Coding to `code::proof`: Practice fundamentals of automated tests for collaborative data analysis development

How to go from no tests, to *whoa* tests!

sketch-draft by lead author, Charles T. Gray

0.1 Version of this manuscript

This manuscript (or manuscripts) will go through three phases:

1. A lead author sketch draft.
2. Implementation with the reproducibiliity team for The repliCATS Project & version listing any team members who wish to co-author.
3. Implementation of repliCATS procedures, explicitly, after we need not blind the algorithms for experimental ingegrity. Invitation for external co-authors to contribute and extend the content, review, and final submission. Possibly for useR2020.

0.1.1 Acknowledgments

People who have contributed feedback or assisted, some of whom might be co-authors on later drafts: Emily Riederer, Elise Gould, Hannah Fraser, Daniel Fryer, Aaron Willcox, David Wilkinson. Also, appreciation for Danielle Navarro, Thomas Pedersen and others on twitter who helped with the visualisation.

1 Coding to `code::proof`

J. S. Bach's Contrapunctus I is not learnt in one sitting [todo: citation]. Indeed, for most pianists, to play the three-minute piece is the result of many months' diligent work, and still it will never feel *done*. To play well, a pianist must employ technqiue, but *how* a pianist is to achieve the desired technique is not clear (Chang 2009).

For a researcher performing an analysis using a computational tool such as R, it can be unclear when an analysis is *done*. And, while there are technical guides on *good enough* practice, it can feel overwhelming as to what to adopt and where. Particularly for an in-development algorithm, which is to say, an algorithm with scripts already started, and some results explored.

This manuscript picks up from where Riederer left off with *RMarkdown-driven development* and suggests a `code::registered` *test-driven workflow* for coding to doneness. This workflow provides a roadmap to completion for a packaged analysis with `code::proof`ed the manuscript (Gray 2019), provided measures of confidence in the implementaiton of the algorithm.

Musicians and, similarly, athletes, do not see themselves as having mastered a skill, but active practitioners of a craft (Galway 1990). To be a flautist is to practice, and to be an athlete is to train. For researcher developers, it can be hard to assess when an analysis is completed. In this manuscript, we consider the *practice* of test-driven analysis development as a means of *coding to doneness* for an algorithm, a workflow that faciliates the analyst achieving sufficient `code::proof` in their algorithm, confidence in the implementation of the aglorithm(Gray 2019).

2 Questionable research practices in scientific computing

Algorithms are coded by people who practice code, and significant problems emerge when algorithms are treated as fixed artifacts, rather than one of the tools utilised by those who *practice* code. In Australia, a heartbreakingly-ongoing example of this kind of problem with the income-reporting data analyis algorithm that assesses if income welfare recipients have been overpaid entitlements. Crude averaging calculations have lead to ongoing incorrect debt notices issued, such as 20,000 people receiving "robodebt" notices for debts they did not owe (McIlroy 2017). Problems in data analysis have real impacts on real people's lives. Perhaps, if the algorithm were considered a workflow practiced and monitored by a team of data scientists, rather than a static object, problems would not be persisting to this day (Karp 2019). Indeed, as noted in Wilson's testing primers (in development) for RStudio ("RStudio Cloud," n.d.),

> Almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software. In 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code (Yuan et al. 2014).

We might view this as a computational instantiation of what Fraser *et al.* denote *questionable research practices* (QRPs). The QRPs Fraser *et al.* provide a taxonomy for refer to various practices in scientific methodology found in ecology and evolution, such as p-hacking, adjusting a model's specification to achieve a desired p-value, or *cherry picking*, failing to report variables or results that do not fit a desired narrative. QRPs are, importantly, often not a result of scientific malpractice, but a question of discipline-specific conventions established in bygone eras not translating well to data-driven research (Fraser et al. 2018). In scientific computing, as the robodebt example illustrates, similar and overlapping errors may occur.

A consideration when providing recommendations of best practice is what me might reasonably expect of a researcher. Indeed, it is likely unrealistic to expect *best practices* in scientific computing (Wilson et al. 2014), perhaps we would be better off asking for *good enough* practices (Wilson et al. 2017) in scientific computing. For while researchers use computational algorithms in science every day, most of them are not trained in computational science. Even mathematicians and statisticians do a great deal of training at the blackboard, rather than the computer.

