# Computational Notebooks in Public Repositories

Speicher, Daniel and Cremers, Armin B.

**Abstract:** ***A computational notebook – most prominently a Jupyter notebook – is a special kind of a document that encompasses data, text, calculations, visualizations. Public repositories – most prominently GitHub – make them available. The number of notebooks is growing almost exponentially as more and more software developers, data scientists and machine learning students, teachers and engineers share their notebooks publicly. They use these repositories not only to store own notebooks but to find solutions to their specific problems at hand. The open exchange of computational narratives in Jupyter notebooks is an overwhelming success story. In this paper we ask the question of how the use and spread of publicly available notebooks affect the quality of code and its embedding in a computational narrative. Drawing on empirical studies and our own experiences in the creation of digital material to support machine learning education and our observation of students' use of notebooks, we comment on the quality. Our findings include that we cannot just reuse the concrete quality criteria that are in use for software in general. Much rather, we must integrate the competing demands in creating a linear computational narrative, thus adapting programming style and design patterns to the data science context.***

**Index Terms:** ***Code quality, Computational notebook, Education, Explanation, Exploration, Machine Learning, Project Jupyter, Public repository, Reproducibility***

## 1. INTRODUCTION

Computational notebooks have an overwhelming success story. The number of publicly available Jupyter notebooks on GitHub has grown exponentially in the recent years (see Figure 1). Perkel describes in his toolbox column in Springer Nature [2], under the title "By Jupyter, it all makes sense", the success story. The widespread use of Jupyter notebooks and the public discussions through scientific articles, conferences, and social media, created a broad awareness of their opportunities as well as the relevant limitations in tools and practices. Here we share in the endeavor of different research groups, who have taken on these limitations and are pushing the boundaries. Particular incentives come from the use of notebooks in data science and machine learning.

A "Jupyter Notebook is [a] web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text." (https://jupyter.org/) The content of code cells is sent on demand to a Python session (called "kernel"), executed, and the output inserted below the cell.

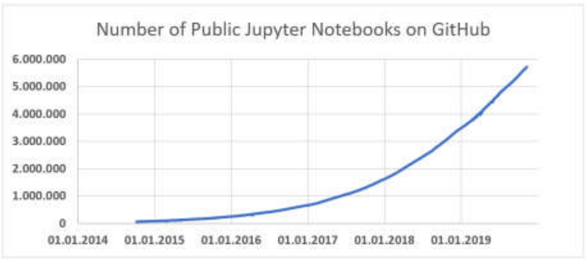


**Figure 1: Number of Notebooks.** The number of public Jupyter Notebooks on Github. according to the nbestimate project.

