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

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

Charles T. Gray, IMeRG*   Elise Gould, QAECO/IMeRG
David Wilkinson, QAECO/IMeRG   Aaron Willcox, IMeRG

Abstract

Despite centuries' accumulation of pedagogy on pianistic technique, there remains a comparative dearth of literature conceptualising the process of obtaining sufficant technical ability. Analogously, a toolchain gap exists between between good enough practices of scientific computation and implementation. A toolchain walkthrough, an opinionated documentation of a scientific workflow, provides a means of addressing this gap. In this toolchain walkthrough, we present a methodology for coding to a measure of `code::proof`, a sense of good enough confidence in the implementation of a statistical algorithm.

1 Coding to `code::proof`

J. S. Bach's Contrapunctus I, a four-part work of counterpoint from *The Art of Fugue*, 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 the piece will never feel *done*. To play well, a pianist must employ technique, 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::register`ed (`c::red`[1]) **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 implementation of the algorithm.

Musicians and, similarly, athletes, do not see themselves as having mastered a skill, but as 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 facilitates the analyst achieving sufficient `code::proof` in their algorithm, confidence in the implementation of the algorithm (Gray 2019).

---

[1]In this manuscript, `c::r` denotes `code::registration`, `c::ring` denotes `code::registering`, and `c::red` denotes `code::registered`

1.1 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 heartbreaking ongoing example of this kind of problem is the income-reporting data analysis 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 signalled 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 we 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 towards a fully reproducible packaged research compendium. Furthermore, these workflows assist the developer to 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.

1.2 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 an analyst, in reproducible computing for research data analyses. The objective is different from advanced pianism, but we characterise testing *practice*, as opposed to *technique*, analogous to how Chang 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: cite], and Czerny's *School of Velocity* [to do: cite]. 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 successful adoption of these techniques. In science, we might call this a *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 in 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 data analysis practitioners. 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 algorithms are as expected (Wickham 2015). Informative messages are output that assist the developer in identifying where code is not performing as expected.

1.3   Collaboration via automated testing

At heart, automated tests are collaborative. This may be with others, but applies at least as much to an analyst approaching their own past work with which they have become unfamiliar, or have become anxious about some aspect of their work. Automated tests provide an efficient way of returning to and extending an analysis; anxiety is reduced by having defined outcomes to explicitly code 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 being identified as a 'vital' part of data analysis (Wickham 2015), automated testing is yet to be widely adopted.

2   A toolchain walkthrough for test-driven analysis workflow

From this section, we now consider the practicality of implementing tests through a **toolchain walkthrough**, an opinionated documentation of a scientific workflow, towards a measure of `code::proof`, or confidence in the algorithm implementation (Gray 2019). In particular, a toolchain walkthrough is a reflection of *one* workflow, whilst other, potentially better, workflows might exist. This is in contrast to a comprehensive review of tools. Instead, a toolchain walkthrough ruminates on considerations *given* 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.

2.1 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 for a developer who is unfamiliar with the code (including new users *and* existing developers returning after some level of temporal distancing).

This is a first effort in identifying the fundamentals of automated testing for the collaborative process of developing an analysis in R. Analogous to Riederer's *RMarkdown-driven development* (Riederer 2019), which de-constructs 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.

2.2 Devops and assumed expertise

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

- tools used: testthat, neet, covr
- GitHub

2.3 A toolchain walkthrough for `c::r` of...

This toolchain walkthrough for `c::r` reflects on the in-development practice of code to completion on several different research projects. Consideration of sufficient `code::proof` of packages `varameta::` and `simeta::`, developed for the oeuvre of doctoral work this manuscript contributes to. Another example project is considered, The repliCATS Project software of statistical aggregation methods, in particular `CATnap::` and `aggreCAT::`.

For The repliCATS Project, the algorithms are not available until the research is complete, in order to blind the other team to the methodology. Although we cannot provide explicit code examples, the process is considered with respect to this project, as it is an example of collaborative in-development `c::r`.

