Opinionated Analysis Development

Hilary Parker
Stitch Fix
hparker@stitchfix.com

August 30, 2017

Motivation

Background

Statistical software is, at it's core, a language one uses to create a convincing analysis. The final creative product is a narrative that will address a scientific or business question in a way deemed satisfactory. Within a scientific setting, this generally will be developing a narrative that allows a scientific finding to be published in a peer-reviewed setting. In a business setting, there are myriad endpoints for an analysis, but most serve the purpose of helping business partners make a decision. Folded into this narrative is the choice of experimental approach and statistical methods with known properties that convincingly model or approximate the data.

As with any mode of expression, a practitioner must first learn the technical skill of the trade before they can use it to create. A photographer must learn how to manage the aperture, shutter speed, and numerous other features of a camera that control light exposure before she can use the tool to create unique and affecting photographs. The statistical analog for this – and the focus of this paper – is the process of creating the technical artifact using the statistical language and other tools, that delivers the analysis narrative to interested parties. In a scientific setting, the technical artifact is often a journal article. Within a business setting, the technical artifact may be a quick email, a slide deck, a white paper or a long-lived dashboard. In all these creative fields, increased fluency and mastery of the tooling means that the practitioner can create uninhibited.

Statistical training often focuses on the narrative aspects of this process: mathematical derivations and proofs of statistical tests, methods and models. This foundational training is crucial to understanding the strengths and limitations of conclusions that can be drawn from a particular approach to analyzing data. However, the process of developing the technical artifact is less frequently taught, or even acknowledged as a set of necessary skills. Given that this process is complex and prone to error, this hamstrings practitioners, keeping them from establishing fluency in the tools and allowing them to make common, avoidable and time-consuming mistakes. The purpose of this paper is to present clear opinions on how, technically, an analysis should be developed, drawing from recent developments in related fields, available tooling and common best practices.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3210v1 | CC BY 4.0 Open Access | rec: 31 Aug 2017, publ: 31 Aug 2017

Process and Human Error

In order to discuss nuances of a process, it's important and useful to define it first. I define the process of technically creating an analysis as "analysis development", directly borrow from software development. "Analysis engineering" is an acceptable alternative, and in both fields these terms are used exchangeably.

By defining the process, we can begin to borrow from the rich field of operations, which focuses primarily on process. One paradigm that proves especially useful is the concept of human error. The seminal book *The Field Guide to Understanding Human Error* argues for a paradigm shift from the "Old World View" (that when an error occurs it is an individual actor's fault) to the "New World View" (that when an error occurs, it is a symptom of a flawed system that failed that individual actor) (Dekker 2014). When an error in an analysis occurs, it is safe to assume (aside from nefarious actors) that the analyst did not want that error to occur. Given that she thought she was producing an analysis free from errors, you must look at the way she developed the analysis to understand where the error occurred, and create safeguards so that the error does not occur again.

Complex processes such as flying an airplane with autopilot are filled with safeguards against error. Anyone who has sufficient experience performing analysis instinctively understands many of these common pitfalls. Furthermore, we don't need to look far to see examples of them in practice (including the rare nefarious actor) (Baggerly and Coombes 2009; Hermans and Murphy-Hill 2015). Nicholas Radcliffe discusses many of these errors, and breaks them into different classes, in his Test-Driven Data Analysis work (Radcliffe 2016). Our hesitation as a field to codify safeguards against these common errors results in the continual repetition of these frustrations. And given that people (not even statisticians) have the "Old World View", it means that many analysts feel or are attributed with personal responsibility when these errors occur, despite not being taught processes that protect against them.

This brings us to the term "opinionated," which is also borrowed directly from software development. Opinionated software is software that pushes the user to follow certain practices (Eccles 2015), from file structure to design features. It is alternatively described as following a "convention over configuration" principle. A software language itself can be considered opinionated if it encourages the programmer to follow certain principles. The creator of Ruby on Rails, an opinionated framework for web development, describes this trade-off:

> It's a strong disagreement with the conventional wisdom that everything should be configurable, that the framework should be impartial and objective. In my mind, that's the same as saying that everything should be equally hard.

(Bedell 2006).

The motivation for defining "opinionated analysis development" is put the "new world view" of human error into action, and codify certain best practices that guard against error during the process of technically creating an analysis. Equally important is to have this abstracted away from specific software choices, where this conversation so often takes place.