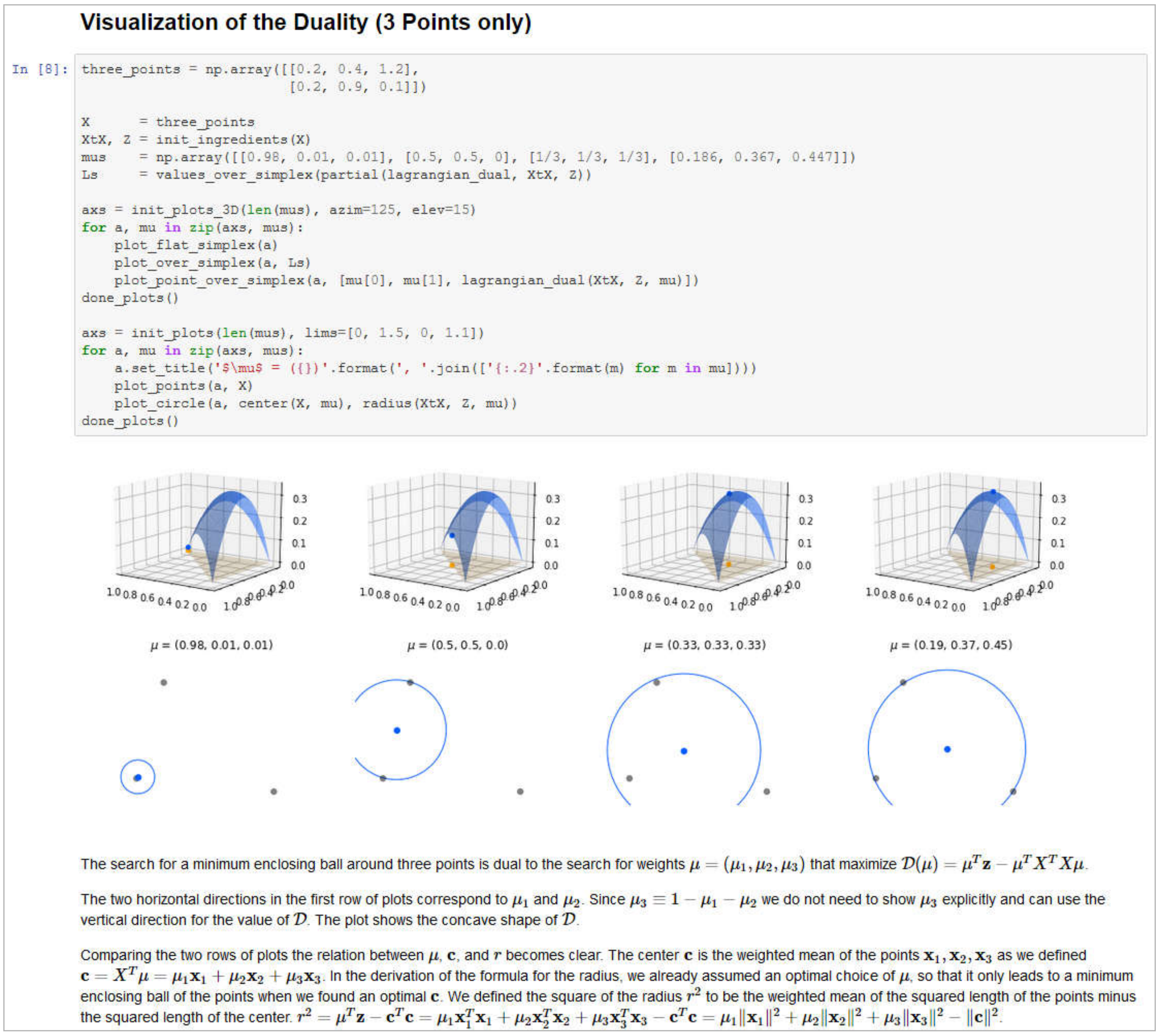
To provide an overall impression of a notebook, Figure 2 shows a screenshot of a larger part of a Jupyter notebook. We see a headline (in a Markdown cell), some code (in a code cell), two visualizations as result of the code, and some explanatory text containing even formulas written in LaTeX (again in a Markdown cell). The two rows of visualizations illustrate here the duality of two problems. The upper row shows a regular optimization problem while the lower row shows the minimum enclosing ball problem.

To illustrate the value of having LaTeX and actual Python code close to each other, Figure 3 shows another part of the same notebook. Having the formulas and the implementation so close to each other, makes it easy to check whether the code indeed implements the formulas

While GitHub offers the means to share and evolve the file, it needs another service called

Manuscript received November 18, 2019. This material was prepared within the project P3ML which is funded by the Ministry of Education and Research of Germany (BMBF) under grant number 01/S17064. The authors gratefully acknowledge this support.

A. B. Cremers and D. Speicher are with the Bonn-Aachen International Center for Information Technology, University of Bonn, Germany (e-mail: abc@bit.uni-bonn.de, dsp@bit.uni-bonn.de)



## Visualization of the Duality (3 Points only)

```
In [8]: three_points = np.array([[0.2, 0.4, 1.2],
                                 [0.2, 0.9, 0.1]])

        X      = three_points
        XtX, Z = init_ingredients(X)
        mus    = np.array([[0.98, 0.01, 0.01], [0.5, 0.5, 0], [1/3, 1/3, 1/3], [0.186, 0.367, 0.447]])
        Ls     = values_over_simplex(partial(lagrangian_dual, XtX, Z))

        axs = init_plots_3D(len(mus), azim=125, elev=15)
        for a, mu in zip(axs, mus):
            plot_flat_simplex(a)
            plot_over_simplex(a, Ls)
            plot_point_over_simplex(a, [mu[0], mu[1], lagrangian_dual(XtX, Z, mu)])
        done_plots()

        axs = init_plots(len(mus), lims=[0, 1.5, 0, 1.1])
        for a, mu in zip(axs, mus):
            a.set_title('$\mu$ = ({})'.format(', '.join(['{:.2}'.format(m) for m in mu])))
            plot_points(a, X)
            plot_circle(a, center(X, mu), radius(XtX, Z, mu))
        done_plots()
```

The search for a minimum enclosing ball around three points is dual to the search for weights $\mu = (\mu_1, \mu_2, \mu_3)$ that maximize $\mathcal{D}(\mu) = \mu^T \mathbf{z} - \mu^T X^T X \mu$.