Riederer identifies several qrps in scientific computing, for example, hardcoded values that interfere with another's ability to reproduce the computational results (Riederer 2019). As researcher software engineers, it behoves us to consider what are questionable research practices in software produced for data analyses. Version control and open code sharing via a platform such as GitHub, is one way to mitigate questionable research practices in scientific computing (Bryan 2017). There is also a growing literature on reproducible research compendia via packaged analyses (Marwick, Boettiger, and Mullen 2018; Wilson et al. 2017).

This manuscript contributes to this literature by focussing on workflows for using automated tests to move analysis a fully reproducible packaged research compendium. Furthermore, these workflosw assist the developer communicate what they have coded to others and their future self, as a means of mitigating questionable research practices in scientific computing. Above all, by providing a workflow for the *practice* of code, the developer anxiety is reduced by having the parameters as clearly defined as they can be from the start.

3 Practice fundamentals

Chuang C. Chang's *Fundamentals of Piano Practice* sets out to address a gap in piano pedagogy (Chang 2009). A similar gap exists in the implementation of *good enough* practices (Wilson et al. 2017), which is to say, what we might reasonably expect of a analyst, in reproducible computing for research data analyses. The objective is different from advanced pianism, but we characterise testing *practice*, as opposed to *techique*, analogous to how Change delineates between *piano* practice and technique. Through attempting to identify the fundamentals of automated testing for collaborative data analysis development, this manuscript aims to articulate the gap in understanding automated testing implementation for analysts.

As Chang notes, whilst there a rich history of technical pedagogy, there is a dearth of guidance for pianists on *learning* the technique (Chang 2009). There are many canonical texts of pianistic technique pedagogy. Bach provides a pathway from small canons [todo: cite], to two-part inventions[todo: cite], three-part sinfonia, and the challenge of *The Well-Tempered Clavier* and *The Art of Fugue*. Bartok provides the *Mikrokosmos* [todo:], and Czerny's *School of Velocity* [to do]. In each case, technical exercises of increasing difficulty are provided. In piano pedagogy, a *technical* exercise isolates a particular aspect of pianistic technique [todo examples]. For example, [todo Czerny staccato]. Or, [todo voicing technique].

The dearth that Chang attempts to address is in pianistic *practice* habits that will lead to succesful adoption of these techniques. In science, we might call this *work flow* [todo cite].

> .. practically every piano learning method consists of showing students what to practice, and what kinds of techniques (runs, arpeggios, legato, staccato, trills, etc.) are needed. There are few instructions on how to practice order to be able to play them, which is mostly left to the student and endless repetitions (Chang 2009).

Wilson et al. followed their work on *best practices* in scientific computing (Wilson et al. 2014), with a reflection on *good enough* practices (Wilson et al. 2017), in recognition that we must account for what we might reasonably request of practitioners of data analysis. In this manuscript, we consider one component of *good enough* practice in data analysis: *automated testing*.

Automated tests are a formalised way of implementing checks that inputs and outputs of algorithsms are as expected (Wickham 2015). Informative messages are output that assist the developer in identifying where code is not performing as expected.

4 Collaboration via automated testing

At heart, automated tests are collaborative. This may be with others, but applies at least as much to a analyst approaching their own past work with which they have become unfamiliar, or have become anxious about some aspect of the work. Automated tests provide an efficient way of returning to and extending an analysis; anxiety is reduced by having defined outcomes to code explicitly for.

Reproducible research compendia provide a means by which analysts can share their work so that it may be extended by others, and automated tests provide `code::proof` (Gray 2019), a measure of confidence in the algorithm for others. Hayes expresses concern about using algorithms published without automated tests (Hayes 2019). However, only one quarter of the largest repository of R packages, The Comprehensive R Archive Network, have any automated tests (Hester 2016), highlighting that, despite testing identified as a 'vital' part of data analysis (Wickham 2015), automated testing is yet to be widely adopted.