If we accept that some practices are better than others at avoiding errors, then a logical conclusion from that is that analyses developed with these practices are inherently better than equivalent analyses developed without these practices. Many statisticians balk at the thought of "cookbookery," and for good reason – developing the narrative of an analysis is a deeply creative process, moreso than we usually acknowledge, and statisticians might be worried that this will be inhibited. By defining and teasing apart the process of creating the technical artifact in a way that minimizes error, my hope is that we as a field can begin to teach analysts skills and process necessary to free up their cognitive energy and creativity for the narrative aspect of the analysis that is more subjective and dependent on the audience and scientific question at hand. Put another way–and borrowing phrasing from Hadley Wickham–the goal of establishing these opinions is to shift the bottleneck of analysis from the common pitfalls to the creative and context-specific choices. In addition to making analysis fundamentally more creative and satisfying, it will make analysts more productive as well.

One final aside is that this paper focuses specifically on the process of developing an analysis, rather than a productionized technical statistical product such as recommendations served on a website. While many of the principles overlap, more energy (within an industry setting) has been focused on this production problem at the expense of careful thought of analysis deliverables themselves.

Opinionated Analysis Development

I propose here that an analysis should have these key features: reproducible and auditable, accurate and collaborative. Below I present opinions for how to achieve each of these features. Many software solutions might exist that allow for the easy implementation of the opinions below. In fact, given the diversity of deliverables needed in the business setting, having multiple tools is necessary. Throughout the remainder of this paper, I provide some examples of software (primarily in R) that provide elegant implementations of the opinions. These examples are not meant to be exhaustive, but rather illustrative.

While these opinions are listed in an order that reflects my perceived hierarchy of needs, not every analysis needs to implement every opinion. A short ad-hoc analysis, for example, might be easily accomplished while implementing only a couple of these opinions, just as a quick software project won't necessarily embody all the principles of a large enterprise software product. However, by establishing the principles, an analyst has the ability to subjectively choose when an analysis becomes sufficiently large or complex to require implementing the opinions.

The key features and associated opinions for developing the technical artifact for an analysis are as follows:

- Reproducible and Auditable
  - Executable analysis scripts
  - Defined dependencies
  - Watchers for changed code and data
  - Version control (individual)
  - Code review
- accurate
  - Modular, tested code
  - Assertive testing of data, assumptions and results
  - Code review
- Collaborative
  - Version control (collaborative)
  - Issue tracking

Table 1: Common questions solved by opinionated analysis development, listed by opinion.

Analysis Feature	Opinionated Approach	Question Addressed
Reproducible and Auditable	Executable analysis scripts	Can you re-run the analysis and get the same results?
		Can someone else repeat your analysis?
		Can you re-run the analysis on different data?
	Defined Dependencies	If an external library you're using is updated, can you still reproduce your original results?
		If you change code, do you know which downstream code need to be re-executed?
	Watchers for changed code and data	If the data or code change but the analysis is not re-executed, will your analysis reflect that it is out-of-date?
	Version Control (individual)	Can you re-run your analysis with new data and compare it to previous results?
		Can you surface the code changes that resulted in a different analysis results?
	Code Review	Can a second analyst easily understand your code?
accurate	Modular, tested code	Can you re-use logic in different parts of the analysis?
		If you decide to change logic, can you change it in just one place?

Analysis Feature	Opinionated Approach	Question Addressed
		If your code is not performing as expected, will you know?
	Assertive testing of data, assumptions and results	If your data are corrupted, do you notice? If you re-run your analysis on different data, are the methods you used still valid?
	Code Review	If you make a mistake in your code, will someone notice it? If you are not using efficient code, will you be able to identify it?
Collaborative	Version Control (collaborative)	Can a second analyst easily contribute code to the analysis? If two analysts are developing code simultaneously, can they easily combine them?
	Issue tracking	Can you easily track next steps in your analysis? Can your collaborators make requests outside of meetings or email?

The remainder of this paper describes each feature and opinion, with further context and examples.

Reproducible and auditable

Reproducibility is a feature of analysis that has been widely discussed. Much has already been written on the subject, and in fact many definitions of reproducibility may contain some of other opinions I have presented here (Stodden, Leisch, and Peng 2014; Ram 2013).