2.3.1 `varameta::`

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

2.3.2 `simeta::`

Words words words

2.3.3 repliCATS Project

Moar words words words

3 `neet::` tests

The workflow described in this toolchain walkthrough differentiates between ...

3.1 The `neet::` package

```
library(neet)
```

A **neet test** checks a function outputs a **non-empty** thing of **expected type**. This is what is referred to as a **boundary condition** test in RStudio's primers (todo: citation).

These expectations are intended to integrate into minimal testing workflow *for the development* of data analyses. When developing a function, we will need parameters, and structure of the pipeline. These tests enable the developer to feel reassured the pipeline's functions are outputting non-empty things of expected type, while the developer decides the best structure for an analysis pipeline.

A `character` string will be checked for being of string-length > 1.

A `numeric` is checked for not being `NA`, `NULL`, `Inf`, or `-Inf`.

A 'list" is checked for being of length > 1.

A `data.frame` is checked for having at least one row.

A `ggplot` object is checked for successfully running.

3.2 Installation

You can install the released version of **neet** from GitHub with:

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

# Use devtools to install package from GitHub
devtools::install_github("softloud/neet")
```

3.2.1 Example

This is a basic example which shows you how to solve a common problem:

```
library(testthat)
library(neet)

## Basic example code

# Expectations check output of code in testthat files
expect_neet(3, "numeric") # test numeric
expect_neet("cat", "character") # test character string
expect_neet(mtcars, "data.frame") # test data frame

# Assertions check inputs of code in function scripts
assert_neet(3, "numeric") # test numeric
```

```
## [1] TRUE
```

```r
assert_neet("cat", "character") # test character string
```
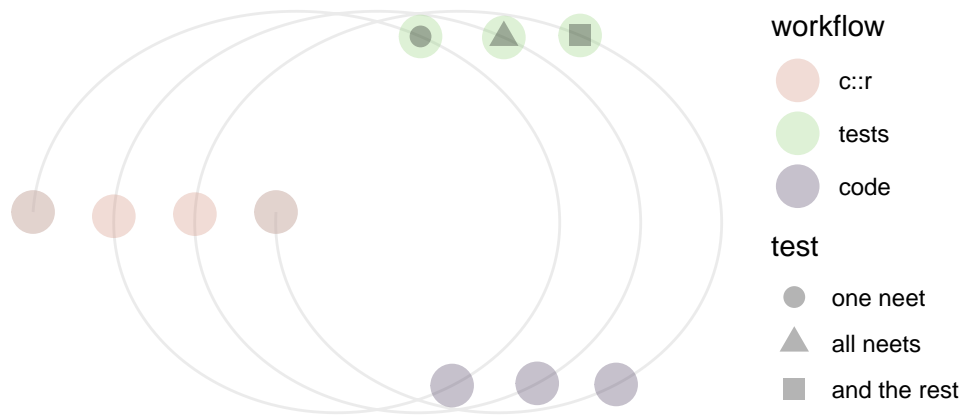
```
## [1] TRUE
```