5 A test-driven toolchain walkthrough for in-development analyses

From this section, we now consider the practicality of implementing tests through a *toolchain walkthrough*, an opinionated documetation of a scientific workflow, towards a measure of `code::proof`, confidence in the algorithm implemented (Gray 2019). In particular, a toolchain walkthrough is a reflection of *one* workflow, whilst others, indeed, better, workflows might exist. This is in contrast to a comprehensive review of tools. Instead, a toolchain walkthrough ruminates on considerations *after* a set of tools have been chosen.

Toolchain walkthroughs aim to identify obstacles and advantages in implementation of scientific workflows. By necessity, a toolchain walkthrough is language specific, but, ideally, observations emerge that are useful for any appropriate language employed for data analysis, such as Python. This toolchain walkthrough examines the *process*, analogous to piano *practice*, of implementing tests, as opposed to defining comprehensively the nature of *good enough* tests, analogous to guidance on pianistic technique.

5.1 Objective of this toolchain walkthrough

This toolchain walkthrough aims to provide guidance on implementing a test-driven workflow for an in-development analysis. Many analyses begin as scripts that develop [todo expan] (Riederer 2019). The central question of this manuscript is what constitutes a minimal level of testing for in-development analysis, where code is still being written and features implemented. Automated tests assist in time-consuming debugging tasks (Wickham 2015), but also in providing information with a developer who is unfamiliar with the code.

This is a first effort in identifying the fundamentals of automated testing for the collaborative process of developing an analysis in R. Analgous to Riederer's *RMarkdown-driven development* (Riederer 2019), which deconstructs the workflow of developing an analysis from .Rmd notebook-style reports to packaged analyses, we consider a set a computational tools that form a workflow to assist in the coherent development of automated tests for data analysis. This is an extension of the workflow suggestions provided in *R Packages*, with a specific focus on collaborative workflows in research.

5.2 Devops and assumed expertise

This toolchain walkthrough assumes a knowledge of R advanced enough to be using the language to be answering scientific research claims.

- tools used: testthat, neet, covr
- GitHub

5.3 Case study

5.3.1 `varameta::`

The `varameta::` package is in-development analysis support software for a forthcoming manuscript, Meta-analysis of Medians.

5.4 Get an overview with `covr::`

For an in-development packaged analysis, the `covr::` package provides a means of asssessing the testing coverage for each function. The analysis functions provided by `varameta::` were put on hold while signficant discussion was written for the thesis of which `varameta::` forms a part. Beginning with `covr::package_coverage`, enables us to get a more informed sense of the `code::proof` of this package, and what `code::proof` is still required to declare the analysis done.

The `covr::` function, `::package_coverage`, provides the percentage of the lines of code run in tests and is a succinct method to get an at-a-glance sense coverage of testing scripts. But this assumes we have considered everything that might require testing. Whilst this is informative, it's not comprehensive. A `100%` in all functions in `covr::`, is informative, but is not done.

Consider a function that takes a number x, and outputs $2\log(x)$. To demonstrate this we make a toy package `testtest::` comprising this single function.

```
logfn <- function(x) {
  2 * log(x)
}
```

Now, we write a test that checks the function returns a numeric.

```
library(testthat)
```

```
test_that("function returns numeric", {
  expect_is(logfn(3), "numeric")
})
```

And if we run `covr::package_coverage`, we get `100%` for overall package coverage.

```
testtest Coverage: 100.00%
R/logfn.R: 100.00%
```

But if we were to add this test, for negative numbers.

```
test_that("log function works", {
  expect_is(logfn(-1), "numeric")
})
```

Then `covr::package_coverage` fails.

```
> covr::package_coverage()
Error: Failure in `/tmp/RtmpDjfm3d/R_LIBS654b3cfe6ca/testtest/testtest-tests/testthat.Rout.fail`
library(testthat)
> library(testtest)
>
> test_check("testtest")
  1. Failure: log function works (@test-logfn.R#10)
any(is.na(thing_to_test)) isn't false.

  2. Failure: log function works (@test-logfn.R#10)
any(abs(as.numeric(thing_to_test)) == Inf) isn't false.

  testthat results
[ OK: 8 | SKIPPED: 0 | WARNINGS: 1 | FAILED: 2 ]
1. Failure: log function works (@test-logfn.R#10)
2. Failure: log function works (@test-logfn.R#10)

Error: testthat unit tests failed
Execution halted
```