Executable analysis scripts

The first, most important and most conventional opinion presented here is that every analysis should ultimately become a reproducible script or set of scripts, from ingesting the raw data to compiling the final deliverable. The purpose of this opinion is simple: it ensures that every step of analysis has been recorded and communicated so that it can be re-run by either the original analyst or a second one, on either the same set of data or an updated one.

This approach to statistical analysis comes at odds with the many programs and programming languages (such as R) that encourage interactive data exploration.

Such interactivity can be key for the creative exploration of a data set or statistical method. However, as the analysis approaches the form of a final deliverable, the analyst should converge to a scripted approach rather than an interactive one.

Defined dependencies

The second opinion is that the analysis should have a defined dependency tree or directed acyclic graph (DAG). This concept is critical for the actual process of re-running an analysis; it ensures that the components of an analysis can be pieced together and re-run. One can, for example, easily imagine a complex analysis with many separate scripts, where it is relatively impossible for fresh eyes to piece together what order they should be run in to reproduce the results.

A secondary benefit to creating a dependency tree is that it reduces the amount of computation time needed to re-run analysis. For example, consider an analysis that compares the results of a method on several different simulated data sets. If the analyst re-simulates one of these data sets, she will know exactly which code needs to be re-executed in order to update the analysis. Saving computational time isn't in itself a necessary component of rigorous analysis development; one can imagine a completely correct analysis that is slow to run. However, reducing computational time can increase the probability that an analyst re-runs analysis every time the code is updated, which in turn increases the probability that the scripts themselves are reproducible.

Included in the dependency tree is software dependency, such as the package versions used in the analysis. The moment a package is updated by an outside developer, it might jeopardize the reproducibility of an analysis. Ostensibly updated software should always be "better" – that is, one can presume that software converges to more correct implementations of theory. However, there are myriad reasons why an analyst might not update her analysis the moment a new software version comes out. For one, it will take time to assess if the analysis has changed, whether it is more correct with the newer software version, and whether it makes practical sense to rerun the analysis. In software engineering this might be thought of as always ensuring that you have a running version of the code.

There are several examples of tools that encourage explicitly defining dependency trees or DAGs for the analysis scripts. `knitr` – an R package for creating dynamic documents in R – has the option of defining code "chunk" dependencies throughout the document (Xie 2016). `ProjectTemplate` – an R package for defining project architecture – encourages users to adopt a specific analysis DAG (White 2014). It accomplishes this through the use of specific code sub-directories as well as functionality to re-execute code from different points within the DAG (such as before or after preprocessing or munging data). GNU Make is a UNIX tool that defines the dependency tree ("GNU Make" 2016), and facilitates executing code according to this dependency tree.

For package versioning, there also exist several solutions that run the spectrum of "ease of reproduction". Docker, a system for creating environments, is an elegant solution for creating, archiving and sharing a disc image with the exact run environment used for an analysis ("Docker" 2016). `packrat` is an R package that creates a local, private package library of every R package used in an analysis (Ushey et al. 2016). This is snap-shotted so that it can be re-built by another user or on a different system. Microsoft archives daily snapshots of CRAN ("The CRAN Time Machine"), a tool that an analysis developer can leverage with the run-date of an analysis ("The Cran Time Machine" 2016). At the very least, using the `sessionInfo` function in R will define exactly what version of packages were used in a specific session – a worthy appendix item for any analysis deliverable.

Watchers for changed data and code

A final opinion for reproducibility is watching – that is, knowing when data or analytical code changes and whether it is synced with the current version of the analysis.

GNU Make, discussed above, includes this watching functionality for free. By mapping an analysis into a makefile, the analyst can re-execute code and GNU Make will only execute downstream dependencies from changed code or files.

Code review

Code review is the process of soliciting feedback on analytical code from another experienced analyst, and appears under two features presented here. Unstructured code review can expand far beyond the scope of establishing whether or not the development of the technical artifact was correct and efficient.

Code review is a critical component of developing a reproducible and auditable analysis because it is the only way to test whether or not it actually accomplishes this goal. Performing analysis – even in isolation – is fundamentally a collaborative endeavor because the fundamental purpose of an analysis is to communicate a result with the supporting code and data. The analytical code itself can be thought of as the language which communicates the thought process and methods used, a concept first developed by Donald Knuth as part of literate programming (Knuth 1992). A primary goal of code review, then, is to test whether or not this goal is accomplished.