```r
assert_neet(mtcars, "data.frame") # test data frame
```

```
## [1] TRUE
```

4 `code::proof` test-driven workflow

In order to... doneness ... propose test driven ... implemented as...

The model begins with a `code::registration`, an outline of the intended algorithm, its inputs, and outputs.

A workflow includes three stages, repeated three times, before returning to the start, the `c::r`.

Each phase consists of:

1. `c::r`
2. tests
3. code

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

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

4.2 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.

- `simeta::`
- `repliCATS`

5 code::registration (c::r)
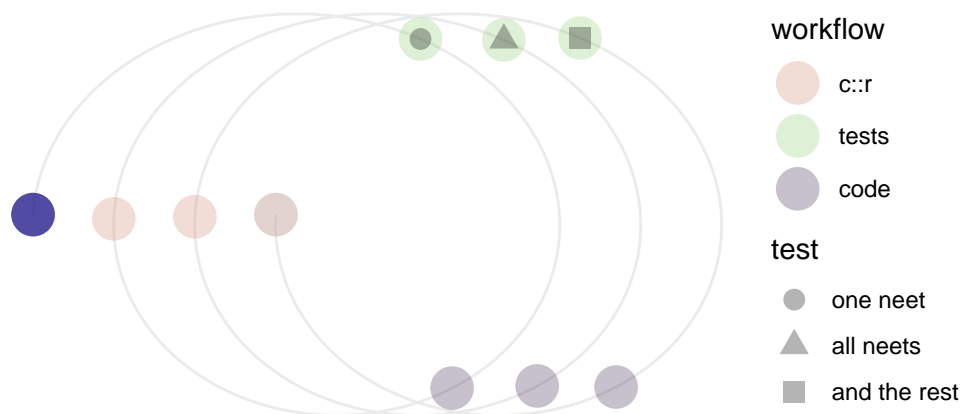
We begin with a sketch outline of a `code::registration` (`c::r`), which we then populate with description in prose, provide some detail of neet testing, before providing case-studies.

As our first testing objective is one neet test per function, we need not necessarily work out the details of all possible things to check for. We simply ask, what will it take to check that each function required for the algorithm produces a non-empty thing of expected type? For the second `code::proof` cycle, we consider all possible inputs, and write tests, and then code for those.

The final phase, which we need not detail immediately, covers testing for anything else we may wish to check. This might be checking there are positive numbers output by the function. Or it might be rigorously checking that the algorithm matches the output of another person's instantiation of the algorithm, as is the case with the `aggreCAT::` package.

1. Objective:

2. Inputs:

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

3. Outputs:

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

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

For the examples below, in the interest of brevity, we will drop the tick boxes, and provide the roughly descriptive prose for the three sections of each `c::r`. Under this workflow, we expect to update the `c::r` as our understanding of the algorithmic structure becomes more nuanced.

5.1 `varameta:: c::r`

```
neet::workflow(1)
```
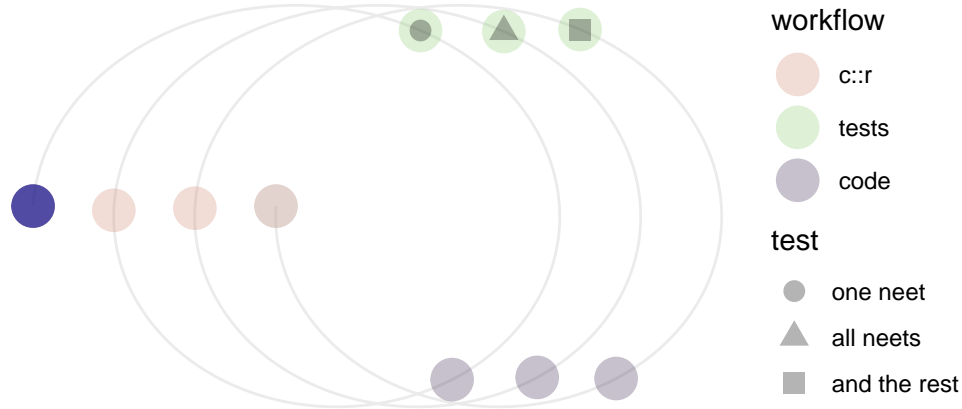

The `varameta::` package has the least complex structure of the examples we are considering.

Objective

The `varameta::` package aims to provide a collection of different estimators for approximating the variance of the sample median.

Inputs

Each estimator takes various sample statistics (quartiles, etc.), sample size.

☐ one neet
☐ all neets
☐ and the rest

Outputs

Each estimator outputs an estimate of spread and centre. Some are approximations of the variance of the sample median, and others approximate the standard deviation and mean.

☐ one neet
☐ all neets
