User guide to UHERO's forecast processes

Peter Fuleky

2024-12-13

Table of contents

1 About

This document describes some useful practices for using R for applied research, especially in the time series and forecasting domain. It also serves as a guide for contributors to the `forecastr` R project. The focus of the project is forecasting using multi-equation behavioral models. The project encompasses data preparation, model selection (work in progress), external forecast generation, local forecast generation (planned), simulations (planned), and forecast distribution to a more granular scale.

1.1 Contents

Chapters 2 - 4 discuss the general setup of a collaborative project under version control. Chapter 4 deals with the setup file that configures the most general aspects of the `forecastr` project. Chapter 5 describes user defined helper functions for the `forecastr` project. Chapter 6 gives examples of best practices for time series manipulation.

2 Project

Projects are useful for organizing work, especially when working on multiple pieces of research simultaneously. They help to keep the workspace clean and avoid conflicts between tasks. Projects also make it easier to share work with others. A project consists of the files associated with a given project (input data, R scripts, analytical results, and figures) kept together in a folder.

2.1 Project setup and conventions

Create a new project locally in RStudio under the File menu or using `usethis::create_project("proj_dir")`. The *.Rproj* file contains the project settings. Open the project by double clicking this file in Finder. The minimum structure of a project includes an *R* folder for scripts, a *data* folder for data, and an *output* folder for reports and plots. If present, the *data/raw* folder contains data external to the project and the *data/processed* folder contains intermediate processed data. Although local projects are sometimes useful to explore an idea, whenever you consider version tracking or collaboration, the project should be initiated from GitHub (see Chapter 3 for details).

Use the renv package to store information about the packages used in the project. The renv package facilitates sharing a project and maintaining the same behavior on different machines. It creates a local package directory for the project. This means that it keeps track of all the packages and package versions that are used in the project, and collaborators can restore the exact same package environment and reproduce the results (see Chapter 4 for details).

Use the here package to create paths relative to the project root. For example, `here::here("data", "raw/file.csv")` returns the path to the file *file.csv* in the *data/raw* folder. Load libraries and put hard coded lines at the top of the script. Use the conflicted package to detect conflicts across packages and assign preferences. For example, `conflict_prefer("filter", "dplyr")` assigns preference to the `filter` function in the dplyr package over the `filter` function in the stats package. Don't save the workspace on exit (Tools > Global Options > General > Save workspace to .RData on exit > Never or `usethis::use_blank_slate()`).

Start each pipe with a comment, and if necessary add comments to each line. Enable Github Copilot for RStudio; it is free for higher education users. Github Copilot will suggest code based on comments, which you can accept with the tab key. Use sectioning comments (#

comments followed by at least four dashes —-) to separate different parts of the script (they show up in the outline section of the editor pane). Use the addin provided by styler package to format the code. Follow the `tidyverse` "dialect" and syntax.

Use R scripts for coding; don't put the analysis into chunks in markdown documents. Only render important results in code chunks of quarto (*qmd*) or Rmarkdown (*Rmd*) documents. Within a *qmd* or *Rmd* document source the R script containing the analysis. Alternatively, save the entire workspace or individual objects from the R script, and then load these in the appropriate code chunks of the markdown document. Make sure the code chunks are looking for objects in the correct working directory.

Store secrets, passwords, and keys with the keyring package. For example, set the UDAMAN token with `keyring::key_set_with_value(service = "udaman_token", password = "-ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890=")` and retrieve it with `keyring::key_get("udaman_token")`. To avoid disclosing your secrets, do not store/assign the retrieved credentials to a variable. If security is not a concern, environment variables, such as API keys, can also be stored in a project specific *.Renviron* file; it must end with `\n`. In addition to *.Renviron*, the *.Rprofile* file is also executed each time R starts up. The latter typically contains a script with options and startup tasks. Both files are located in the project root folder.

2.2 Additional resources

Overview of R setup:
https://rstats.wtf

Best practices:
https://kdestasio.github.io/post/r_best_practices/

Considerations for structuring projects:
https://www.r-bloggers.com/2018/08/structuring-r-projects/

Set up your work in projects:
https://r4ds.hadley.nz/workflow-scripts#projects

Efficient data management in R:
https://www.r-bloggers.com/2020/02/efficient-data-management-in-r/

Efficient R programming:
https://csgillespie.github.io/efficientR/

Data science workflow:
http://dcl-workflow.stanford.edu

3 Version control

Version control tracks changes in files over time, allows you to revert files back to a previous state and compare changes over time. It is essential for the collaborative development of a project, but is also useful for individual projects.

3.1 Version control terms in Git(Hub)

- **.gitignore**: A file that specifies which files and directories should be excluded from version control.
- **Branch**: A separate line of development or parallel version of the repository. Branches are used to develop features or fix bugs without affecting the main branch.
- **Clone**: A copy of a repository on your local machine. Cloning a repository allows you to work on the project locally.
- **Commit**: A snapshot of the project at a specific point in time. Commits are used to save changes to the repository.
- **Conflict**: When two branches have made changes to the same line in a file, a conflict occurs. Conflicts need to be resolved before the changes can be merged.
- **Fork**: A copy of a repository on GitHub. This copy is independent of the original and you can make changes to it without affecting the original project.
- **HEAD**: A reference to the most recent commit in the repository, tip of the branch that is currently checked out.
- **Main or Master**: The default branch in a Git repository.
- **Merge**: Combining changes from one branch into another branch.
- **Origin**: The default name of primary version of the repository on GitHub.
- **Pull**: Getting changes from GitHub to your local repository.
- **Pull request**: A request to merge proposed changes from one branch into another branch. Pull requests are used to review and discuss changes before merging them into the main branch.

- **Push**: Sending changes from your local repository to GitHub.
- **Remote**: A remote repository on GitHub that you can pull from and push to.
- **Repository**: A folder that contains all the files and history of a project. There can be a local repository (on your computer) and a remote repository (hosted on GitHub).
- **Stage**: Staging allows you to select which changes should be included in the next commit.
- **Upstream**: The original repository that you forked from on GitHub.

3.2 Version control setup

RStudio has built in support for Git-based version control for a project. Check if Git is installed by running `git --version` in the terminal. Git is included in Xcode command line tools, which can be installed by running `xcode-select --install` in the terminal. Provide your GitHub user name and email to Git via `usethis::use_git_config(user.name = "Jane Doe", user.email = "jane@example.org")`. Next, generate and store a personal access token for GitHub via `usethis::create_github_token()` and `gitcreds::gitcreds_set()`, respectively. Finally, make sure version control is enabled in RStudio (RStudio > Global Options > Git/SVN > Enable version control interface for RStudio projects > check box or `usethis::use_blank_slate()`). This concludes the setup for Git and GitHub, making it possible to establish a connection between RStudio and GitHub.