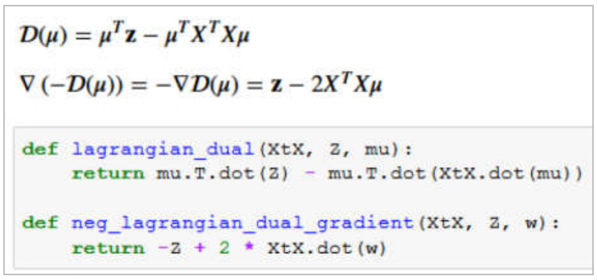
The two horizontal directions in the first row of plots correspond to $\mu_1$ and $\mu_2$. Since $\mu_3 \equiv 1 - \mu_1 - \mu_2$ we do not need to show $\mu_3$ explicitly and can use the vertical direction for the value of $\mathcal{D}$. The plot shows the concave shape of $\mathcal{D}$.

Comparing the two rows of plots the relation between $\mu$, $\mathbf{c}$, and $r$ becomes clear. The center $\mathbf{c}$ is the weighted mean of the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ as we defined $\mathbf{c} = X^T \mu = \mu_1 \mathbf{x}_1 + \mu_2 \mathbf{x}_2 + \mu_3 \mathbf{x}_3$. In the derivation of the formula for the radius, we already assumed an optimal choice of $\mu$, so that it only leads to a minimum enclosing ball of the points when we found an optimal $\mathbf{c}$. We defined the square of the radius $r^2$ to be the weighted mean of the squared length of the points minus the squared length of the center. $r^2 = \mu^T \mathbf{z} - \mathbf{c}^T \mathbf{c} = \mu_1 \mathbf{x}_1^T \mathbf{x}_1 + \mu_2 \mathbf{x}_2^T \mathbf{x}_2 + \mu_3 \mathbf{x}_3^T \mathbf{x}_3 - \mathbf{c}^T \mathbf{c} = \mu_1 \|\mathbf{x}_1\|^2 + \mu_2 \|\mathbf{x}_2\|^2 + \mu_3 \|\mathbf{x}_3\|^2 - \|\mathbf{c}\|^2$.

**Figure 2 Jupyter Notebook.** The Minimum Enclosing Ball Problem and a dual optimization problem. We show it here to give an impression of the different elements of a Jupyter notebook. (The notebooks is available online.)

nbviewer to display all aspects of the notebook correctly and it needs a third service called binder to use the notebook interactively. (See our notebook online on github, nbviewer and binder.)

## 2. PRELIMINARIES



$$\mathcal{D}(\mu) = \mu^T \mathbf{z} - \mu^T X^T X \mu$$

$$\nabla(-\mathcal{D}(\mu)) = -\nabla \mathcal{D}(\mu) = \mathbf{z} - 2X^T X \mu$$

```
def lagrangian_dual(XtX, Z, mu):
    return mu.T.dot(Z) - mu.T.dot(XtX.dot(mu))

def neg_lagrangian_dual_gradient(XtX, Z, w):
    return -Z + 2 * XtX.dot(w)
```

**Figure 3: LaTeX and code.** The math and the code do not translate immediately into each other, but the reader may infer their relation from comparing both with each other

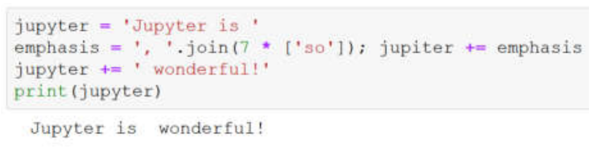
There is one challenge in working with Jupyter notebooks that is worth mentioning upfront, and that is the impact of the execution order. It does not become apparent unless one observes how developers interact with notebooks. It is the strongest aspect of the criticism of notebooks in Grus' widely recognized presentation [3] and proved quite relevant in the empirical study that we will summarize in 3.2.

The result of a notebook depends on the execution order and may be impacted by hidden state. Let us illustrate this. A developer may have written the following four lines of code and executed the cell:

```
jupiter = 'Jupiter is '
emphasis = ', '.join(7 * ['so']); jupiter += emphasis
jupiter += ' wonderful!'
print(jupiter)
```

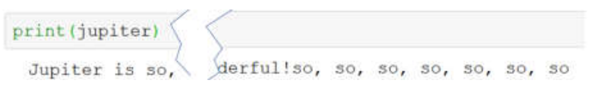
```
Jupiter is so, so, so, so, so, so, so wonderful!
```

