RSPARROW v2.0 code review

Marcus W. Beck

Oct 1, 2023

Thanks again for the opportunity to conduct the code review of *RSPARROW* v2.0. I believe this will be an incredibly valuable piece of software for the water management community and I commend the team for adapting the tool to an open source environment. I also appreciate the responsiveness to my comments provided during beta testing of v2.0, specifically regarding the removal of the *maptools* and *rgdal* dependencies.

The master branch containing *RSPARROW* v2.0 was downloaded to my personal computer on September 26th. I installed the downloaded version as during beta testing by creating an RStudio project in the unzipped directory and downloading the required R dependencies following the console prompts after running `devtools::load_all('RSPARROW_master')`. Also note that my beta testing used R version 4.2.3 and my current R installation uses v4.3.1, as was included in the repository.

The remainder of this document provides the code review after running the calibrated total phosphorus model for Tampa Bay, as used in the file *UserTutorialDynamic/results/sparrow_control.R*. I have also used the ROpenSci review template. Items not marked are elaborated in the review comments.

Package Review

- **Briefly describe any working relationship you have (had) with the package authors.** I have been involved with this project since 2019 as an end user at the Tampa Bay Estuary Program. My involvement has been testing development versions of *RSPARROW* and sharing these tools with our management conference.

☒ As the reviewer I confirm that there are no conflicts of interest for me to review this work.

Documentation

The package includes all the following forms of documentation:

- ☒ **A statement of need:** clearly stating problems the software is designed to solve and its target audience in README
- ☒ **Installation instructions:** for the development version of package and any non-standard dependencies in README
- ☒ **Vignette(s):** demonstrating major functionality that runs successfully locally
- ☐ **Function Documentation:** for all exported functions
- ☐ **Examples:** (that run successfully locally) for all exported functions
- ☒ **Community guidelines:** including contribution guidelines in the README or CONTRIBUTING, and DESCRIPTION with `URL`, `BugReports` and `Maintainer` (which may be autogenerated via `Authors@R`).

Functionality

- ☒ **Installation:** Installation succeeds as documented.
- ☒ **Functionality:** Any functional claims of the software been confirmed.
- ☒ **Performance:** Any performance claims of the software been confirmed.
- ☐ **Automated tests:** Unit tests cover essential functions of the package and a reasonable range of inputs and conditions. All tests pass on the local machine.

Review Comments

Code and file structure issues

There are several .Rmd files in the R folder. I suggest moving these `inst/doc` so that only R files are included in the former. These all look like parameterized templates or child files, which conventionally are not included in in the `R` folder. This will also allow for greater transparency of the package since users can quickly view the location of these templates. For example, the location of a template in the `inst/doc` folder could easily be found as follows if placed in `inst/doc`:

```
system.file("doc/checkDrainageareaErrors_station.Rmd", package = "RSPARROW")
```

Related, the `batch` subdirectory in the root directory of `RSPARROW_master` aren't conventional folders for a package. I believe the best practice is to place them in `inst` and reference the associated files accordingly. However, it looks like the `batch` folder includes R code, so it might be better to create them as functions and place them in the `R` folder. This would also

eliminate the need to `source()` them directly. There are some `.RData` files in the folder that might also be better placed in a `data` folder in the root directory.

There are a few instances where it looks like the global settings of the R session are altered, which may not necessarily cause problems, but is discouraged for package development. For example, there are several locations where it looks like the global `options()` are changed (e.g., in `batchGeoLines.R`, `batchlineShape.R`, etc.) and it's unclear if the default options are reset after function execution. I'm guessing this isn't an issue if a user is following methods in the documentation, but it may cause unintended artifacts if additional R analyses outside of the package are used.

The general coding style for the functions don't always follow best practices. This is not critical, and I leave it up to authors on how/if they want to address, but I've noted a few issues below that are worth considering. Most of them relate to changes to improve readability.

- Add spaces around mathematical operators and the assignment operator, e.g., `x <- 2 / 3` and not `x<- 2/3`. Same for commas, e.g., `data <- data[order(data$hydseg), ]` and not `data <- data[order(data$hydseg),]`.
- `if/else` statements should also include spaces:

  ```
  if (something) {
   # do something
  } else {
   # do something else
  }
  ```

  and not

  ```
  if(something){
   # do something
  }else{
   # do something else
  }
  ```

- Many function names uses camelcase (e.g., `estimateBootstraps`), consider using snake-case instead (`estimate_bootstraps`).
- A simple tool to quickly modify code-styling is the `styler` package. I would try using this instead of manually verifying the styling if you choose to modify any of the styling.

Some functions include arguments that are commented. These should be removed, e.g., `predictMaps.R` includes unused arguments called `Rshiny`, `regular`, and `scenarios`.

Some functions include internal functions that are probably better created as standalone utility functions in their own `.R` files. This can help improve debugging and transparency of coding. For example, line 157 in `executionTree.R` includes an internal function called `unFormat()`.

In `hyseq.R`, lines 42, 43, 56, and 57 use the `aggregate()` function that applies `min()` or `max()` as `function(x) min(x)` or `function(x) max(x)`. Why not just supply `min()` or `max()` to the `FUN` argument instead of using a generic function as a wrapper?