We pick up the long-neglected `varameta::` package at `70%` package coverage. But as we have seen above, this is informative, but not completely informative. From here we examine how we might use tests to achieve sufficient `code::proof`, confidence in the implementation of the algorithm.

```
library(covr)
package_coverage("~/Documents/repos/varameta/")
#> varameta Coverage: 70.71%
#> R/dist_name.R: 0.00%
#> R/g_cauchy.R: 44.44%
#> R/g_norm.R: 71.43%
#> R/hozo_se.R: 92.31%
#> R/bland_mean.R: 100.00%
#> R/bland_se.R: 100.00%
#> R/effect_se.R: 100.00%
#> R/g_exp.R: 100.00%
#> R/g_lnorm.R: 100.00%
#> R/hozo_mean.R: 100.00%
#> R/wan_mean_C1.R: 100.00%
#> R/wan_mean_C2.R: 100.00%
#> R/wan_mean_C3.R: 100.00%
```

```
#> R/wan_se_C1.R: 100.00%
#> R/wan_se_C2.R: 100.00%
#> R/wan_se_C3.R: 100.00%
```

Created on 2020-02-01 by the reprex package (v0.3.0)

An excellent way to arrive at *done* for an algorithm is to define, even if only broadly, what *done* looks like *before* we set out. We define what `code::proof` is required.

6 Code with intent

When we think of an algorithm, it's easy to feel overwhelmed by the complexity. Here are the relationships between inputs, and just some of the estimators, and outputs in `varameta`, presented as a randomised *graph*, a visual representation of a set of nodes, V, and the edges, $V \times V$, between them. We understand each function's inputs and outputs, but understanding the way they relate to each other, is often muddled without planning. Our `code::brain`, the way we conceptulaise the code, is disorganised.

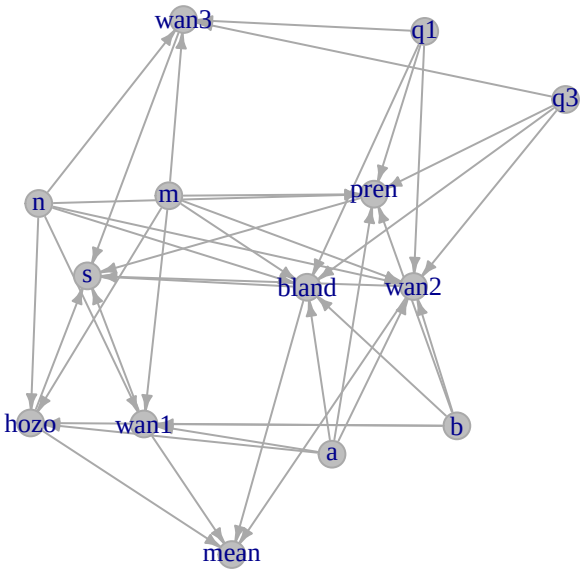


Figure 1: 'code::brain' before 'code::registration'.

```
## NULL
```

The Center for Open Science recommend *preregistering* an experiment, stating what the hypothesis is and how the analyst intends to assess the hypothesis as one safeguard against inadvertent questionable research practices. Analogously, we might think of preregistering our code as providing some `code::proof` of the implementation of our algorithm. In this manuscript, we will think of *code::registration*, stating what the intention of an algorithm is, as - todo: complete

To code::register an algorithm:

6.1 `code::registration`

In an issue on GitHub:

1. Describe the algorithm's intended purpose.
2. Describe the input parameters and how they will be tested.

3. Describe the output parameters and how they will be tested.

Update the code::registration as needed.

6.2 For complex projects

6.2.1 On a piece of paper.

1. Draw a diagram of the inputs and outputs of the algorithm.
2. Draw where this algorithm fits in the pipeline of the package, if appropriate.

6.2.2 Write issue

1. Describe the algorithm's intended purpose.
2. Describe the input parameters and how they will be tested.
3. Describe the output parameters and how they will be tested.

6.2.3 Sketch-diagram

An image or diagram is very help, if not burdonsomely time consuming (but it often is). This section is only useful for those who enjoy graph theory and visualisation, and is not essential.