After the successful execution of the cell, the developer realizes that she used the name of the planet but not of the notebook and changes the variable name and a string constant as seen below. Executing the cell gives an unexpected result:


```
jupyter = 'Jupyter is '
emphasis = ', '.join(7 * ['so']); jupiter += emphasis
jupyter += ' wonderful!'
print(jupyter)
```

```
Jupyter is  wonderful!
```

What did happen? The result of the cell should have been the same. Just instead of “Jupiter” we wanted to see “Jupyter”. The developer had overlooked one occurrence of the variable `jupiter` in the second line. The mental model of the developer, the code read linearly, and the state of the Python process got out of sync. a) The developer thought there was just the one variable `jupyter`. b) The code shows a variable `jupyter` that is initialized in the first line and a variable `jupiter` that is *not initialized above* the place where it is read. c) The Python process knows a value of a variable `jupiter` that was *set in an earlier* execution:


```
print(jupiter)
```

```
Jupiter is so,
derful!so, so, so, so, so, so, so
```

Note how easily this error happens through well motivated changes to the code. The impact can obviously be substantial, and if we work with more complex data types like matrices, it is much harder for the developer to detect mistakes.

## 3. Empirical Studies

Two large empirical studies analyzed the publicly available Jupyter notebooks on GitHub.

### 3.1 Notebooks for Exploration and Explanation

Rule, Tabard and Hollan [12] analyzed approximately 1.25 million Jupyter notebooks retrieved from public repositories on GitHub in July 2017. These notebooks cover about 95% of the publicly available notebooks in the nearly 200,000 public GitHub repositories containing at least one notebook. The primary intention of the authors was to understand how people create narrative texts in notebooks to explain their findings. Nevertheless, they found that, leaving aside the quarter of the notebooks having no text at all, half of the remaining notebooks had less than 218 words. The authors suggest, as reason for the low amount of text, that many researchers “view their notebooks as personal and messy works-in-progress”. However, some (unfortunately the paper does give a more precise number here) of the notebooks with substantial text were judged to be “truly remarkable in the way they elegantly explain complex analyses”.

Notebooks thus allow for at least two substantially different uses. Fast feedback and the visible record of results are helpful to explore a problem and search for good solutions. An exploratory workflow nevertheless does not automatically lead to a notebook that effectively communicates the structure of the problem and of the solution. Creating a notebook that elaborates an explanation, requires additional effort and needs to take the intended audience into account. The step from exploration to explanation is substantial. Only recently some tooling for this transition became available, as we will describe below in 5.2.

From a programming perspective it is remarkable that little effort was made to modularize the code: Only 37.3% of the notebooks defined a function and only 12.3% defined a class. At least, 62.1% had comments in the code.

### 3.2 Reproducibility in Notebooks

One year later Pimentel, Murta, Braganholo and Freire [9] conducted a similar study with a stronger focus on reproducibility. This study, started out with 1.45 million notebooks created between January 2013 and mid of April 2018, found in them over 265,000 public GitHub repositories with at least one Jupyter Notebook. After filtering invalid, empty, or duplicated notebook files 1.16 million notebooks were left. The duplication detection compared only text and code and ignored metadata and results. A cloned notebook that had been executed again and delivered different results, but had otherwise not been changed, was thus considered to be still the same notebook. A case that is especially relevant for notebooks that are cloned by many students of the same course. Focusing on notebooks with Python code reduced the number to 1.08 million.

The study found similar results as [12] with respect to the distribution of notebooks to repositories, use of Markdown, definition of functions, and classes. Beyond the shared observations, it reports the following further insights: Users had chosen overall meaningful and long names for their notebooks. Judging by the imported modules, 1.54% of the valid Python notebooks contain tests. Over 85% of the notebooks did store some test results during their lifetime. However, more than about a million notebooks examined show that tests as such were done as was hoped in the right order, but often with skipped cells or out-of-order answers with the tests.

The authors attempted to execute the notebooks and to reproduce their results. The most common reason for execution failures were missing dependencies either because the dependencies declared in a `requirements.txt`, `setup.py` or `Pipfile` could not be installed or none of these configuration files were given. In the latter case, notebooks were tested in a full Anaconda environment. In total, executability of almost 789,000 notebooks was tested. 24.11% of the runs were successful but only 4.03% produced the same results.

The reasons for failures were missing dependencies (29.23%: `ImportError`, `ModuleNotFoundError`), presumably out-of-order execution and hidden state (14.53%: `NameError`), missing data or absolute file paths (12.59%: `FileNotFoundError`, `IOError`).

