code cells that have changed since the last execution to the kernel in case a cell was edited but not executed. Then, any `Out` access checks the referenced cell to see if it or any of its dependencies need to be (re-)executed. If there was an update, it recursively updates and re-executes the cell. If no changes were made, it pulls the previously computed output from a cache just as the original `Out` cached outputs. Note that we need to store a dependency graph and check *all* dependencies to

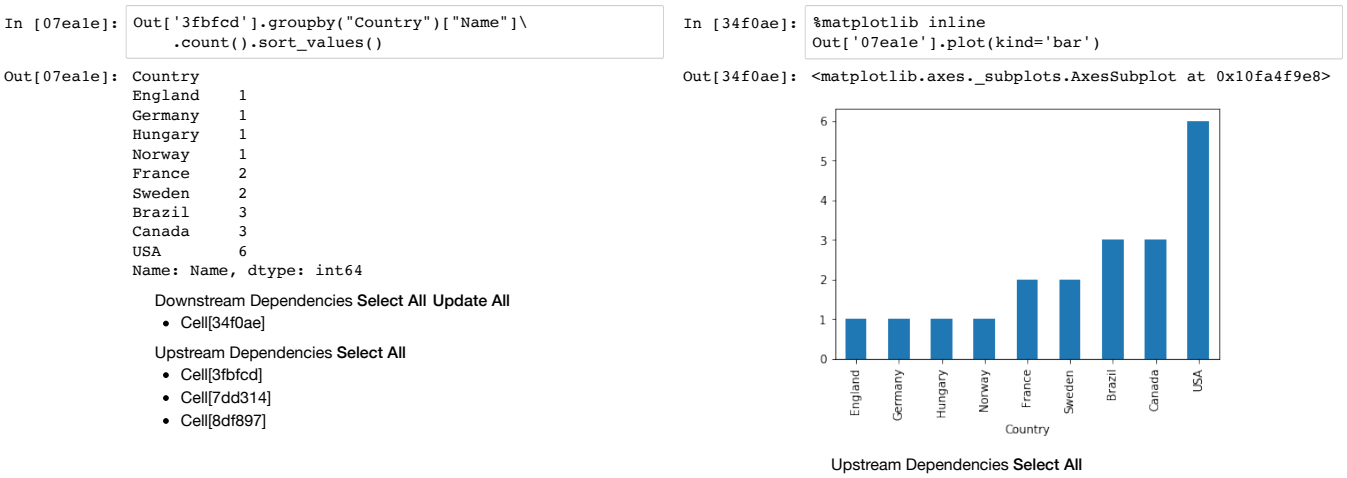


Figure 3. A view of the dataflow notebook interface. After adding the `sort_values` call to the cell shown on the left, we can both view and choose to update the downstream dependencies (cell on the right) from the left cell.

ensure we recompute when necessary. Also note that while a modification in one cell may change upstream dependencies, these will be discovered as we recurse so computation is not wasted.

A key concern in dataflow notebooks was to make sure the user was aware of all of the computation taking place by updating the interface accordingly. While Jupyter allows the notebook to trigger sequences of cell executions, the kernel could not dictate new executions to the notebook. In addition, the notebook interface assumed all output streams, errors, and results belonged to the currently executing cell, a rule that breaks when the kernel directs new executions. To address these issues, we modified the kernel to issue new messages that both informed the notebook of dependent executions and their results and also tagged each message with the unique cell id. This allowed the notebook interface to provide feedback to the user about dependent cells that were run, show results as they were computed, and tag errors with the cells that contained them. Thus, the executed notebook looks no different than if one had executed each cell individually in dataflow order.