The process of code review provides critical feedback on how the technical artifact was developed. It also will serve a psychological purpose when an analyst is in the process of developing the artifact. If a statistician is developing her analysis with code review in mind, it will influence how she writes the code, from adding additional, thorough comments to choosing specific functions that are most readable.

Accurate

The idea that an analysis should be accurate is probably such a strong assumption within a scientific or business setting that it is rarely discussed.

Modular, tested code

Creating modular and tested code (most frequently, creating functions with unit tests) is the process of creating a unit-tested function to perform repeated aspects of an analysis. There is no specific moment when a function should be created – it is a balance of the time needed to develop the function versus the time saved from having the function. However, creating functions is key for analysis because it means that changes only need to be made in one place. Not creating a function leaves the possibility of updating the methods for one implementation of the code but not another.

A key benefit of creating modular, tested code is that an analyst can test the validity of the functions using unit tests. Unit testing within software engineering is an established practice. A software engineer will write a function, and then create an assertive test that replies “true” or “false” to certain implementations of the function, in order to check whether the function is performing as expected. Downstream functionality that depends on the function might be triggered to fail if the unit test returns a false.

The process of creating functions for code is one that is usually emphasized as a part of reproducibility. However, I tease it apart from the reproducibility opinion because it is possible to have a reproducible process that contains no functions. While tool-makers often focus on cutting edge statistical methodology, a large portion of statistical analysis (especially in a business setting) involves no special methodology aside from aggregations and sub-setting. In those cases, reproducibility is still critical but an emphasis on writing modular code would not reap extensive benefits.

Assertive testing of data and assumptions

Statisticians are well-trained on the assumptions of specific statistical methodologies. This may draw from the fact that the asymptotic proofs for statistical methods rely on certain assumptions, and are therefore emphasized in both theoretical and applied training. As with many of the other issues discussed in this paper, statisticians are often left with the advice to check assumptions and quality-control data, but without the practical tools necessary to do so conveniently within the technical artifact. This is made even more of a dangerous territory as reproducibility is adopted. While reproducibility drastically reduces the number of errors and opacity of analysis, without assertive testing it runs the risk of applying an analysis to corrupted data, or applying an analysis to data that have drifted too far from assumptions.

The act of assertive testing in data analysis will, in practice, look very similar to unit testing. The difficulty therein is making sensible decision rules based on the outcome of the tests. Taken one way, this is the very challenge of inferential statistics generally as it applies to making decisions based on conventional rules (such a the widely-covered p-value debate). The key principle, therefore, isn't the establishment of specific rules. Rather, it is the establishment that any defined rule should be explicitly tested, with possible consequences within the analysis document based on those test results.

The same can be said for data testing. Almost always, by the time a data analyst has gotten to the point of performing an analysis, she has made the assumption that the data are not corrupted, usually by inspecting the data or a subset of the data manually. A negative value for a weight measurement, for example, would immediately signal to the analyst that she should figure out what "went wrong". Assertive testing means establishing these quality-control checks – usually based on past knowledge of possible corruptions of the data – and halting an analysis if the quality-control checks are not passed, so the analyst can investigate and hopefully fix (or at least account for) the problem. This could take the form of throwing an error in the analysis script, or displaying an error within the analysis document.

Test-driven data analysis is a comprehensive example of defining different checkpoints within an analysis workflow that would benefit from testing (Radcliffe 2016). The authors define errors of implementation, error of interpretation, errors of process and errors of applicability as key errors to test for within the data analysis process (Surry and Radcliffe, n.d.).

Creating more flexible testing frameworks is an area of active development within the R community. The `validate` R package, for example, provides infrastructure for creating rules, testing data against those rules and concisely displaying the results (van der Loo and de Jonge 2016). The `assertr` package similarly provides functions for creating tests, with the intent of them breaking the analysis pipeline when they fail (Fischetti 2015).

Version control – individual

Version control refers broadly to the process of saving intermediate versions of a project – rather than the current final deliverable – into a centralized or distributed repository. Version control has become the status quo within software development; it would be extremely rare to find a large software project that did not employ some type of version control. It is especially critical in this context because if new code within a project breaks the software, the developers want the ability to "roll back" to the last working version of the software as they debug.