These problems may be avoided by following certain good practices. The authors of this paper [9] give suggestions as part of the paper. The authors of the paper [12], discussed in the previous section, offer advice in the papers [10] and [11].

## 4. Code Quality in Notebooks

Coming from a Software Engineering background, we were substantially puzzled by the impact a medium notebook had on our programming style. Long standing rules as published e.g. in [8] or [16] seemed to be challenged. The following two sections are an elaboration of parts of our extended abstract [13].

### *4.1* Programming Style

Notebooks that were obviously developed with great care by other researchers showed similar styles as ours: Global variables are very common. Functionality was rarely encapsulated in functions or objects but provided by top-level statements. There was almost no information hiding. How did the medium shove us into following a programming style that every experienced developer would naturally frown upon?

On a second look it became clear, that the very idea of having a computational narrative presented as a linear sequence of cells, leads to this style. The global variables contain the data that the notebook sets out to analyze and some intermediate result. They are like the protagonists of the narration. The top-level statements describe what is done to these protagonists and ideally the result of the cell tells us the state they have reached afterwards. In contrast, a function definition has no immediate result beyond showing up as a defined function. It does not contribute to the state reached. Since the notebook presents a calculation, we do not want to hide information. As far as the code is only a minor detail, hiding it would still be nice.

By now it should have become clear that we cannot simply reuse established code quality criteria, but we need to take the goal of creating a linear computational narrative into account.

### *4.2 Design Patterns*

While we cannot just reuse the concrete quality criteria we have for software in general, we may reuse how we find and describe good solutions. The notion of a design pattern being a “solution to conflicting forces in a context” (see e.g. Section 1.1. of [1]) is helpful. We will repeat three patterns that were already published in [13]. Our general context is a calculation presented as a linear narrative. We will only describe the most essential aspects of a pattern. The “forces” are all the conditions that are relevant to a certain design decision. The “solution” is a design idea that gives justice to all the forces.

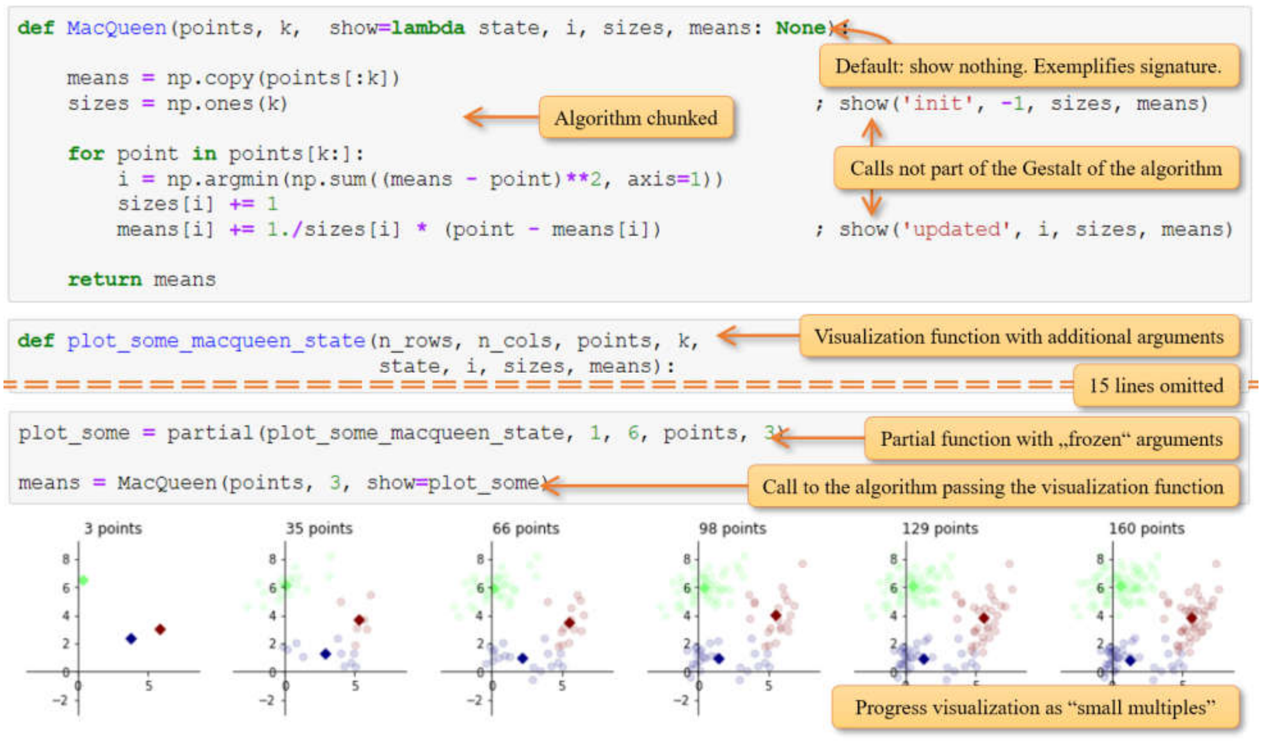
“Function Exemplification” - *Forces:* We mentioned that a function definition itself does not immediately produce output and thus does not immediately contribute to the narrative. Nevertheless, a function is helpful for internal reuse, if for example the same calculation should be executed a few times with different parameters. Additionally, splitting a longer calculation into a few functions with intention of revealing names, may provide structure and help understanding.