For those comfortable with *graphs* understood as mathematical objects as a set of vertices from V, and a set of edges $V \times V$, there are visualisation options where the vertices can be tagged with attributes. The code for constructing the following `nodes` and `edges` dataframes has been omitted for brevity, but all code can be found in this manuscript's associated repository `neet` on GitHub.

```
library(tidyverse)
library(ggraph)
library(tidygraph)
```

By tagging the nodes in this graph with the attribute `state` in the algorithm: starting with `input` for the estimators, sample quartiles (which we denote $a, q1, m, q3, b$, in order from smallest to largest); `estimator`, a collection of functions that provide statistical methods for preparing summary medians for meta-analysis; and the `output` of the estimator.

```
nodes
```

```
## # A tibble: 24 x 2
##    node    state
##    <chr>   <fct>
##  1 a       input
##  2 b       input
##  3 q1      input
##  4 q3      input
##  5 m       input
##  6 n       input
##  7 hozo    estimator
##  8 pren_c3 estimator
##  9 pren_c1 estimator
## 10 bland   estimator
## # ... with 14 more rows
```

Edges are specified by a two-column dataframe, `edges`, where each row contains a `to` and `from` vertex identifier, the row number of the `nodes` dataframe.

```
edges
```

```
## # A tibble: 84 x 2
##     from    to
##    <dbl> <dbl>
##  1     5     7
##  2     6     7
##  3     1     7
##  4     2     7
##  5     7    15
##  6     7    16
##  7     5    11
##  8     6    11
##  9     1    11
## 10     2    11
## # ... with 74 more rows
```

These two dataframes cone be converted into a graph object that is compatible with the `igraph::` package and `tidyverse::` syntax.

```
graph <- tbl_graph(nodes, edges)
```

These can be

```
graph %>%
  mutate(state = fct_relevel(state, "output")) %>%
  ggraph() +
  geom_edge_link(arrow = arrow(), colour = "lightgrey") +
  geom_node_label(aes(label = node, colour = state),
                  size = 5,
                  fill = "lightgrey",
                  alpha = 0.6) +
  theme_graph() +
  hrbrthemes::scale_color_ipsum() +
  theme(legend.position = "none",
        panel.background = element_rect(colour = "#ffffe0"),
        plot.background = element_rect(colour = "#ffffe0")) +
  scale_y_reverse() +
  coord_flip()
```

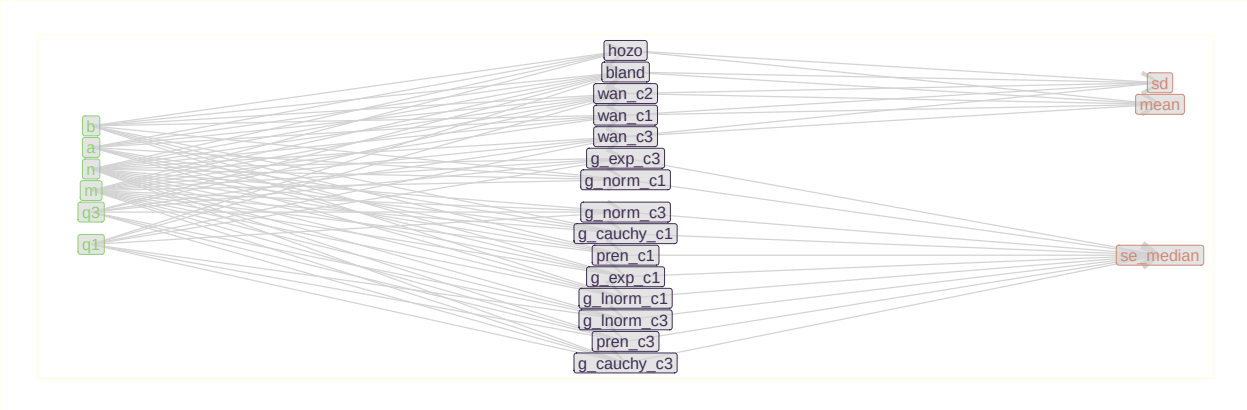
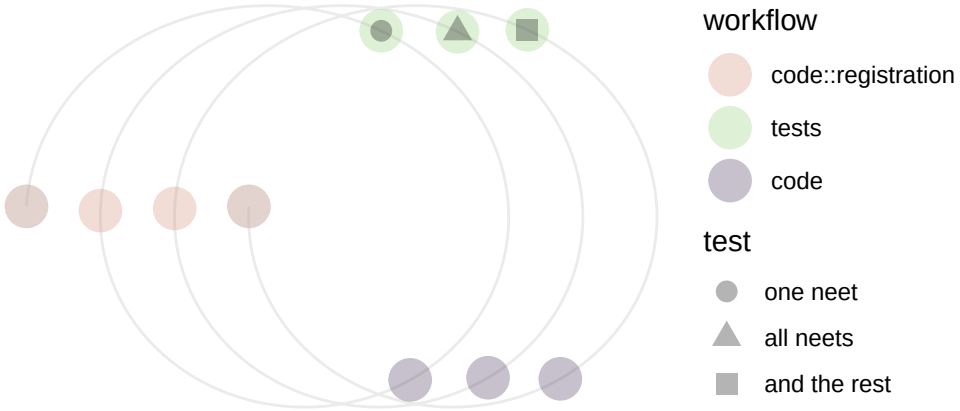