Always start setting up version control on GitHub. Even if you already have a project on your computer, begin by setting up a repository on GitHub. This can be done by clicking on the "+" in the top right corner of the GitHub website and selecting "New repository". Give the repository a name and description, make it public or private, add a readme file, and choose the R template for .gitignore. After creating the repository, copy the URL and open RStudio. The next paragraph describes a robust local setup method with the `usethis` package, but there is also a menu based option: create a new project from version control by selecting File > New Project > Version Control > Git and pasting the URL in the "Repository URL" field. Use the repository name as the project directory name. Choose a folder on your computer where the project will be stored locally and click "Create". This will clone the repository to your computer and create a new RStudio project. The project is now connected to the repository on GitHub.

The local setup described above can be automated with `usethis::create_from_github("repo_url", "proj_dir")`. If you have permission to push to the remote GitHub repository because you are an owner or collaborator on the project, then `create_from_github` will clone it. If you do not have parmission to push to the repository, `create_from_github` will fork and clone it. In either case, do not work in the main branch. Instead, create a "dev" branch in RStudio and work in that branch: in the Git pane, click on the "New Branch", enter "dev" as the

branch name, keep the remote origin, and chek the sync with remote box. This will create a new branch and switch to it.

While in the dev branch, make changes to scripts or other files and save them. When you are ready to commit the changes, stage the files in the Git pane ("Command a" selects all files), click "Commit" at the top of the Git pane. In the pop-up window, the "Diff" button allows you to browse the changes made to the files in the repository, while the "History" button shows the commit history of the repository. These can be analyzed more conveniently on GitHub as described below in Section 3.3. Enter a commit message in the text box, and click the "Commit" button. Then push the changes to the remote repository. Go to the GitHub page of the dev branch and create a pull request to merge the dev branch into the main branch of the remote repository. After the pull request is merged, delete the dev branch on GitHub. Delete the local dev branch by executing `git branch -d dev` in the terminal. Finally, switch back to the main branch in RStudio and pull the changes from the remote repository.

3.3 Time travel on GitHub

From your repo's landing page, access commit history by clicking on "123 Commits" under the green Code button. Once you're viewing the history, notice three ways to access more info for each commit:

- The clipboard icon copies the SHA of the commit. This can be handy if you need to refer to this commit elsewhere, e.g. in an issue thread or a commit message or in a Git command you're forming for local execution.
- Click on the abbreviated SHA itself in order to the view the diff associated with the commit.
- Click on the double angle brackets <> to browse the state of the entire repo at that point in history.

Back out of any drilled down view by clicking on YOU/REPO to return to your repo's landing page. This brings you back to the present state and top-level of your repo.

Once you've identified a relevant commit, diff, or file state, you can copy the current URL from your browser and use it to enhance online discussion elsewhere, i.e. to bring other people to this exact view of the repo. The hyperlink-iness of repos hosted on GitHub can make online discussion of a project much more precise and efficient.

What if you're interested in how a specific file came to be the way it is? First navigate to the file in the repo, then notice "Blame" on the left and "History" in the upper right.

- The "blame" view of a file reveals who last touched each line of the file, how long ago, and the associated commit message. Click on the commit message to visit that commit. Or click the "stacked rectangles" icon to move further back in time, but staying in blame view. This is handy when doing forensics on a specific and small set of lines.
- The "history" view for a file is very much like the overall commit history described above, except it only includes commits that affect the file of interest. This can be handy when your inquiry is rather diffuse and you're trying to digest the general story arc for a file.

When viewing a file on GitHub, you can click on a line number to highlight it. Use "click … shift-click" to select a range of lines. Notice your browser's URL shows something of this form: https://github.com/OWNER/REPO/blob/SHA/path/to/file.R#L27-L31 If the URL does not contain the SHA, type "y" to toggle into that form. These file- and SHA-specific URLs are a great way to point people at particular lines of code in online conversations. It's best practice to use the uglier links that contain the SHA, as they will stand the test of time.

Search is always available in the upper-righthand corner of GitHub. Once you enter some text in the search box, a dropdown provides the choice to search in the current repo (the default) or all of GitHub. GitHub searches the contents of files (described as "Code"), commit messages, and issues. Take advantage of the search hits across these different domains. Again, this is a powerful way to zoom in on specific lines of code, revisit an interesting time in project history, or re-discover a conversation thread.

3.4 Version control workflow

After the initial setup, the workflow should always follow the following sequence:

1. in the local/main branch, click on the "Pull" button in the Git pane to pull changes from the main branch of the remote repository, which is

   - upstream if the repository was forked,
   - origin if the repository was cloned,

   in the case of forked repo, after pulling from upstream, push to origin,
2. create a new dev branch and switch to it,
3. make changes in the dev branch,
4. commit changes in the dev branch,
5. push changes to the dev branch of the remote origin repository,
6. create a pull request on GitHub to merge the origin dev branch a) into upstream main if the repository was forked, b) into origin main if the repository was cloned,

7. merge the pull request (or wait for it to be merged by the owner),
8. after merging, delete the dev branch on the remote and locally,
9. repeat step 1. (and then repeat it again before you resume your work on the project).

This workflow is recommended to avoid conflicts with other collaborators.

3.5 Dealing with conflicts

If a push is rejected, pull the changes from the remote repository. If there are conflicts, resolve them by editing the files and committing the changes. Every merge conflict inserts three delimiters:

`<<<<<<` feature branch name, the start of the merge conflict
`======` the separator between the content of both branches
`>>>>>>` base branch name, the end of the merge conflict

Fix the merge conflict by directly editing the script at the indicated locations. Often you can fix it by simply deleting the content of one of the branches within the conflict. Potentially you need to keep a mix of both. Don't forget to also delete the three delimiters when you're ready.

3.6 Additional resources

Visual Git guide:
https://inbo.github.io/git-course/index.html

Exhaustive discussion of Git for R users:
https://happygitwithr.com

A research workflow based on Github:
https://www.carlboettiger.info/2012/05/06/research-workflow.html

For more advanced tasks, use GitHub Desktop:
https://desktop.github.com

4 Package management

The renv package helps create **r**eproducible **env**ironments for your R projects. Use renv to make R projects more isolated, portable and reproducible.

- **Isolated**: Installing a new or updated package for one project won't break other projects, and vice versa. That's because renv gives each project its own private library.
- **Portable**: Easily transport projects from one computer to another, even across different platforms. renv makes it easy to install the packages the project depends on.
- **Reproducible**: renv records the exact package versions the project depends on, and ensures those exact versions get installed by others who work on the project.