☐ and the rest

5.2 `simeta:: c::r`

The `simeta::` package is being developed to evaluate the efficacy of the estimators provided in the `varameta::` package.

Objective:

Randomly generate meta-analysis data, calculate a confidence interval based on a user-specified estimator (currently only works on `varameta::`), and evaluates if the true parameter falls within; repeat this |trials| number of times, and return the coverage, the proportion of times the true parameter falls within the confidence interval.

Summarise and provide a dataframe of coverage for different sample sizes, group effect ratios, variability between studies, and distributions. Also provide summary visualisation tools.

☐ Initial objective: provide analysis support for the meta-analysis of medians paper.

☐ Secondary objective: allow the user to specify the estimator.

Inputs:

User inputs: default to *just* the estimator.

Outputs

A dataframe of coverage probabilities.

5.3 `aggreCAT::`

As the `aggreCAT::` package forms part of a pipeline of packages designed to build a continuously integrated analysis for living data, there are more detailed supplementary materials detailing requirements for inputs and outputs to turn to, thus the `c::r` is somewhat brief on first pass.

Objective

Provide export-ready data for each of the 19 aggregation functions.

Inputs

Expert judgements to be aggregated. Some special functions take other supplementary datasets, too.

Outputs

A wrapper function outputs each dataset in wide client-ready format.

6   First cycle: one `neet` test per function

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

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

If functions are already written, then we only need to *confirm* the functions pass the `neet` tests.

7   Second `code::proof` cycle

8   Third `code::proof` cycle

8.1   And the rest

Now that we have neet tests for each function, we might ask what else needs to be tested.

In the case of `varameta::`, we might calculate a fixed-value output and compare it, but calculations rapidly become too complex to calculate explicitly.

In the case of `aggreCAT::` for the repliCATS Project, we have several aggregation methods to compare. In a large-scale collaborative project such as this, we have several different researchers contributing methods which the Reproducibility Team are engineering as open-source research software. In this case, the algorithms

are being computationally implemented in software by different researchers than those who conducted the initial analyses. And that means, with a bit of collating, we have access to many testing datasets.

In Figure @ref(fig:aggregators), we present the results of in-development tests on the aggregation methods developed for The RepliCATS project.