In addition, we have updated the interface to add two entries to the bottom of each cell output to allow inspection and potential execution of upstream and downstream dependencies. Specifically, the *Upstream Dependencies* entry provides a clickable link to highlight all upstream cells, and a user may expand this item to show the identifiers of each of those cells, with clickable links. The *Downstream Dependencies* entry is similar but also includes a link to execute all downstream dependencies. In Figure 3, a user updated the list of countries to be sorted, and then executed the downstream cells by clicking on the "Update All" button (regardless of where in the notebook that cell is). Because users may wish to step through downstream cells or make other changes before executing everything, we do not attempt to execute all downstream cells automatically. However, we do enforce that upstream cells that have changed are updated (during execution) to ensure consistency.

5. Provenance

The provenance of dataflow notebooks is very similar to dataflow provenance, and because cell references are followed dynamically, we can capture some finer-grain dependencies. Our focus is on the higher-level provenance at the cell level, enabling an abstracted understanding of the steps in the analyses. Using existing techniques, e.g. those from noWorkflow, we can capture data being read or written, and compute which cells depend on particular inputs or contributed to particular outputs. The addition is the ability to reason about the provenance at the level of the cell akin to the modules in scientific workflows or the groupings that tools like YesWorkflow allow. We can augment the cell-based provenance with more detailed provenance that tools like noWorkflow provide (Pimentel et al. 2015), but this is not required. In addition, the code and the references are constructive and discoverable upon *execution* while YesWorkflow blocks require annotation.

It is useful to understand what provenance can be gleaned from the current Jupyter notebook. As described earlier, the history command in IPython allows a user to see all of the code executed in a session, providing a window into how a particular result was generated. However, matching this code listing with the current state of the notebook may be very difficult because cells may have been edited, moved, or deleted. Some of this history may be relevant as in the example in Figure 2, but there may also be entries, for example those generated when a user fixed a bug in a single cell, that do not impact any stored result. In some cases, it my be possible to derive a putative order of cell executions,

but finding dependencies between cells is also problematic without the robust identifiers in dataflow notebooks.

In dataflow notebooks, the used and wasGeneratedBy provenance relations can be determined as cells are executed in notebook. Specifically, for any execution of a cell, we can record all cells that were directly referenced, generating the proper relations between those cell outputs and the current cell. The cell outputs are entities while the executions of the cells are activities. With additional information, like the hooks that noWorkflow uses to recognize file reads and writes, we can also construct data dependencies. It is important to note that while the provenance is cell-based, the upstream cells a particular cell references need not include every reference in the code. If a cell contains a conditional whose branches reference different cells, the provenance will reflect the cells actually used.

6. Discussion and Future Work

This work addresses issues in computational notebooks with a goal of developing a notebook that is more reproducible and reusable. However, it largely ignores the *versioning* question in notebooks; given a result, what version of the notebook, including the code in the cells, generated that result. For provenance, versioning cells would add meaningful data about past executions and results while versioning notebooks grants information about past *orderings* of cells. Note that both of these are enabled by persistent, unique identifiers for cells. For cell differences, we may miss some cases where cell text was copied from one cell to another, but these additional checks could be added via extra heuristics.

There has been some work both to examine differences between cells and version notebooks. Specifically, nbdime has sought to add more useful differencing mechanisms to Jupyter notebooks, allowing changes to cells in the notebook's JSON format to be more meaningfully understood (nbdime). Note that nbdime (version 0.2.0) currently does not try to determine if cells have been reordered. RCloud, a collaborative notebook environment for R, automatically creates a new version of the notebook whenever a command is run (North et al. 2015), but it does not worry about the dependencies between cells and thus their order.

Another issue with notebooks is that while they contain code and results, the data a notebook references may or may not be preserved with it. General solutions like Docker or ReproZip (Chirigati et al. 2016) encapsulate dependencies and may also include data dependencies which would help if notebooks are used by different people. Solutions like mybinder also aid in notebook-specific solutions for the dependency problem. Integrating noWorkflow's file detectors at the kernel level could help add this information to the generated provenance.