4.1 Getting started

To convert a project to use renv, call `renv::init()`. This adds three new files and directories to the project:

- The project library, *renv/library*, is a library that contains all packages currently used by the project[1]. This is the key magic that makes renv work: instead of having one library containing the packages used in every project, renv gives you a separate library for each project. This provides the benefits of isolation: different projects can use different versions of packages, and installing, updating, or removing packages in one project doesn't affect any other project.
- The lockfile, *renv.lock*, records enough metadata about every package that it can be re-installed on a new machine.
- renv uses *.Rprofile* to configure the R session to use the project library. This ensures that once renv is turned on for a project, it stays on, until it is deliberately turned off.

The next important pair of tools is `renv::snapshot()` and `renv::restore()`. `snapshot()` updates the lockfile with metadata about the currently-used packages in the project library. Sharing the lockfile allows other people or other computers to reproduce the current project environment by running `restore()`, which uses the metadata from the lockfile to install exactly

[1] If you'd like to skip dependency discovery, you can call `renv::init(bare = TRUE)` to initialize a project with an empty project library.

the same version of every package. This pair of functions provides the benefits of reproducibility and portability: you are now tracking exactly which package versions you have installed so you can recreate them on other machines.

4.2 Collaboration

One of the reasons to use renv is to make it easier to share code in such a way that everyone gets exactly the same package versions. As above, start by calling `renv::init()`. You'll then need to commit *renv.lock*, *.Rprofile*, *renv/settings.json* and *renv/activate.R* to version control, ensuring that others can recreate your project environment. If you're using git, this is particularly simple because renv will create a *.gitignore*, and you can just commit all suggested files.

Now when one of your collaborators opens this project, renv will automatically bootstrap itself, downloading and installing the appropriate version of renv. It will also ask them if they want to download and install all the packages it needs by running `renv::restore()`.

4.3 Installing packages

If you use renv for multiple projects, you'll have multiple libraries, meaning that you'll often need to install the same package in multiple places. It would be annoying if you had to download (or worse, compile) the package repeatedly, so renv uses a package cache. That means you only ever have to download and install a package once, and for each subsequent install, renv will just add a link from the project library to the global cache. You can learn more about the cache in `vignette("package-install")`.

After installing the package and checking that the code works, you should call `renv::snapshot()` to record the latest package versions in your lockfile. If you're collaborating with others, you'll need to commit those changes to git, and let them know that you've updated the lockfile and they should call `renv::restore()` when they're next working on a project.

4.4 Updating packages

Regularly (at least once a year) update the packages in your project to get the latest versions of all dependencies. Similarly, if you're making major changes to a project that you haven't worked on for a while, it's often a good idea to start with an `renv::update()`[2] before making any changes to the code.

[2] You can also use `update.packages()`, but `renv::update()` works with the same sources that `renv::install()` supports.

After calling `renv::update()`, you should run the code in your project and verify that it still works (or make any changes needed to get it working). Then call `renv::snapshot()` to record the new versions in the lockfile. If you get stuck, and can't get the project to work with the new versions, you can call `renv::restore()` to roll back changes to the project library and revert to the known good state recorded in your lockfile. If you need to roll back to an even older version, take a look at `renv::history()` and `renv::revert()`. `renv::update()` will also update renv itself, ensuring that you get all the latest features.

4.5 Workflow for setting up a project with version control and renv

The ideal sequence of steps to set up a project with version control and renv is as follows:

1. github
2. usethis::create_from_github()
3. renv::init() or renv::restore() git::pull …
4. get work done
5. renv::snapshot()
6. git

4.6 Additional resources

Overview of `renv`:
https://rstudio.github.io/renv/articles/renv.html

Leveraging git and renv at the end of a working session

Now that we have our nice project setup, we should not forget to leverage it. At the end of a working session you should follow the following steps:

Create a renv::snapshot() to save all the packages you used to your local package directory. Commit all your changes to git. This can be easily done by using the Git pane in RStudio. Push everything to GitHub. Whenever you re-open your Rproject, make sure to start your working session with a Pull from GitHub. That way, you will always work with the most recent version of your project.

Collaborating with `renv`: https://rstudio.github.io/renv/articles/collaborating.html

TL;DR: https://inbo.github.io/git-course/index.html https://happygitwithr.com/

5 Setup of the forecastr project

The `setup.R` file contains general information used throughout the project. The contents are listed below.

5.1 Start with a clean slate

First remove all objects from global environment:
`rm(list = ls())`

If only some objects need to be removed, search for them via wildcards:
`rm(list = ls(pattern = glob2rx("*__*")))`

Detach all loaded packages:

```
if (!is.null(names(sessionInfo()$otherPkgs))) {
invisible(
  suppressMessages(
    suppressWarnings(
      lapply(
        paste("package:", names(sessionInfo()$otherPkgs), sep=""),
        detach,
        character.only = TRUE,
        unload = TRUE
        )
      )
    )
  )
}
```

5.2 Packages

The setup file clarifies its own location relative to the project root and loads the necessary packages.

Navigate within a project using the `here()` package. Start by specifying:
`here::i_am("R/setup.R")`

Then load necessary packages

```
library(here) # navigation within the project
library(conflicted) # detect conflicts across packages
library(tidyverse) # a set of frequently used data-wrangling tools
library(magrittr) # more than just pipes
library(lubridate) # dealing with dates
library(tsbox) # dealing with time series
# library(bimets)
```

Detect conflicts across packages and assign preferences

```
conflict_scout()
conflict_prefer("filter", "dplyr") # dplyr v stats
conflict_prefer("first", "dplyr") # dplyr v xts
conflict_prefer("lag", "dplyr") # dplyr v stats
conflict_prefer("last", "dplyr") # dplyr v xts
conflict_prefer("extract", "magrittr") # magrittr vs tidyr
```

Verify top level project directory
`here()`

5.3 Package descriptions

Use the `here` package to deal with file paths:
https://here.r-lib.org

Suppose you have a dataset in csv format. Use:
`readr::read_csv(here::here("<The subfolder where your csv file resides>", "<The CSV file.csv>"))`

Only load essential packages with many useful functions (don't load a whole package to access a single function).
Refer to individual functions in not loaded packages by `namespace::function()`

Resolve conflicts across multiple packages with `conflicted`:
https://conflicted.r-lib.org

Core `tidyverse` packages:
https://www.tidyverse.org

Non-core `tidyverse` packages (need to be loaded separately):
https://magrittr.tidyverse.org
https://lubridate.tidyverse.org

Time series tools in `tsbox` (learn them and use them, very useful). All start with `ts_`.
https://www.tsbox.help

Forecasting with multi-equation behavioral models (only load `bimets` if actually doing forecasts, no need for data manipulation):
https://cran.r-project.org/web/packages/bimets/index.html

`bimets` depends on `xts` (if not loaded, can access necessary functions via `xts::function()`):
https://cran.r-project.org/web/packages/xts/index.html

Prefer using `tsbox` and `tidyverse` functions whenever possible, but understand the components and behavior of `xts` objects: https://rc2e.com/timeseriesanalysis

5.4 Additional info in setup

Define project-wide constants:

```
bnk_start <- ymd("1900-01-01")
bnk_end <- ymd("2060-12-31")
```

Load user defined utility functions (details in next section):
`source(here("R", "util_funs.R"))`

6 User defined utility functions

Functions not available in existing packages are stored in util_funs.R. A pdf version of this document is available here.