```
knitr::include_graphics("cs_diff_by_method_id.png")
```

8.2 Test coverage with `covr::`

- preamble

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
```
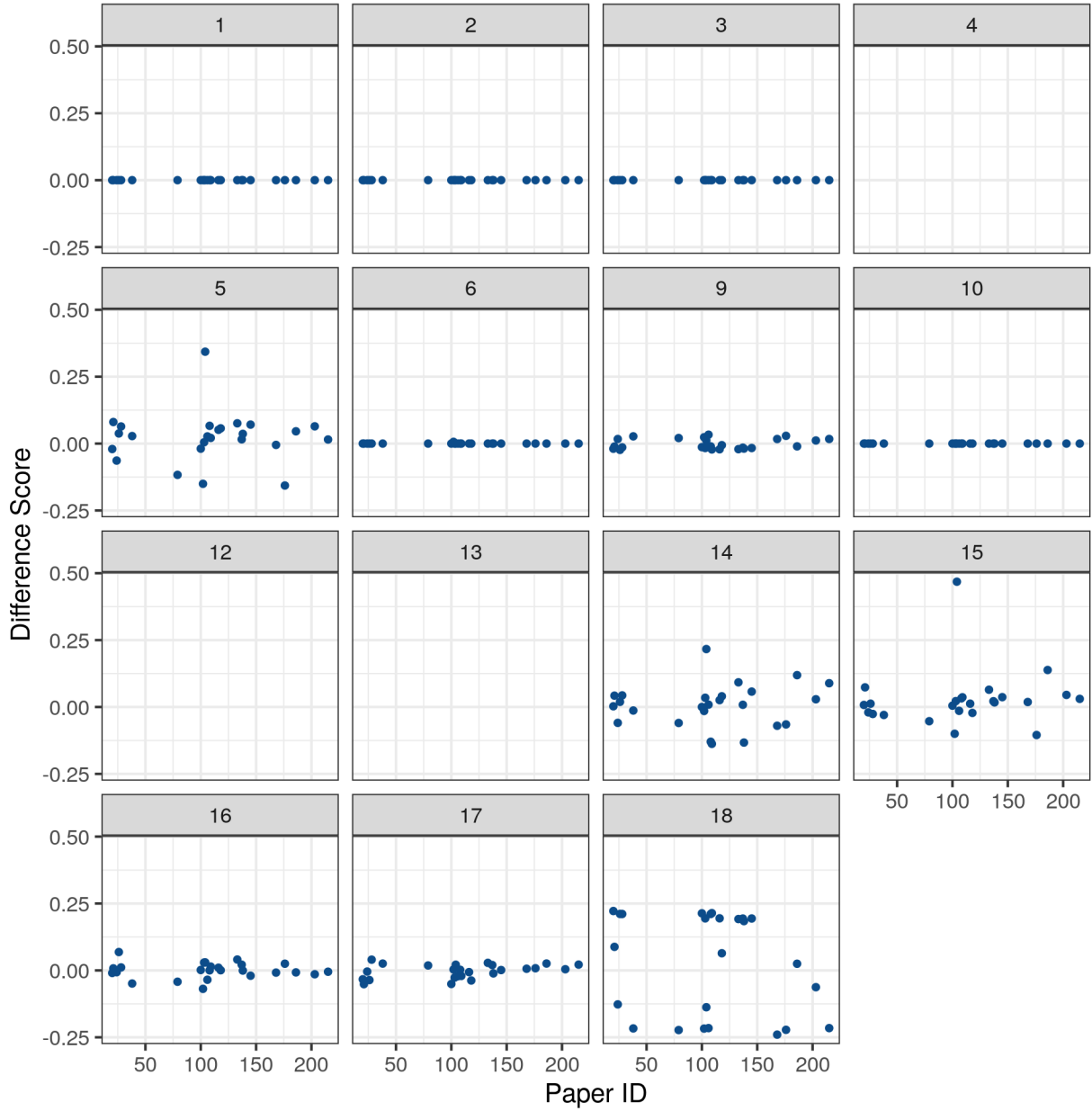
Figure 1: After neet tests are written, a comparison of aggregation methods with other researchers' analyses. Methods such as 1,6, 10, are in close agreement. There are some methods that are relatively close, such as 9 and 17. These may be deemed close enough, in consultation with the researchers who developed the methods. Then there are methods to investigate, such as 18 and 16, where there is variability between the two analyses. The aggregation methods are deidentified and the code for this visualisation will not be available until the repliCATS Project is complete, to blind the other research teams on the SCORE Program to the methods used for analysis.

```
[ 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
```