In `shinyMap2.R`, there are multiple `library()` calls to load the dependent packages (lines 90-108). The standard way to load dependencies is using the Roxygen documentation, e.g., `@import leaflet` or `@importFrom` to import selected functions from a package as opposed to the entire package. Defining dependencies this way ensures the `NAMESPACE` file for the package includes the correct imports. In fact, the `NAMESPACE` file is completely empty. This file is typically auto-generated using `devtools::document()`, where the imports are identified from the `DESCRIPTION` file and the Roxygen comments. An accurate `NAMESPACE` file is critical to ensure there are no conflicts with other packages that may be loaded in the current session.

There are several warning and error messages that might be better handled using the base R functions `warning()` and `stop()`, respectively. For example, line 34 in `checkMissingData1Vars.R` simply uses a text string with `WARNING`. If there's good reason to not use the base R functions, then you can leave as is, but I have rarely seen cases where there use is not preferred. For example, `stop()` will return an error and immediately stop execution of a function given a critical error preventing further execution, whereas the error messages in `RSPARROW` simply print (or use `cat`) while continuing execution. It looks like the intent is to create an error log. This makes sense given how most modelling tools are programmed, but it's not entirely consistent for R packages.

Documentation

In addition to my comments below about general documentation needs for future versions, you might consider simplifying some of the Roxygen parameter components for functions that share the same parameters. For example, `estimateFeval.R` and `estimateFevalNoadj.R` appear to share the same parameters, so instead of duplicating the `@param` entries between the two, you could use `@inheritParams estimateFeval` in the Roxygen for `estimateFevalNoadj.R` (or vice versa) to copy the parameter entries in the rendered `.Rd` files.

Versioning

I noticed some potential issues with the versioning used by *RSPARROW*. Since this code review is meant to cover changes in version 2.0 compared to the previous, I was hoping to find some information in the version control on GitLab to easily compare 2.0 with previous versions. There wasn't anything obvious in the repository indicating the version releases, nor were there any clear indications in the commit logs or repository graphs. The only reference I found to previous versioning was a tag for v1.0.0 on a commit from October 2019. The previous code

review by L. Platt was for version 1.1 and I could not find any indication in the repository where this version ended and where version 2.0 started. Could the authors retroactively tag the different versions in the commit log? Or alternatively provide a change log between the versions (example from dplyr)? The latter would be a good addition to the user manual.

Considerations for future versions

Most of my comments relate to expected future developments of *RSPARROW* and, as such, do not need to be addressed at this time. I provide them here for consideration for the developers moving forward.

I strongly encourage that *RSPARROW* be developed as a standalone package made available on CRAN. The user workflow for downloading, installing, and using the package is unconventional compared to other R packages. I realize the current design was intentional given the large number of dependencies, both for R packages and supporting datasets for calibrated models. However, the workflow required by this format is unfamiliar to most R users and there will be a substantial learning curve to using the model (although, again, I commend the authors for providing sufficiently detailed documentation in the user manual). If the authors are able to further develop *RSPARROW* as a standalone R package, the following could be more easily implemented:

- **More complete function documentation**: Although the functions include typical components of Roxygen documentation (e.g., function title, parameter descriptions, etc.), many other components are missing, such as example code or details on the returned values.
- **Limit the number of R package dependencies**: Although there is no hardset rule, most R packages include no more than ten or so dependencies in the DESCRIPTION file. *RSPARROW* v2.0 includes 40 R packages in the depends field. This not only increases installation time, it also makes *RSPARROW* vulnerable to missing packages. It is not uncommon for packages to be removed from CRAN or existing dependencies have newer versions with breaking changes. *RSPARROW* will have a greater chance of being successfully maintained and usable in the future if the authors can greatly reduce the number of dependencies. Perhaps start by identifying which dependencies are used the least and determining if they are absolutely necessary. Sometimes entire functions from other packages can be copied/reused in the current package to remove a dependency.
- **Add unit tests**: I know this was mentioned by L. Platt, but I also encourage the authors to include specific unit tests (using testthat) to provide greater assurance that the code does what it's intended to do. I realize that the user tutorial models are compared using `compareModels()`, which is sufficient for version 2.0, but a future version could use the more conventional unit tests that are common in R packages.

- **Remove or reduce size of large files**: The *RSPARROW* package directory is 34.5 MB and most packages are < 5MB. This size requirement is really only needed if the authors intend to submit the package to CRAN, but reducing the file size will also reduce the package load time. Some large files I noticed were `batch/shinyBatch.RData` (13.5 MB), `inst/doc/RSPARROW_docV1.1.0.pdf` (4.5MB), `inst/doc/RSPARROW_docV2.0.pdf` (6.1MB), and the subdirectory `inst/doc/figures` (4.74 MB). The latter three should not be included with the package since and could be hosted elsewhere/linked in the vignette.
- **Migrate to GitHub**: Future versions of *RSPARROW* could migrate to GitHub since the source code for the current version on GitLab can only be viewed by verified users. This might only be possible once *RSPARROW* is developed as a bona fide R package with minimal file sizes (see previous comment). This would also streamline any community feedback in the issues (as currently suggested at https://github.com/USGS-R/RSPARROW/issues) by linking code changes in response to bug reports or enhancement requests.
- **TN model**:Lastly, a working TN model would be greatly appreciated for Tampa Bay end users. I know this isn't really a comment about the code, nor is this entirely within the control of the package developers, but the *RSPARROW* model will have the greatest impact in our region if we can better understand TN loads and inputs. Our estuary is not phosphorus-limited, so the current tutorial models are of limited use for understanding nutrient impacts on water quality.