6.1 AtoQ

Description:

```
Linear interpolation based on aremos command reference page 292
```

Usage:

```
AtoQ(ser_in, aggr = "mean")
```

Arguments:

```
ser_in: the xts series to be interpolated (freq = a)

aggr: interpolation method: aggregate via mean (default) or sum
```

Value:

```
interpolated xts series (freq = q)
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
```

6.2 explode_xts

Description:

```
Splitting of xts matrix to individual xts vectors (don't use,
pollutes global environment)
```

Usage:

```
explode_xts(xts_in)
```

Arguments:

```
xts_in: the xts matrix to be split into individual xts vectors
```

Value:

```
nothing (silently store split series in global environment)
```

Examples:

```
get_series_exp(74, save_loc = NULL) %>%
  ts_long() %>%
  ts_xts() %>%
  explode_xts()
rm(list = ls(pattern = glob2rx("*@HI.Q")))
```

6.3 find_end

Description:

```
Find the date of the last observation (NAs are dropped)
```

Usage:

```
find_end(ser_in)
```

Arguments:

```
ser_in: an xts series
```

Value:

```
date associated with last observation
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2060, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2018"] <- c(323127513, 325511184, 327891911)
find_end(`ncen@us.sola`)
```

6.4 find_start

Description:

```
Find the date of the first observation (NAs are dropped)
```

Usage:

```
find_start(ser_in)
```

Arguments:

```
ser_in: an xts series
```

Value:

```
date associated with first observation
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2017/2021"] <- c(325511184, 327891911, 330268840, 332639102, 334998398)
find_start(`ncen@us.sola`)
```

6.5 get_series

Description:

```
Download a set of series from udaman using series names
```

Usage:

```
get_series(ser_id_vec)
```

Arguments:

```
ser_id_vec: vector of series names
```

Value:

```
time and data for all series combined in a tibble
```

Examples:

```
get_series(c("VISNS@HI.M", "VAPNS@HI.M"))
```

6.6 get_series1

Description:

```
Download a single series from udaman using series name
```

Usage:

```
get_series1(ser_id)
```

Arguments:

```
ser_id: udaman series name
```

Value:

```
time and data for a single series combined in a tibble
```

Examples:

```
get_series("VISNS@HI.M")
```

6.7 get_series_exp

Description:

```
Download series listed in an export table from udaman
```

Usage:

```
get_series_exp(exp_id, save_loc = "data/raw")
```

Arguments:

```
exp_id: export id

save_loc: location to save the csv of the retrieved data, set to NULL
     to avoid saving
```

Value:

```
time and data for all series combined in a tibble
```

Examples:

```
get_series_exp(74)
get_series_exp(74, save_loc = NULL)
```

6.8 get_var

Description:

```
Construct a series name from variable components and retrieve the
series
```

Usage:

```
get_var(ser_in, env = parent.frame())
```

Arguments:

```
ser_in: a variable name (string with substituted expressions)

env: environment where the expression should be evaluated
```

Value:

```
variable
```

Examples:

```
ser_i <- "_NF"
cnty_i <- "HI"
get_series_exp(74, save_loc = NULL) %>%
  ts_long() %>%
  ts_xts() %$% get_var("E{ser_i}@{cnty_i}.Q")
```

6.9 make_xts

Description:

```
Create xts and fill with values
```

Usage:

```
make_xts(start = bnk_start, end = bnk_end, per = "year", val = NA)
```

Arguments:

```
start: date of series start (string: "yyyy-mm-dd")

end: date of series end (string: "yyyy-mm-dd")

per: periodicity of series (string: "quarter", "year")

val: values to fill in (scalar or vector)
```

Value:

```
an xts series
```

Examples:

```
make_xts()
make_xts(start = ymd("2010-01-01"), per = "quarter", val = 0)
```

6.10 p

Description:

```
Concatenate dates to obtain period
```

Usage:

```
p(dat1, dat2)
```

Arguments:

```
dat1: date of period start (string: yyyy-mm-dd)

dat2: date of period end (string: yyyy-mm-dd)
```

Value:

```
string containing date range
```

Examples:

```
p("2010-01-01", "2020-01-01")
```

6.11 pca_to_pc

Description:

```
Convert annualized growth to quarterly growth
```

Usage:

```
pca_to_pc(ser_in)
```

Arguments:

```
ser_in: the series containing annualized growth (in percent)
```

Value:

```
series containing quarterly growth (in percent)
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
ts_c(test1 %>% ts_pca() %>% pca_to_pc(), test1 %>% ts_pc())
```

6.12 pchmy

Description:

```
Calculate multi-period average growth
```

Usage:

```
pchmy(ser_in, lag_in = 1)
```

Arguments:

```
ser_in: name of xts series for which growth is calculated

lag_in: length of period over which growth is calculated
```

Value:

```
series containing the average growth of ser_in (in percent)
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
ts_c(pchmy(`ncen@us.sola`, lag_in = 3), ts_pc(`ncen@us.sola`))
ts_c(pchmy(test1, lag_in = 4), ts_pcy(test1), ts_pca(test1), ts_pc(test1))
```

6.13 plot_1

Description:

```
Interactive plot of a single variable with level and growth rate
```

Usage:

```
plot_1(
  ser,
  rng_start = as.character(Sys.Date() - years(15)),
  height = 300,
  width = 900
)
```

Arguments:

```
ser: time series to plot

rng_start: start of zoom range ("YYYY-MM-DD")

height: height of a single panel (px)

width: width of a single panel (px)
```

Value:

```
a dygraph plot
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
plot_1(`ncen@us.sola`, rng_start = "2017-01-01")
plot_1(test1, rng_start = "2017-01-01")
```

6.14 plot_comp

Description:

```
Three-panel plot of levels, index, and growth rates
```

Usage:

```
plot_comp(
  sers,
  rng_start = as.character(Sys.Date() - years(15)),
  rng_end = as.character(Sys.Date()),
  height = 300,
  width = 900
)
```

Arguments:

```
sers: a vector of series to plot

rng_start: start of the zoom range ("YYYY-MM-DD")

rng_end: end of the zoom range ("YYYY-MM-DD")

height: height of a single panel (px)

width: width of a single panel (px)
```