8.3 `aggreCAT::` from the repliCATS Project

A luxury afforded in this large-scale collaborative project is that the software implementation is implemented by different analysts than those who developed the statistical algorithms. Different analysts developed different algorithms, and thus, when the research software engineers on the Reproducibility Team came to implement the algorithms there is a wealth of pre-existing analyses to benchmark the software against.

For a small-scale analysis, the method and implementation are often developed by one person, with minimal hands-on input from others, as is the case with the `simeta::` and `varameta::` packages also discussed. However, when the development and implementation are divided between people, an opportunity for quality assurance presents itself.

This is analogous to the quality assurance for data entry suggested by Yenni *et al.* (Yenni et al. 2019). To update a living data set, collected in the field, the authors suggest two researchers perform the data entry into different spreadsheets, which are then put through text comparison algorithms to identify typos and inconsistencies.

In the case of `aggreCAT::`, there are nineteen aggregation methods, each of which has been contributed by a researcher. Thus, each researcher who developed a method has provided a testing dataset for the Reproducibility Team to compare the results of the software implementing the aggregation methods.

8.4 `c::r` of doneness

```
workflow(10)
```
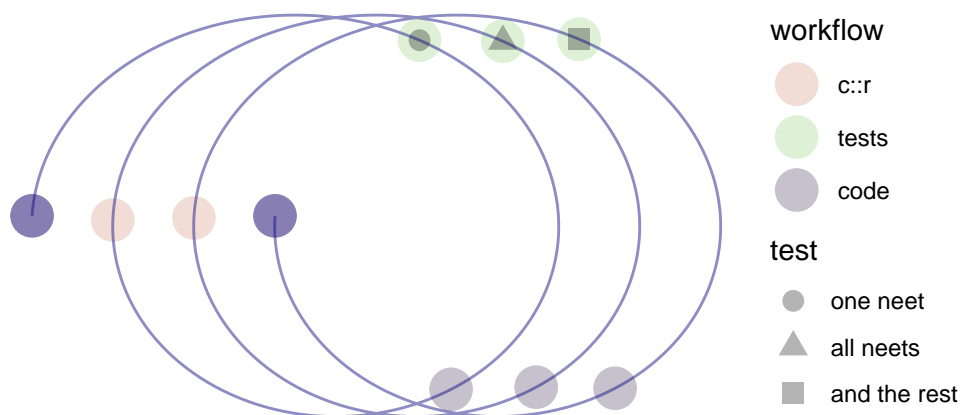

Under this workflow, the `c::r` is the project overview any developer can return to. So, this final registration not only summarises what has been tested, but also notes any idiosyncrasies for other developers, *especially* the future self of the developer who recalls so little of the code they may as 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 has been reached. 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 `c::rs` 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 `c::r` 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 pre-registration of a computational workflow. This workflow is for `team reprocat::` of the repliCATS Project to develop reproducible 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 research project, but may offer particular tools and evidence to further the process of scientific discovery.

Another research software engineer may be cognisant of a tool you are unfamiliar with. By providing the package, and the `c::r`, 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. `c::r` provides a means of providing accessibility and transparency to the next developer. *Especially* if the next developer is your future self.

10   older draft

10.1   Get an overview with `covr::`

For an in-development packaged analysis, the `covr::` package provides a means of assessing the testing coverage for each function. The analysis functions provided by `varameta::` were put on hold while significant 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.

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.

11 Code with intent

When we think of an algorithm, it's easy to feel overwhelmed by the complexity. And, analogous to the pitfalls of the piano student, a developer may fall prey to the inevitable rabbit holes and distractions that arise. It's tempting to consider these distractions benign, but, in fact, these actions are frequently the result of technical anxiety about the task, and are anathema to mastering the technique in question.

11.1 Practice with intent

In learning Bach's Contrapunctus I, the student has various conventional, albeit questionable, methods of approach.

The student might begin from the start each time. But, as the piece is not yet mastered, the student will inevitably encounter something they cannot play with ease. This is often encountered in the first bar of a Bach piece. If the student plays up to the mistake, and begins again, then they risk becoming adept the start, and becoming increasingly anxious about the end of the piece.
In an advanced piece such as Contrapunctus I, there are many challenging technical aspects. In particular, the work is polyphonic; Contrapunctus I is a central focus of the string quartet in Vikram Seth's *An Equal Music* [todo citation]. There is a melody; formally referred to as a *subject.*

> todo: figure of subject

This figure reoccurs in bars....

> Piano teachers know that students must practice musically in order to acquire technique. Both musicality and technique require accuracy and control. Practically any technical flaw can be detected in the music. Nonetheless, many students tend to practice neglecting the music, preferring to "work" when no one is around to listen. Their reasoning is, "I'll practice non- musically (which is easier because you can shut off the brain) until I can play it well, then I'll add the music." This never works because learning piano is all about training the brain, not finger callisthenics (Chang 2009).