*Solution:* Illustrate the use of the function in the next cell. Placing the call to the function in another cell separates the definition from the exemplification. Choosing the next cell keeps them close enough to communicate the connection. Giving such an example call should at least be possible for functions without side effects, short runtime, and easy to provide parameters.

“Updated Progress Line” - *Forces:* For regular software program code, progress information and results are presented in different locations. In the notebook everything is presented next to each other. Therefore, it is essential that progress information does not take up too much space once the calculation is finished. Nevertheless, if the calculation is long running, it is essential to get feedback about the progress. Only with progress information the user knows that the calculation is not stuck and how long she needs to wait for the result.

*Solution:* Let the calculation repeatedly overwrite only temporarily interesting progress information in the same line. `'\r'` positions the cursor at the beginning of the current line so that progress feedback can be given via:

```
print(
  'Progress: {} of {}.'.format(i, n),
  end='\r').
```


```
def MacQueen(points, k,  show=lambda state, i, sizes, means: None):

    means = np.copy(points[:k])
    sizes = np.ones(k)                                      ; show('init', -1, sizes, means)

    for point in points[k:]:
        i = np.argmin(np.sum((means - point)**2, axis=1))
        sizes[i] += 1
        means[i] += 1./sizes[i] * (point - means[i])       ; show('updated', i, sizes, means)

    return means
```

```
def plot_some_macqueen_state(n_rows, n_cols, points, k,
                             state, i, sizes, means):
```

```
plot_some = partial(plot_some_macqueen_state, 1, 6, points, 3)

means = MacQueen(points, 3, show=plot_some)
```
Default: show nothing. Exemplifies signature.
Algorithm chunked
Calls not part of the Gestalt of the algorithm
Visualization function with additional arguments
15 lines omitted
Partial function with „frozen" arguments
Call to the algorithm passing the visualization function
3 points
35 points
66 points
98 points
129 points
160 points
Progress visualization as "small multiples"

Figure 4: Visualization Callback Pattern. The implementation of MacQueen's k-Means algorithm is almost free of visualization concern and can be configured with different callback functions. Here we have chosen a function that creates plots for start and end state and a fixed number of plots in-between. The plots show the clusters with their means. (The notebooks is available online.)

Alternatively, different libraries have functions or objects providing the same effect. The essence of this pattern is a visual presentation of the progress information in limited space.

"Visualization Callback" - *Forces:* To illustrate an algorithm, its implementation should not be influenced by other concerns. In addition, we often want to show intermediate state of the algorithm. The same implementation should be usable with or without visualization. If it is not visualized, it should be fast. It is often interesting to visualize algorithms in varying detail and with respect to different aspects.

*Solution:* We pass a function as a parameter to the function that implements the algorithm. See **Error! Reference source not found.** for an example. As a default value this parameter gets an anonymous function doing nothing. The performance cost of invoking this function is below the cost of three integer additions (as discovered by experiment) and thus acceptable if not used in an innermost loop that contains otherwise extremely cheap calculations. The algorithm function calls the parameter function passing all potentially interesting information in. Visualization functions that do show something may have additional parameters that can be ``frozen'' by creating a partial function. If the functions were objects, we would talk about a "Strategy" Pattern with a "Null Object" as default strategy. If the space in the cell allows, we might position the calls further to the right so that they are pre-attentively perceived as separated from the algorithm.

**Error! Reference source not found.** shows the "Visualization Callback" in one of the notebooks we implement in our recent project [17]. The callback allows to keep the algorithm (here MacQueen's k-means algorithm) unconcerned with visualization. There is only the function parameter and the calls, separated by the layout. The callback is still able to produce a visualization consisting of small multiples (see page 67 in [14]) illustrating that the means move mostly at the start and reach plausible positions.