Value:

```
a list with three dygraph plots (level, index, growth)
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
plot_comp(ts_c(`ncen@us.sola`, test1), rng_start = "2017-01-01")
get_series_exp(74, save_loc = NULL) %>%
  ts_long() %>%
  ts_xts() %>%
  extract(, c("E_NF@HI.Q", "ECT@HI.Q", "EMN@HI.Q")) %>%
  plot_comp()
```

6.15 plot_comp_2

Description:

```
Two-panel plot of levels, index, and growth rates
```

Usage:

```
plot_comp_2(
  sers,
  rng_start = as.character(Sys.Date() - years(15)),
  rng_end = as.character(Sys.Date()),
  height = 300,
  width = 900
)
```

Arguments:

```
sers: a vector of series to plot

rng_start: start of the zoom range ("YYYY-MM-DD")

rng_end: end of the zoom range ("YYYY-MM-DD")

height: height of a single panel (px)

width: width of a single panel (px)
```

Value:

```
 a list with two dygraph plots (level, index, growth)
```

Examples:

```
 `ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
   ts_xts()
 `ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
 test1 <- AtoQ(`ncen@us.sola`)
 plot_comp_2(ts_c(`ncen@us.sola`, test1), rng_start = "2017-01-01")
 get_series_exp(74, save_loc = NULL) %>%
   ts_long() %>%
   ts_xts() %>%
   extract(, c("E_NF@HI.Q", "ECT@HI.Q", "EMN@HI.Q")) %>%
   plot_comp_2()
```

6.16 QtoA

Description:

```
 Conversion from quarterly to annual frequency
```

Usage:

```
 QtoA(ser_in, aggr = "mean")
```

Arguments:

```
ser_in: the xts series to be converted (freq = q)

aggr: aggregate via mean (default) or sum
```

Value:

```
converted xts series (freq = a)
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
test2 <- QtoA(test1) # for stock type variables mean, for flow type variables sum
print(test1)
print(cbind(`ncen@us.sola`, test2))
```

6.17 QtoM

Description:

```
Interpolate a tibble of series from quaterly to monthly freq
```

Usage:

```
QtoM(data_q, conv_type)
```

Arguments:

```
data_q: tibble containing variables at quarterly freq

conv_type: match the quarterly value via "first", "last", "sum",
     "average"
```

Value:

```
tibble containing variables at monthly freq
```

Examples:

```
`ncen@us.sola` <- ts(NA, start = 2016, end = 2021, freq = 1) %>%
  ts_xts()
`ncen@us.sola`["2016/2021"] <- c(323127513, 325511184, 327891911, 330268840, 332639102, 334
test1 <- AtoQ(`ncen@us.sola`)
QtoM(ts_tbl(test1), "average")
ts_frequency(QtoM(ts_tbl(test1), "average") %>% ts_xts())
```

6.18 QtoM1

Description:

```
Interpolate a single series from quarterly to monthly freq
```

Usage:

```
QtoM1(var_q, ts_start, conv_type)
```

Arguments:

```
var_q: vector containing a single variable at quarterly freq

ts_start: starting period as c(year, quarter) e.g. c(2001, 1)

conv_type: match the quarterly value via "first", "last", "sum",
     "average"
```

Value:

```
vector containing a single variable at monthly freq
```

Examples:

```
QtoM1(test1, c(2010, 1), "average")
```

6.19 qtrs

Description:

```
Convert period in quarters to period months
```

Usage:

```
qtrs(nr_quarters)
```

Arguments:

```
nr_quarters: number of quarters in period (integer)
```

Value:

```
number of months in period
```

Examples:

```
qtrs(3)
ymd("2020-01-01") + qtrs(3)
```

7 Best practices for time series data manipulation

Use capital letters for series names. Special characters in variable names require putting the name between backticks (e.g. `N@US.A`). Eliminate special characters using a long tibble.

```
hist_q_mod <- hist_q %>%
  ts_tbl() %>%
  mutate(id = str_replace_all(id, c("@" = "_AT_", "\\." = "_")))

# revert back to udaman notation
hist_q <- hist_q_mod %>%
  ts_tbl() %>%
  mutate(id = str_replace_all(id, c("_AT_" = "@", "_Q" = "\\.Q")))
```

Use the `xts` format whenever possible. Observations in a multivariate `xts` can be accessed by time and series name in two ways: `mul_var_xts[time, ser_name]` or `mul_var_xts$ser_name[time]`.

Make sure all series are defined on the same range (default start = bnk_start, end = bnk_end). Take advantage of `make_xts()` (and its defaults, e.g. start and end period).

```
import_xts <- read_csv(here("data/raw", str_glue("{exp_id_a}.csv"))) %>%
  arrange(time) %>%
  ts_long() %>%
  ts_xts() %>%
  ts_c(
    temp = make_xts(per = "year") # temporary variable to force start and end in import_xts
  ) %>%
  extract(, str_subset(colnames(.), "temp", negate = TRUE)) # remove temp
```

Don't break up multivariate time series (think databank) into individual series in the global environment.

If referring directly to a series with a static name, use the `bank$series` notation (this can be used on both the right and the left hand side of the assignment, while `bank[, series]` can only be used for existing series in bank).

```
# find the last value in history
dat_end <- find_end(hist_q$N_AT_US_Q)
# same as
dat_end <- find_end(hist_q[, N_AT_US_Q])
```

Use `[p()]` to select a period in `xts` objects, otherwise use `ts_span()`.

```
# extend series with addfactored level
sol_q$N_AT_US_SOLQ <- hist_q$N_AT_US_Q[p("", dat_end)] %>%
  ts_bind(sol_q$NCEN_AT_US_SOLQ[p(dat_end, "")] +
    as.numeric(sol_q$N_AT_US_SOLQ_ADDLEV[dat_end]))

# addfactor for growth
sol_q$N_AT_US_SOLQ_ADDGRO[p(dat_end + qtrs(1), dat_end + qtrs(4))] <- -0.35

# extend history using growth rate
sol_q$N_AT_US_SOLQ <- sol_q$N_AT_US_SOLQ[p("", dat_end)] %>%
  ts_chain(ts_compound(sol_q$N_AT_US_SOLQ_GRO[p(dat_end, "")]))
```

The `bank[,seriesname]` notation only works for *existing* `xts` series on the left of the assignment (it can also be used on the right). `seriesname` can be determined at runtime

```
# initialize the lhs series in the "bank"
hist_a$temp <- make_xts()
names(hist_a)[names(hist_a) == "temp"] <- str_glue("E{ser_i}_AT_{cnty_i}_ADD")

# calculate expression and assign to lhs
hist_a[, str_glue("E{ser_i}_AT_{cnty_i}_SH")] <-
  (hist_a[, str_glue("E{ser_i}_AT_{cnty_i}")] / hist_a[, str_glue("E_NF_AT_{cnty_i}")])
```