11.2 Mindful coding

In the `varameta::` package, we wish to compare various estimators. There are inputs, an estimator, which is to say a computational process and a similar output, however the devil is in the details, as

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 conceptualise the code, is disorganised.
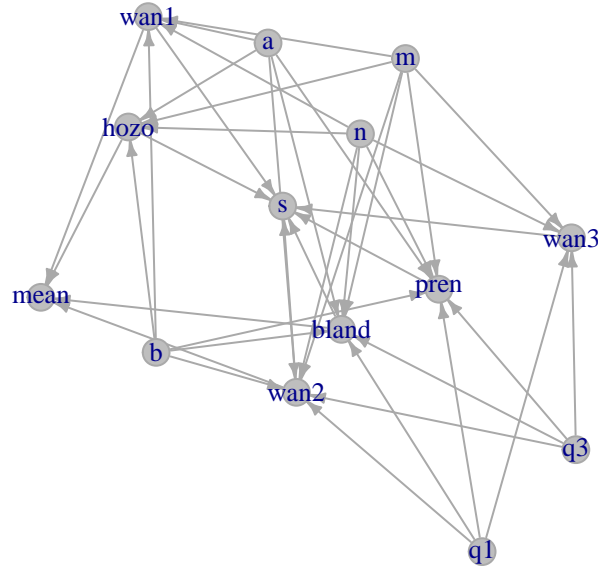
```
## NULL
```

Figure 2: 'code::brain' before 'c::r'.

The Center for Open Science recommend *pre-registering* 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 pre-registering our code as providing some `code::proof` of the implementation of our algorithm. In this manuscript, we will think of *c::r*, stating what the intention of an algorithm is, as - todo: complete

To code::register an algorithm:

12 `code::registration (c::r)`

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 c::r as needed.

12.1 For complex projects

12.1.1 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.

12.1.2 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.

12.1.3 Sketch-diagram

An image or diagram is very helpful, if not burdensomely 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 can be converted into a graph object that is compatible with the `igraph::` package and `tidyverse::` syntax.

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

These can be

```r
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()
```
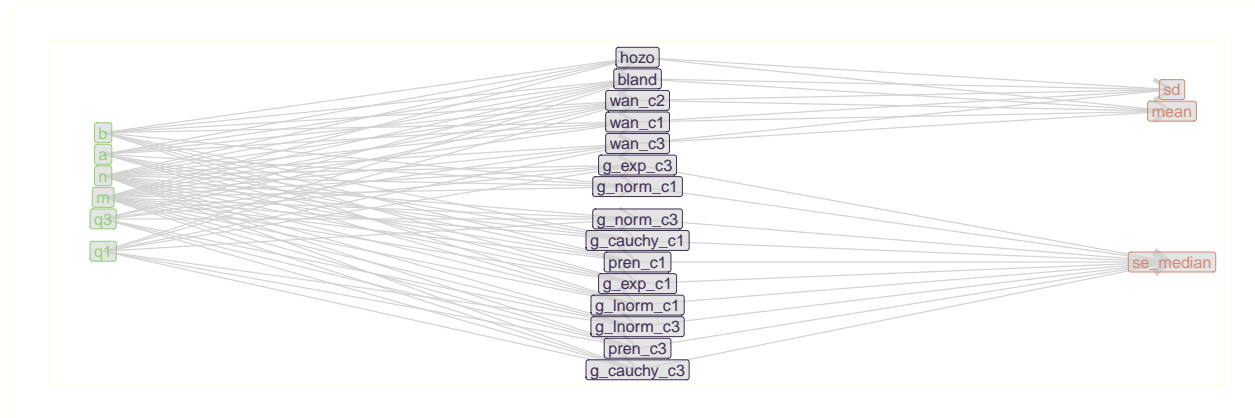

Figure 3: 'code::brain' after 'c::r'.

13 Workflow for algorithmic structure

- hard to find ap lace t ostart

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::`.

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

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

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.

Yenni, Glenda M., Erica M. Christensen, Ellen K. Bledsoe, Sarah R. Supp, Renata M. Diaz, Ethan P. White, and S. K. Morgan Ernest. 2019. "Developing a Modern Data Workflow for Regularly Updated Data." *PLOS Biology* 17 (1): e3000125. https://doi.org/10.1371/journal.pbio.3000125.

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.