In dataflows, a cell may generate multiple outputs and other cells may use any or all of them. With the notebook, there is single result per cell. However, bundling multiple results is possible by encapsulating them in a named tuple or associative array. Then, one might reference a particular variable `a` in cell `7fe4bc` as `Out[7fe4bc].a`. One way to accomplish this more generally is via an IPython magic command that specifies the variables to be made available as output (e.g. `%%output a`).

A potential issue lies with scoping and references to past computations. Specifically, strict encapsulation might keep all variables local to their respective cells, but this would be annoying to users who have libraries or functions they wish to import or define *once* and use in multiple cells. At the same time, the possibility of a downstream cell mutating a past output could cause serious issues. However, note that this situation is *no worse* than the current state; if a variable is used in multiple cells and mutated in one, that may affect a computation in the other. In such cases, some type of copy-on-write would be useful and may be more easily enforced in dataflow notebooks.

7. Conclusion

This paper introduces dataflow notebooks as an environment where cells become more than boundaries between snippets of code; their results can be chained to other cells in a robust manner. This enables dataflow execution of cells, providing higher-level provenance and allowing users to be better aware of the dependencies, both upstream and downstream, of each cell. In the future, we plan to work on integrating version control features and evaluate the usability of the dataflow notebook.

References

apache. Apache Zeppelin. `http://zeppelin.apache.org`.

beaker. Beaker Notebook. `http://beakernotebook.com`.

F. Chirigati, R. Rampin, D. Shasha, and J. Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2085–2088. ACM, 2016.

J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop*, pages 10–18. Springer, 2006.

Jupyter. Jupyter notebook. URL `http://jupyter.org`.

Laser Interferometer Gravitational-Wave Observatory (LIGO). Signal processing with GW150914 open data. `https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html`, 2016.

P. Leach, H. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, 2005. URL `https://www.ietf.org/rfc/rfc4122.txt`.

B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, , F. Chirigati, S. Dey, J. Freire, D. Huntzinger, C. Jones, D. Koop, P. Missier, M. Schildhauer, C. Schwalm, Y. Wei, J. Cheney, M. Bieda, and B. Ludaescher. Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts. *International Journal of Digital Curation*, 10(1):298–313, 2015.

L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*, pages 71–83. Springer, 2014.

nbdime. nbdime: Jupyter notebook diff and merge tools. `https://github.com/jupyter/nbdime`.

S. North, C. Scheidegger, S. Urbanek, and G. Woodhull. Collaborative visual analysis with rcloud. In *Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on*, pages 25–32. IEEE, 2015.

T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL `http://ipython.org`.

J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire. Collecting and analyzing provenance on interactive notebooks: When ipython meets noworkflow. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, Edinburgh, Scotland, 2015. USENIX Association. URL `https://www.usenix.org/conference/tapp15/workshop-program/presentation/pimentel`.

J. Santilli, P. Parente, B. Granger, F. Pérez, J. Grout, B. Ragan-Kelley, and M. Bussonnier. Jupyter notebook UX survey. `https://github.com/jupyter/design/tree/master/surveys/2015-notebook-ux`, 2016.

W. A. Stein et al. *SageMath, the Sage Mathematics Software System*. The Sage Development Team, 2017. `http://www.sagemath.org`.

R. H. Styron and E. A. Hetland. Estimated likelihood of observing a large earthquake on a continental low-angle normal fault and implications for low-angle normal fault activity. *Geophysical Research Letters*, 41(7):2342–2350, 2014. ISSN 1944-8007. doi: 10.1002/2014GL059335. URL `http://dx.doi.org/10.1002/2014GL059335`. Notebook version: `https://github.com/cossatot/lanf_earthquake_likelihood/blob/master/notebooks/lanf_manuscript_notebook.ipynb`.

Wolfram Research, Inc. Mathematica. `https://www.wolfram.com/mathematica/`, a.

Wolfram Research, Inc. Introduction to dynamic. `http://reference.wolfram.com/language/tutorial/IntroductionToDynamic.html`, b.