Alternatively, make multiple series in *bank* available by `%$%` and retrieve inividual series by `get_var()` on the right.

```
hist_a[, str_glue("E{ser_i}_AT_{cnty_i}_SH")] <- hist_a %$%
  (get_var("E{ser_i}_AT_{cnty_i}") / get_var("E_NF_AT_{cnty_i}"))
```

Bimets requires data in a particular `tslist` format. Convert `xts` to `tslist` using `ts_tslist()`.

```
# store series as tslist
hist_a_lst <- hist_a %>%
  ts_tslist() %>%

# convert series to bimets format
hist_a_bimets <- hist_a_lst %>%
  map(as.bimets)

# bimets strips the attributes, need to reset them for further manipulation by tsbox
hist_a <- hist_a_bimets %>%
  set_attr("class", c("tslist", "list")) %>%
  ts_xts()
```

For series collected in a `tslist` on the left of the assignment use the `bank[[seriesname]]` notation (it can also be used on the right). Here the lhs series `seriesname` does not need to exist, and it might easier to work with `tslist` than `xts` when variable names are determined at runtime.

```
# similar to above with a tslist variable
hist_a_lst[[str_glue("E{ser_i}_AT_{cnty_i}_ADD")]] <- hist_a_lst %$%
  (get_var("E{ser_i}_AT_{cnty_i}") - get_var("E_NF_AT_{cnty_i}"))
```

7.1 Harness the power of tsbox

Use the converter functions in tsbox to shift between various data types (`ts_tbl()`, `ts_xts()`, `ts_ts()`, `ts_tslist()`) and reshaping to the long and wide format (`ts_long()`, `ts_wide()`). `tsbox` further contains funtions for time period selection (`ts_span()`), merging and extension operations (`ts_c()`, `ts_bind()`, `ts_chain()`), transformations (`ts_lag()`, `ts_pc()`, `ts_pca()`, `ts_pcy()`, `ts_diff()`, `ts_diffy()`), and index construction (`ts_compound()`, `ts_index()`). Consider these before turning to solutions that are specific to the `xts`, `ts`, `dplyr` or `tidyr` packages.

8 Model selection

The model selection process can be run line-by-line from an R script directly (R/gets_model_select.R) or via sourcing an Rmd document (notes/gets_model_select.Rmd) which collects all model selection results in an easier to digest html file. Running the full script (source) takes about 1 minute.

8.1 Main user settings

- Start and end of period used for model selection.
- End of period used for estimation (selected model can be re-estimated for different sample).
- Start and end of quasi-forecast period (for model evaluation).
- Maximum number of lags considered in models.
- Response variable.
- List of predictors.

8.2 Data preparation (`tidyverse`)

- Download all series used in the model selection process from UDAMAN (about 500 rows and 1200 columns) and eliminate special characters from the series names.
- Log-transform all variables.
- Load (create) all indicators (dummies for impulse, level shift, seasonal) and trend.
- Combine all variables into a single dataset.

- Set date range for model selection.
- Generate 8 lags of predictors.
- Filter data set down to specific variables considered in a particular model, including trend and season dummies.

8.3 Model selection steps (`gets`)

https://cran.r-project.org/web/packages/gets/index.html
- Formulate a general unrestricted model.
- Run the gets (general to specific) model selection algorithm.
- Identify outliers in the relationship.
- Repeat gets model selection over specific model and outliers.
- Verify that no additional outliers arise due to greater model parsimony.
- If estimation period is shorter than model selection period, remove predictors containing zeros only (e.g. outlier past the end of estimation period).
- Re-estimate final model.
- Save model equation as a txt file (not plugging in estimated coefficients here to keep it general). If happy with the model, copy this equation into file containing all model equations.

8.4 Produce a quasi-forecast with the selected model (`bimets`)

https://cran.r-project.org/web/packages/bimets/vignettes/bimets.pdf
- Load model from txt file.
- Load data used by the model.
- Estimate the model (if estimation period ends before the last data point also run a Chow test of model stability).
- Simulate model.
- Evaluate simulation by plotting quasi-forecast and actual history.

9 Stochastic simulations

The model selection process stores a set of general equations (coefficient estimates are not plugged in) in a text file. Before simulation can commence, several steps need to take place: compile the system of equations, add data to the equations, estimate equations. `bimets` does not automatically adjust the sample for missing data points, so need to identify the time period with a rectangular sample for the estimation of each equation. For forecasting, deal with the ragged edge of the data by finding the last data point for each series and "exogenize" the series up to that point (use actuals in simulation).

9.1 Main user settings

- Start of forecast period.
- End of forecast period.
- End of estimation period.
- Maximum number of lags in models.

9.2 Data preparation

- Download all series used in the model selection process from UDAMAN (about 500 rows and 1200 columns) and eliminate special characters from the series names.
- Load (create) all indicators (dummies for impulse, level shift, seasonal) and trend.
- Combine all variables into a single dataset.

9.3 Simulation prep

- Compile model (load equations from text file and let `bimets` digest the info).
- Add variables to model.

- Set date range for estimation (`bimets` does not automatically drop periods with NA's).
- Set exogenization range to deal with ragged edge in simulation.

- Estimate model equations and save estimation results to text file for inspection.
- Set add factors.

9.4 Simulation

- Simulate model deterministically to obtain mean forecast.
- Extract forecast and combine it with history.
- Inspect the forecast via plots.
- Set parameters for stochastic simulations.
- Run stochastic simulation.
- Extract simulated paths and obtain deviations from the mean forecast.
- Inspect the paths via plots.

10 Notes

10.1 Project setup

Coding Conventions in R:

Basic ideas for a reproducible workflow:

Use RStudio projects with sub-directories

- R - R code.
- data/raw - data external to the project.
- data/processed - intermediate processed data.
- notes - Rmd, and Rmd output, notes, papers, supporting documents, Rmd, etc.
- output - reports, tables, etc.
- output/plots - plots.
- renv - used for library management (don't edit).
- man - help files (don't edit).