For long running computations the developer may choose a callback function that just generates an "Updated Progress Line".

A pattern is only a pattern if it is used frequently enough. We convinced ourselves that this is the case for the presented three patterns. "Function exemplification" is not only found in our own notebooks but even more consequently in notebooks of commercial deep learning courses. "Updated Progress Line" is not only a feature within our own notebooks but as well in the deep learning library Keras. "Visualization Callback" is a variant of the pervasive "Strategy" pattern and used at least four times with at least six different visualization functions in our own notebooks.

## 5. Pushing the Boundaries

Based on the previous discussion and our lab experiences, we give an outlook on new developments like collaborative editing, a code gathering tool that extracts consistent code from explorational coding sessions, and some good practices.

### 5.1 The Benefits of Real-Time Collaboration – Observational Study: Remote Pair Work

While a notebook may be shared as easily as any other document, it was not possible to let more than one person edit it at the same time. Wang et al. report in [15] on an observational study of 24 data scientists working in pairs remotely either on a shared notebook ("shared condition") or on separate notebooks with the explicit permission to send code snippets or entire notebooks to each other ("non-shared condition"). The pairs were under both conditions allowed to communicate through further channels of their choice.

The pairs were given a "typical data science predictive modeling problem" to solve. Under the non-shard condition participants worked sequentially through the phases of 1) preparing, 2) cleaning, 3) modeling, 4) feature engineering and 5) submission. Under the shared condition the pairs worked in the same time overall twice through the phases 2) till 4). Working under the shared condition led to better prediction results (average 0.17 compared to 0.27), a higher number of models (6.17 compared to 3.00) and more lines (186.67 compared to 90.33). Only the percentage of annotation cells dropped slightly (19% compared to 20%). The key finding is that "working on synchronized notebooks can improve the collaboration outcomes by reducing communication costs and encouraging more exploration in a shared context."

### 5.2 Transition from Exploration to Explanation – In-Lab Usability Study: Code Gathering Tool

Head, Hohman, Barik, Drucker, and DeLine presented in [4] tool support for the transition from a notebook created during exploration to a notebook presenting an explanation. Speaking in reengineering terms their code gathering tool calculates the backward slice within the whole notebook history starting from interesting results as slicing criterion.

In more detail: While the data analyst interacts with the notebook, the tool records a history of the executed cells. When the user selects a result as a starting point for the gathering, the tool analyses the history backwards and keeps only those parts of the code that contributed to the result. This excerpt can then be exported into a new notebook or pasted into the notebook again. Note that this process reproduces the cells in execution order, even if they are in a different physical order in the notebook.

The authors conducted an in-lab usability study with 12 participants cleaning two given notebooks and executing an exploratory data analysis on another given dataset. The participants considered the possibility to gather code into a new notebook to be very useful (12/12 for the cleaning, 10/12 for the exploratory data analysis). The possibility to highlight lines the code depends on, was considered helpful as well (7/11 very useful, 10/12 at least somewhat useful). Reordering gathered code as cells was very useful to those, who used the feature during exploratory data analysis (6/7). We are convinced that the code gathering tool could be of high value to make the transition from exploration to explanation more seamless.

### 5.3 Practices

Not all steps to better notebooks need better tools. Some guidance can make a huge difference. Let us list Rule et al.'s ten simple rules published in [10] and [11]:

1) Tell a story for an audience
2) Document the process, not just the results
3) Use cell divisions to make steps clear
4) Modularize code
5) Record dependencies
6) Use version control
7) Build a pipeline
8) Share and explain your data
9) Design your notebooks to be read, run, and explored
10) Advocate for open research

Rule 5) helps to resolve the problem of missing dependencies prevalent in [9]. Rules 4) and 6) follow the demand for more software engineering in notebooks requested in [3]. To avoid the problem of out-of-order executions and hidden state it is furthermore recommendable to rerun the notebook and to even restart the kernel regularly.

## 6. Conclusion

Computational notebooks were created as a medium to present data science results as a computational narrative. Users found further use cases and stored their notebooks in public repositories. We shared our own insights into the specific quality criteria that result from the fact that the code does not only provide functionality but is itself already a message to the user. The public availability of the tooling as well as the notebooks open up to public both review for criticism and inspiration. Two empirical studies were able to identify shortcomings in published notebooks and suggest improvements. Research groups took up the challenge to improve the tooling and provided advice on how to produce notebooks of good quality.