Figure 2: ‘code::brain‘ after ‘code::registration‘.

7 Test-driven workflow



This section describes the test-driven workflow presented above as a `code::proof`ed coding to doneness.

The model workflow includes three stages, repeated three times, before returning to the start, the `code::registration`.

Each phase consists of:

1. `code::registration`
2. tests
3. code

7.1 `code::proof` workflow for coding to doneness

The tests vary each time in complexity, so that the complete model cycle consists of ten phases of work:

1. `code::registration`
2. neet tests, one per function
3. code
4. `code::registration`
5. neet tests, for all inputs for each function
6. code
7. `code::registration`
8. tests, and the rest, i.e., any other cases to test for
9. code
10. `code::registration`

We use *model* cycle to denote the workflow may be adapted for different use-cases. Our next section steps through each phase of work.

8 Coding to `code::proof` toolchain walkthrough

In this section we step through the Coding to `code::proof` for `varameta::`, an in-development analysis that has seen many iterations.

8.1 one `neet` test per function

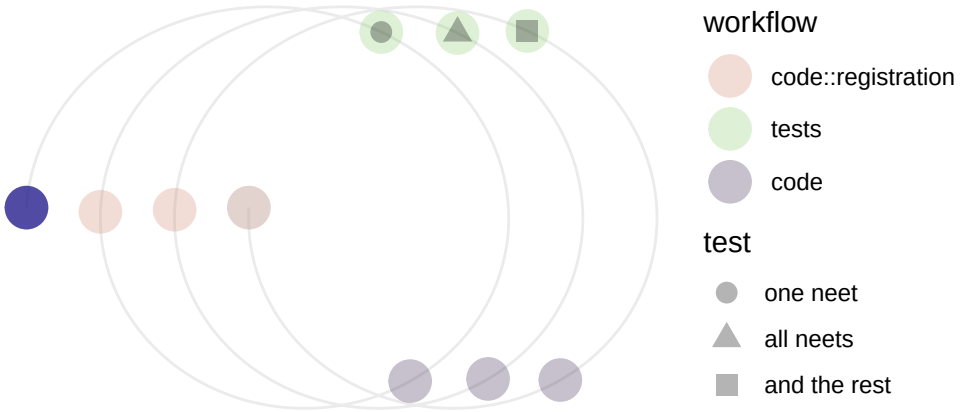
For the first `code::proof` cycle, we follow the steps:

1. A rough, first-draft,`code::registration`.
2. Write one `neet` test per function.
3. Write code to make those `neet` tests pass, for each function.

In this case, as the functions are written, we are *confirming* the functions pass the `neet` tests.