Preference settings:

10.2 Git

A quick overview: https://github.com/llendway/github_for_collaboration/blob/master/github_for_collaborat

10.3 Git step by step

If you don't have Git, install it:
https://happygitwithr.com/install-git.html

Make sure .gitignore contains the following files:
.Renviron .Rprofile

Introduce yourself to Git:
In the shell (Terminal tab in RStudio):
git config –global user.name 'Jane Doe'

git config –global user.email 'jane@example.com'
git config –global –list

For more advanced tasks, use GitHub Desktop:
https://desktop.github.com

Store your GitHub PAT (Personal Access Token):
https://happygitwithr.com/https-pat.html

Use one of three ways to add your project to GitHub:
Brand new project:
https://happygitwithr.com/new-github-first.html
Existing project without version control:
https://happygitwithr.com/existing-github-first.html
Existing project under local version control:
https://happygitwithr.com/existing-github-last.html

Troubleshooting if RStudio can't detect Git:
https://happygitwithr.com/rstudio-see-git.html

Git vocabulary:
https://happygitwithr.com/git-basics.html

Remote setups (try to stick to GitHub first discussed above):
https://happygitwithr.com/common-remote-setups.html

Useful Git workflows and dealing with conflicts:
https://happygitwithr.com/workflows-intro.html
https://happygitwithr.com/push-rejected.html
https://happygitwithr.com/pull-tricky.html

Additional resources:
https://happygitwithr.com/ideas-for-content.html

Suggested workflow:
1) Initialize repository on GitHub.com under the UHERO account.
2) Clone it via RStudio project setup.
3) Commit changes, pull, resolve issues, push. 3*) If work in a branch (create in RStudio), commit to branch, (pull) push to remote, pull request on GitHub.com from branch to main, merge, delete branch on GitHub.com.

Render results from R scripts via Rmd: 1) source your R code from within Rmd 2) only render important results in Rmd chunks

Use here() from the here package to write file paths

Suppose you have a dataset in csv format. Use readr::read_csv(here::here("The subfolder where your csv file resides", "The CSV file.csv"))

Do not use setwd() and rm(list = ls())

Do not save the workspace to the .Rdta file

Use library() not require()

Use version control (useful for recording changes between different versions of a file over time - see below for Git integration)

See the resources below:

Best Practices & Style Guide for Writing R Code: https://github.com/kmishra9/Best-Practices-for-Writing-R-Code

R Code – Best practices: https://www.r-bloggers.com/2018/09/r-code-best-practices/

R Best Practices by Krista L. DeStasio: https://kdestasio.github.io/post/r_best_practices/

Project-oriented workflow: https://www.tidyverse.org/blog/2017/12/workflow-vs-script/

R coding style best practices: https://www.datanovia.com/en/blog/r-coding-style-best-practices/

What They Forgot to Teach You About R by Jennifer Bryan and Jim Hester: https://rstats.wtf/save-source.html

Conflicted: a new approach to resolving ambiguity: https://www.tidyverse.org/blog/2018/06/conflicted/

Introduction to renv package: https://rstudio.github.io/renv/articles/renv.html#future-work-1

Row-oriented workflows in R with the tidyverse: https://github.com/jennybc/row-oriented-workflows#readme

Structuring R projects: https://www.r-bloggers.com/2018/08/structuring-r-projects/

Defensive Programming in R: https://bitsandbugs.io/2018/07/27/defensive-programming-in-r/#8

Nice R code: https://nicercode.github.io/blog/2013-04-05-projects/

Workflow basics: https://r4ds.had.co.nz/workflow-basics.html

Namespace package: https://r-pkgs.org/namespace.html

Writing R packages in RStudio: https://ourcodingclub.github.io/tutorials/writing-r-package/

It is dangerous to change state: https://withr.r-lib.org/articles/changing-and-restoring-state.html

The targets R Package User Manual: https://books.ropensci.org/targets/

Github and R:

Install git on the R system from here: https://git-scm.com/downloads

Go to RStudio → Global Options → Git/SVN → Make sure the box "Enable version control interface for RStudio projects" is checked

Tell RStudio where your Git executable is in the Git/SVN under Global Options

Create a new project in R (make sure the check box "Create a git repository" is checked)

Create a new task file in R (New File → Rscript) and save it as a .R file

To use Git version control on the .R file we need to commit that file

To commit a file with Git in RStudio go to the Git tab in the top right pane in R → Select one or more files by checking the box

Checking the box means that it is ready to be committed

To actually commit the file click the "Commit" button (will open up a commit window)

Include a commit message then click on the second "Commit" button

For collaboration on Github:

Load the usethis package and type in ?use_github in the R console

In the Authentication section, click on GitHub personal access token (PAT)

Click on the button to generate a new token

Put a Note and use repo permission for your token and then click on "Generate token"

Copy the token ID number (needs to be stored)

Type in edit_r_environ() in the R console and then type in GITHUB_PAT ='copy and paste token ID number here'

In R console type in use_github(protocol ='https', auth_token = Sys.getenv("GITHUB_PAT"))

Run it and will ask if you are sure. Select 3

This will create a Github repository and will set up the syncing

Another way to collaborate on Github (easier so follow this!):

Go to http://github.com and create an account

Create a new repository and give it a name (click "Add a README file)

Go to R → Install the usethis package and include library(usethis) → Type in use_git_config(user.name = "Your Name on the GitHub account", user.email = "Your email address on the GitHub account")

In the newly created repository, click the "Code" button on GitHub. Copy the URL under the "Clone with HTTPS"

Go to R → New Project → Version Control → Git → Repository URL (copy and paste the HTTP URL from your Github repository) - this will connect what's on the cloud on Github to your computer (also called cloning your repository)

Can start a new R script and would be able to see the Git tab in R

Can commit and include a commit message (will add the files to your depository)

Need to push to fully make the changes go through and to show up on your GitHub account

Under the History tab you would be able to see the changes you made and committed

Can link the SSH keys from settings on your account and into R under the Git/SVN tab (have to create a SSH RSA key if it has not been created already)

If there is a merge conflict when collaborating on making simultaneous changes together then pull first and then fix the merge conflict. Then can commit by finalizing on which changes to keep by eliminating the "====" and "»»" and push it out. The other person will have to pull in the changes in her hand.

Creating a new branch will allow you to do things on your own. Click on the branch button to create a new branch and name it. A new branch will allow you to make changes on it and work separately on it. The other person will have to pull to see the new branch and your changes on it. In this way, we can work independently when working together at the same time. Then will have to merge the independent branches.

Open a pull request by clicking on the Compare and pull request button on the Github site to merge the separate branches together. Can delete your separate branch if desired. Then go to R and pull the changes down.

For .Renviron have to use specific user credentials such as user name, password, Github and udaman tokens.

The .Rprofile can be ignored in gitignore if there is a problem with different paths across Macs and PCs.