## Acknowledgment

We thank Tiansi Dong, Jörg Zimmermann, Alexandra Reitelmann, and Michael Mahlberg, who provided feedback on an earlier version of this paper. We thank Olaf Cremers for his contributions to the notebooks and for sharing his insights into Jupyter notebooks. We thank Christian Bauckhage for sharing his source code on which we built the Jupyter notebooks published at [17].

## References

[1] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Perkel, J. M. (2018). By Jupyter, It All Makes Sense. *Springer Nature*, 145–146. https://doi.org/10.1038/d41586-018-07196-1

[3] Grus, J: I Don't Like Notebooks, JupyterCon 2018, online, accessed 11.11.19, https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI/, https://youtu.be/7jiPeIFXb6U

[4] Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019, April). Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (p. 270). ACM. https://doi.org/10.1145/3290605.3300500

[5] Kazantsev, V., Nerush, K.: Clean Code in Jupyter Notebooks, PyData 2017 Berlin. https://de.slideshare.net/katenerush/clean-code-in-jupyter-notebooks, https://www.youtu.be/2QLgf2YLlus

[6] Kery, M. B., & Myers, B. A. (2018, October). Interactions for Untangling Messy History in a Computational Notebook. In 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (pp. 147-155). IEEE. https://doi.org/10.1109/VLHCC.2018.8506576

[7] Kery, M. B., Radensky, M., Arya, M., John, B. E., & Myers, B. A. (2018, April). The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (p. 174). ACM. https://doi.org/10.1145/3173574.3173748

[8] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053-1058. https://doi.org/10.1145/361598.361623

[9] Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019, May). A large-scale study about quality and reproducibility of jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories* (pp. 507-517). IEEE Press. https://doi.org/10.1109/MSR.2019.00077

[10] Rule, A., Birmingham, A., Zuniga, C., Altintas, I., Huang, S. C., Knigh, R., Moshiri, N., Nguyen, M. H., Rosenthal, S. B., Pérez, F., & Rose, P. W. (2018). Ten simple rules for reproducible research in Jupyter notebooks. *arXiv preprint https://arxiv.org/abs/1810.08055*.

[11] Rule, A., Birmingham, A., Zuniga, C., Altintas, I., Huang, S. C., Knight, R., Moshiri, N., Nguyen, M. H., Rosenthal, S. B., Pérez, F., & Rose, P. W. (2019). Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. *PLoS computational biology*, *15*(7). https://doi.org/10.1371/journal.pcbi.1007007

[12] Rule, A., Tabard, A., & Hollan, J. D. (2018, April). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (p. 32). ACM. https://doi.org/10.1145/3173574.3173606

[13] Speicher, D., Dong, T., Cremers, O., Bauckhage, C., & Cremers, A. B. (2019) Notes on the Code Quality Culture on Jupyter (Notebooks). 21. Workshop Software-Reengineering & Evolution (WSRE) 2019, Bad Honnef, Germany.

[14] Tufte, E. *Envisioning Information*. Graphics Press, Cheshire, CT, USA, 1990.

[15] Wang, A. Y., Mittal, A., Brooks, C., & Oney, S. (2019). How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW), 39. https://doi.org/10.1145/3359141

[16] Wulf, W. A., & Shaw, M. (1973). Global variable considered harmful. https://doi.org/10.1145/953353.953355

[17] p3ml.github.io | *P3ML Material on GitHub*, https://p3ml.github.io/, accessed 11.11.2019

**Daniel Speicher** received his diploma in mathematics from the University of Bonn in 2003. He taught lectures in Object-Oriented Software Construction, Advanced Topics in Software Construction, and Aspect-Oriented Software Development at the Bonn-Aachen International Center for Information Technology (b-it). He contributed to a successful series of Agile Software Development Labs as part of the International Program of Excellence (IPEC) at the b-it. His main research interest lies in automatic code quality evaluation with reason, i.e. code quality evaluation that takes context into account, can be adapted to incorporate expert knowledge and provides explanations. One of his contributions to the P3ML project was establishing the JupyterHub installation.

**Armin B. Cremers** received his doctoral degree in mathematics and his lectureship qualification in computer science from the University of Karlsruhe (now KIT). From 1973 he has served on the Computer Science Faculties of the University of Southern California, the University of Dortmund, and, since 1990, the University of Bonn as Head of the Artificial Intelligence / Robotics / Intelligent Vision Systems Research Groups. In 2002 he became Founding Director of the Bonn-Aachen International Center for Information Technology (b-it), in 2004 Dean of the School of Mathematics and Science and in 2009 University Vice President. Emeritus since 2015. Works on AI Foundations and AI Systems Engineering.