8.1.1 `code::registration`

```
workflow(1)
```



Inputs:

- ☐ one neet[1]
- ☐ all neets
- ☐ and the rest

Outputs:

- ☐ one neet
- ☐ all neets
- ☐ and the rest

8.1.2 At least one `neet` test per function

The `neet::` package is part of this manuscript's research compendia, we think of compendia, a collection of components that form a research project. In the case of this manuscript, there is an extension package of `testthat::`.

```
# install devtools
install.packages(devtools)

# use devtools to install package from github
devtools::install_github("softloud/neet")
```

8.1.3 `code::registration`

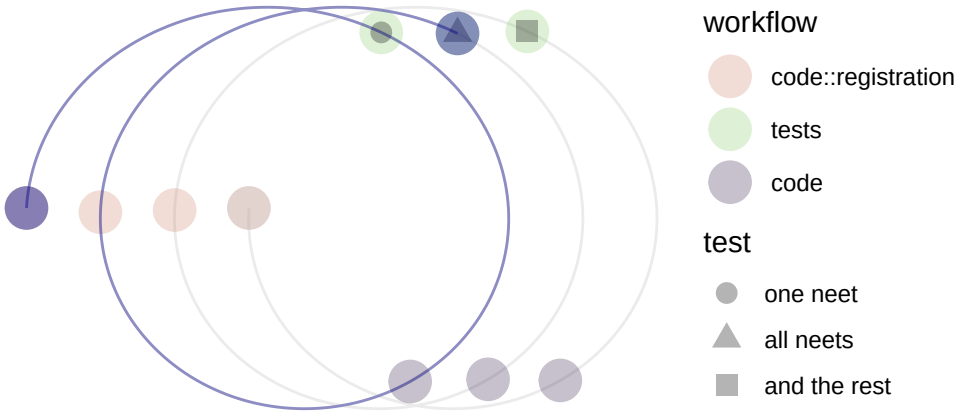
Update or rewrite `code::registration`, as needed.

- ...for the varemta package

[1]Note that `- [ ]` will create an unchecked tickbox and `- [x]` will create a tickbox in both GitHub and RMarkdown.

8.2 `neet` tests for all inputs

```
workflow(5)
```



Once we have established there is one `neet` test per function, we might now perform `neet` tests for each possibility for inputs.

8.2.1 `code::registration`

Update or rewrite `code::registration`, as needed.

8.2.2 `neet` tests for all inputs

8.2.3 Coding to `code::proof`

8.3 Testing for doneness

8.3.1 `code::registration`

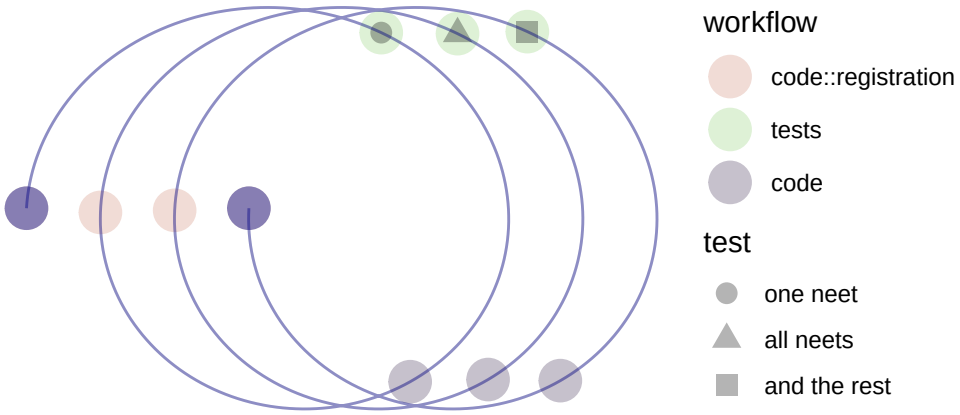
Update or rewrite `code::registration`, as needed.

8.3.2 All tests required for purpose

8.3.3 Coding to `code::proof`

8.4 `code::registration` of doneness

```
workflow(10)
```



Under this workflow, the `code::registration` is the project overview any developer can return to. So, this final registration not only summarises what has been tested, but also notes any idiosyncrasies for other developers, *expeically* the future self of the developer, who recalls so little of the code they may as well be a different developer.

At this point, the developer makes a final assessment of whether the stated objective of the implementation of the algorithm. Assess the `code::proof`, through the development process, new ideas, boundaries to check (recall the `logfn` and negative numbers example in Section 5.4[todo: hyperlink]). We might characterise done as when we feel we have sufficient `code::proof`.

If the algorithm does not yet have sufficient `code::proof`, the cycle can be repeated, and `code::registrations` updates.