Resources:

Happy Git and GitHub for the useR: https://happygitwithr.com/

Github for collaboration: https://github.com/llendway/github_for_collaboration/blob/master/github_for_col

My research workflow, based on Github: https://www.carlboettiger.info/2012/05/06/research-workflow.html

Collaborating with renv: https://rstudio.github.io/renv/articles/collaborating.html

R style guide: http://adv-r.had.co.nz/Style.html

UHERO R style guide:

Use block letters for R file names (because the NAS file server is case sensitive)

Comment your code

Time Series Modeling:

Forecasting: Principles and Practice (3rd ed) by Rob J Hyndman and George Athanasopoulos : https://otexts.com/fpp3/index.html

An Introduction to Statistical Learning (1st ed): https://www.statlearning.com

Manipulating Time Series Data in R with xts & zoo: https://rstudio-pubs-static.s3.amazonaws.com/288218_117 https://rpubs.com/mpfoley73/504487 Time Series in R, The Power of xts and zoo: https://ugoproto.github.io/ugo_r_doc/time_series_in_r_the_power_of_xts_and_zoo/ xts Cheat Sheet: Time Series in R: https://www.r-bloggers.com/2017/05/xts-cheat-sheet-time-series-in-r/

R For Data Science Cheat Sheet by DataCamp: https://s3.amazonaws.com/assets.datacamp.com/blog_assets/x

Evaluate the R packages: gets, ARDL, etc.

The gets package is used for Multi-path General-to-Specific (GETS) modelling of the mean and/or variance of a regression, and Indicator Saturation (ISAT) methods for detecting structural breaks in the mean.https://cran.r-project.org/web/packages/gets/index.html

The ARDL package creates complex autoregressive distributed lag (ARDL) models providing just the order and automatically constructs the underlying unrestricted and restricted error correction model (ECM). It also performs the bounds-test for cointegration as described in Pesaran et al. (2001). https://cran.r-project.org/web/packages/ARDL/index.html https://github.com/Natsiopoulos/ARDL

Tidy tools for time series modeling under tidyverts: https://tidyverts.org - The fable package applies tidyverse principles to time series modeling used for forecasting: https://fable.tidyverts.org/ - The tsibble package provides a tidy data structure for time series: https://cran.r-project.org/web/packages/tsibble/index.html - The tsibbledata package provide a different types of datasets in the tsibble data structure: https://cran.r-project.org/web/packages/tsibbledata/index.html - The tsibbletalk package introduces shared key to the tsibble, to easily {crosstalk} between plots on both client and server sides (i.e. with or without shiny): https://cran.r-project.org/web/packages/tsibbletalk/tsibbletalk.pdf- The feasts package provides a collection of features, decomposition methods, statistical summaries and graphics functions for the analysing tidy time series data: https://cran.r-project.org/web/packages/feasts/index.html - The fable.prohphet package provides an interface allowing the prophet forecasting procedure to be used within the fable framework: https://cran.r-project.org/web/packages/fable.prophet/vignettes/intro.html

The xts or Extensible Time Series package provides an extensible time series class, enabling uniform handling of many R time series classes : https://cran.r-project.org/web/packages/xts/index.html xts: Extensible Time Series: https://cran.r-project.org/web/packages/xts/vignettes/xts.pdf

Think about dummies, breaks, outliers

Figure out how bimets deals with ragged edge, add-factors, goal search

The bimets is an R package developed with the aim of easing time series analysis and building up a framework that facilitates the definition, estimation and simulation of simultaneous equation models: https://cran.r-project.org/web/packages/bimets/index.htmlbimets - Time Series And Econometric Modeling In R: https://github.com/cran/bimets https://cran.r-project.org/web/packages/bimets/vignettes/bimets.pdf

Structural Equation Models (SEM): https://rviews.rstudio.com/2021/01/22/sem-time-series-modeling/

Look at tidy models

The tidymodels package is a collection of packages for modeling and machine learning using tidyverse principles: https://www.tidymodels.org

Port the Gekko code into R: http://t-t.dk/gekko/

Look at DiagrammeR package, also the Gantt charts it can produce

https://rich-iannone.github.io/DiagrammeR/

A Beginner's Guide to Learning R:

A (very) short introduction to R: https://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf

Rstudio Education: https://github.com/rstudio-education

Remaster the tidyverse: https://github.com/rstudio-education/remaster-the-tidyverse

Introduction to R and Rstudio: https://jules32.github.io/2016-07-12-Oxford/R_RStudio/

An intro to R for new programmers: https://rforcats.net

fasteR: Fast Lane to Learning R!: https://github.com/matloff/fasteR

RStudio Cheatsheets: https://rstudio.com/resources/cheatsheets/

R for Data Science: https://r4ds.had.co.nz

Data wrangling, exploration, and analysis with R: https://stat545.com

R Markdown: The Definitive Guide: https://bookdown.org/yihui/rmarkdown/

Data Visualization with R: https://rkabacoff.github.io/datavis/

Modern R with the tidyverse: https://b-rodrigues.github.io/modern_R/

R Cookbook, 2nd Edition: https://rc2e.com

Advanced R by Hadley Wickham: http://adv-r.had.co.nz

UC Business Analytics R Programming Guide: http://uc-r.github.io/descriptive

R Programming for Data Science: https://bookdown.org/rdpeng/rprogdatascience/

Hands-On Programming with R: https://rstudio-education.github.io/hopr/

Efficient R programming: https://csgillespie.github.io/efficientR/index.html

R for Fledglings: http://www.uvm.edu/~tdonovan/RforFledglings/index.html

R Intermediate Level (includes applications):

Advanced Statistical Computing: https://bookdown.org/rdpeng/advstatcomp/

Feature Engineering and Selection: A Practical Approach for Predictive Models: http://www.feat.engineering/index.html

Advanced Quantitative Methods: https://uclspp.github.io/PUBLG088/index.html

Principles of Econometrics with R: https://bookdown.org/ccolonescu/RPoE4/

Modern Data Analysis for Economics: https://jiamingmao.github.io/data-analysis/Resources/

Data Science for Economists: https://github.com/uo-ec607/lectures

Data Science for Psychologists: https://bookdown.org/hneth/ds4psy/10-time.html

Rewriting R code in C++: https://adv-r.hadley.nz/rcpp.html

Writing R Extensions: https://cran.rstudio.com/doc/manuals/r-devel/R-exts.html

Other R packages for data analysis:

The data.table package is used for fast aggregation of large data (e.g. 100GB in RAM), fast ordered joins, fast add/modify/delete of columns by group using no copies at all, list columns, friendly and fast character- separated-value read/write: https://cran.r-project.org/web/packages/data.table/

The mlr3 (Lang et al. 2019) package and ecosystem provide a generic, object-oriented, and extensible framework for classification, regression, survival analysis, and other machine learning tasks for the R: https://mlr3book.mlr-org.com

purr package tutorial: https://jennybc.github.io/purrr-tutorial/

Data Visualization with R:

Data Analysis and Visualization Using R: http://varianceexplained.org/RData/

Data Analysis and Visualization in R for Ecologists: https://datacarpentry.org/R-ecology-lesson/

Data Visualization with R by Rob Kabacoff: https://rkabacoff.github.io/datavis/

R Graphics Cookbook, 2nd edition: https://r-graphics.org

ggplot2: elegant graphics for data analysis: https://ggplot2-book.org