Most statisticians have probably implemented some type of version control, even if it is as simple as saving updated versions of a paper as a new file with a slightly different file name. Additionally, version control is implemented into Google Docs. However, using a formal version control system, such as git or svn, will greatly increase the probability that the process of saving versions will be functionally helpful. Saving intermediate results, for example, will not be helpful if the statisticians cannot go back and find the version she needs. Peter Ellis provides a thorough discussion of the multiple benefits of using a robust version control system in the context of data analysis (Ellis 2016).

Learning a formal version control system might present a larger technical barrier than other points raised in this article. Fortunately, there are some training resources available from the more popular version control resources. GitHub, for example, provides free introductory online classes for version control ("GitHub Services: Training" 2016). Additionally, some professors have integrated version control into their courses with great success. Jenny Bryan, from University of British Columbia, incorporates GitHub into her introductory statistics course (Bryan 2016). The course materials itself are a Github repository, and students submit their homework via pull request (the process for formally incorporating code into the central repository).

Collabortive

Version control – collaborative

Version control is a valuable method for a statistician who is developing an analysis on her own. It also provides critical infrastructure when the process of developing an analysis becomes collaborative. The number of moving parts, and the complexity of potential problems, for collaborative developing a code base increases substantially with the number of collaborators. Any analyst who has attempted to create a collaborative project has, no doubt, run into the pitfalls of making sure that everyone has the updated code and data, and understands exactly what aspects of the analysis they should be tackling.

The benefits of a robust version-control system, such as git or svn, are numerous within a collaborative context. They are fundamentally designed to allow for changes of code to be merged. A version control system will provide a centralized repository of code that all the collaborators can access, and a defined method for merging different pieces of code. Many also include communication tooling, discussed next.

One advantage of establishing version control best-practices (such as using GitHub) in an individual context is that it allows for a relatively seamless transition to a collaborative environment.

Issue tracking and other communication tooling

Issue tracking is the process of creating and exposing a list of problems that need to be solved within a project. A robust issue-tracking system can solve three main problems: it can communicate known problems that an analyst wishes to tackle, it can also communicate problems that she already solved or chose to not solve, and it can allow for outside parties to communicate feedback or new issues. Issue tracking and communication is as much about establishing effective collaborative process as it is about anything inherent to the process of analysis itself.

Establishing a robust issue-tracking system can reduce cognitive burden of tracking to-do items. Given the collaborative nature of many analyses, it also provides the infrastructure necessary to have efficient conversations about how specific items will be tackled. For example, the best issue-tracking systems allow for discussion and visible archival of issues, which reduces the need to have redundant conversations around the approach taken in an analysis.

GitHub provides an effective model for issue tracking. It provides the ability to create issues, have archived and public or private conversations around issues, and finally tag issues as "complete" or "did not finish" along with custom labels. Additionally, GitHub provides the ability to connect an issue to a specific pull request or chunk of code. By doing this, it is very easy to link the additional code written to solve a certain problem within an analysis.

Code review

Code review – discussed once before for developing a reproducible and auditable analysis – is also a key feature for developing an accurate analysis. Children are taught from an early age to have someone read over their writing to check for spelling and grammatical errors. Coding should be no different. Code review will not guarantee an accurate analysis, but it's one of the most reliable ways of establishing one that is more accurate than before.

As discussed in the introduction, the purpose of this paper is to established the opinions for developing the technical artifact, rather than developing the narrative of an analysis. This distinction proves important for the code review process, as soliciting review on an analysis will likely spark feedback on both the narrative as well as the code itself. An analyst can be structured in the way she solicits code review feedback, by directly stating that she is soliciting feedback on whether or not her code correctly and efficiently implements a set statistical method, rather than whether or not that statistical method is inherently correct. Mike Birbilgia, for example, in discussing Ron Howard's approach to feedback, states: "He doesn't do it to be told what the movie's vision should be, but to understand whether his vision is coming across" (Birbiglia 2016).

Finally, a tangential benefit to code review by more experienced developers is that it allows for feedback on code efficiency and practicality. As is common in other aspects of the field, the idea that as long as the code accomplishes the goal then it is valid is not helpful for the fledgling analyst. Every analytical language has optimized for certain approaches (such as vectorized calculations over loops), and providing feedback on implementation is often the only way someone will learn about it.