9 Sufficient `code::proof` of software for doneness

It does not feel a great leap to assume that we none of us will think of every use case, every aspect, of solving an algorithm computationally. Thus, rather than disputing what is or is not *done*, we simply set out to achieve *what we can think of doing*. A `code::registration` provides a summary of what we could think of doing to 'further the process of scientific discovery', as Devezer *et al.* might put it (Devezer et al. 2019), towards our goal.

The first version of this manuscript, itself, is a preregistration of a computational workflow. This workflow is for `team reprocat::` of The repliCATS Project to develop rerproducible algorithms more happily together. The first version provides a description of the workflow for the team; the second version incorporates the team's feedback. This arguably literalises the scientific practices of publishing manuscripts; no manuscript, no matter how definitive, provides everything required for a resaerch project, but may offer particular tools and evidence to further the process of scientific discovery.

Another researcher software engineer may be cognisant of a tool you are unfamiliar with. By providing the package, and the `code::registration`, we provide a method to facilitate what Stodden describes as the *extensibility* of the algorithm (Stodden, Borwein, and Bailey 2013).

The workflow presented in this manuscript aims to address: `code::xiety`, the anxiety associating with programming; *good enough* scientific practices, by incorporating automated testing (Wilson et al. 2017); and collaborative benefits. `code::registration` provides a means of providing accessibility and transparency to the next developer. *Especially* if the next developer is your future self.

References

Bryan, Jennifer. 2017. "Excuse Me, Do You Have a Moment to Talk About Version Control?" *PeerJ*

PrePrints 5: e3159.

Chang, Chuan C. 2009. *Fundamentals of Piano Practice*.

Devezer, Berna, Luis G. Nardin, Bert Baumgaertner, and Erkan Ozge Buzbas. 2019. "Scientific Discovery in a Model-Centric Framework: Reproducibility, Innovation, and Epistemic Diversity." *PLOS ONE* 14 (5): e0216125. https://doi.org/10.1371/journal.pone.0216125.

Fraser, Hannah, Tim Parker, Shinichi Nakagawa, Ashley Barnett, and Fiona Fidler. 2018. "Questionable Research Practices in Ecology and Evolution." *PLOS ONE* 13 (7): e0200303. https://doi.org/10.1371/journal.pone.0200303.

Galway, James. 1990. *Flute*. Kahn & Averill.

Gray, Charles T. 2019. "Code::Proof: Prepare for Most Weather Conditions." In *Statistics and Data Science*, edited by Hien Nguyen, 22–41. Communications in Computer and Information Science. Singapore: Springer. https://doi.org/10.1007/978-981-15-1960-4_2.

Hayes, Alex. 2019. "Testing Statistical Software - Aleatoric." Blog.

Hester, Jim. 2016. "Covr: Bringing Test Coverage to R."

Karp, Paul. 2019. "Robodebt: The Federal Court Ruling and What It Means for Targeted Welfare Recipients." *The Guardian*, November.

Marwick, Ben, Carl Boettiger, and Lincoln Mullen. 2018. "Packaging Data Analytical Work Reproducibly Using R (and Friends)." e3192v2. PeerJ Inc. https://doi.org/10.7287/peerj.preprints.3192v2.

McIlroy, T. 2017. "20,000 People Sent Centrelink 'Robo-Debt' Notices Found to Owe Less or Nothing." *Canberra Times* 13.

Riederer, Emily. 2019. "RMarkdown Driven Development (RmdDD)." *Emily Riederer*.

"RStudio Cloud." n.d. https://rstudio.cloud/learn/primers.

Stodden, Victoria, Jonathan Borwein, and David H. Bailey. 2013. ""Setting the Default to Reproducible" in Computational Science Research." *SIAM News* 46 (5).

Wickham, H. 2015. *R Packages: Organize, Test, Document, and Share Your Code*. O'Reilly Media.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. "Best Practices for Scientific Computing." Edited by Jonathan A. Eisen. *PLoS Biology* 12 (1): e1001745. https://doi.org/10.1371/journal.pbio.1001745.

Wilson, Greg, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2017. "Good Enough Practices in Scientific Computing." Edited by Francis Ouellette. *PLOS Computational Biology* 13 (6): e1005510. https://doi.org/10.1371/journal.pcbi.1005510.

Yuan, Ding, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems." In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 249–65. OSDI'14. Broomfield, CO: USENIX Association.