Conclusion

The state of the art for analysis development has come to a point where opinionated methods and tooling need to become the norm. Statisticians have long shied away from teaching process, with the complaint that it might limit the creativity necessary to tackle different analytical problems. However, by not teaching opinionated analysis development, we subject fledgling data to each individually spin their wheels in coming up with process for avoiding common and generalized problems.

The opinions in this paper address specific, known, and common pitfalls in the analysis development process. By encouraging the use of and fluency in tooling that implements these opinions, we can foster the growth of processes that fail the practitioners as infrequently as possible.

Acknowledgements

Thank you to Jenny Bryan and Skipper Seabold for their contributions in shaping the ideas in this paper. Many thanks also to Karl Broman, Roger Peng, John McDonnell and Ben Marwick for their feedback.

Bibliography

Baggerly, Keith A, and Kevin R Coombes. 2009. "Deriving Chemosensitivity from Cell Lines: Forensic Bioinformatics and Reproducible Research in High-Throughput Biology." *The Annals of Applied Statistics*. JSTOR, 1309–34.

Bedell, Kevin. 2006. "Opinions on Opinionated Software." *Linux Journal* 2006 (147). Belltown Media: 1.

Birbiglia, Mike. 2016. "Mike Birbiglia's 6 Tips for Making It Small in Hollywood. or Anywhere." Edited by New York Times. \url{http://www.nytimes.com/2016/09/04/movies/mike-birbiglias-6-tips-for-making-it-small-in-hollywood-or-anywhere.html}.

Bryan, Jenny. 2016. "Stat 545." http://stat545.com/.

Dekker, Sidney. 2014. *The Field Guide to Understanding'human Error'*. Ashgate Publishing, Ltd.

"Docker." 2016. https://www.docker.com/.

Eccles, Stuart. 2015. "The Rise of Opinionated Software." https://medium.com/

@stueccles/the-rise-of-opinionated-software-ca1ba0140d5b#.etoe6fbd2.

Ellis, Peter. 2016. "Why You Need Version Control." http://ellisp.github.io/blog/2016/09/16/version-control.

Fischetti, Tony. 2015. *Assertr: Assertive Programming for R Analysis Pipelines*. https://CRAN.R-project.org/package=assertr.

"GitHub Services: Training." 2016. https://services.github.com/training/.

"GNU Make." 2016. https://www.gnu.org/software/make/.

Hermans, Felienne, and Emerson Murphy-Hill. 2015. "Enron's Spreadsheets and Related Emails: A Dataset and Analysis." In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, 7–16. IEEE Press.

Knuth, Donald E. 1992. "Literate Programming." *CSLI Lecture Notes, Stanford, CA: Center for the Study of Language and Information (CSLI), 1992* 1.

Radcliffe, Nicholas. 2016. "Why Test-Driven Data Analysis?" http://www.tdda.info/why-test-driven-data-analysis.

Ram, Karthik. 2013. "Git Can Facilitate Greater Reproducibility and Increased Transparency in Science." *Source Code for Biology and Medicine* 8 (1): 7.

Stodden, Victoria, Friedrich Leisch, and Roger D Peng. 2014. *Implementing Reproducible Research*. CRC Press.

Surry, Patrick, and Nicholas Radcliffe. n.d. "Four Ways Data Science Goes Wrong and How Test-Driven Data Analysis Can Help." *Predictive Analytics Times*.

"The Cran Time Machine." 2016. https://mran.microsoft.com/timemachine/.

Ushey, Kevin, Jonathan McPherson, Joe Cheng, Aron Atkins, and JJ Allaire. 2016. *Packrat: A Dependency Management System for Projects and Their R Package Dependencies*. https://CRAN.R-project.org/package=packrat.

van der Loo, Mark, and Edwin de Jonge. 2016. *Validate: Data Validation Infrastructure*. https://CRAN.R-project.org/package=validate.

White, John Myles. 2014. *ProjectTemplate: Automates the Creation of New Statistical Analysis Projects*. http://projecttemplate.net/.

Xie, Yihui. 2016. *Knitr: A General-Purpose Package for Dynamic Report Generation in R*. http://yihui.name/knitr/.