Generate R Package Example
roxygen2 and pkgdown

Go back to fan's R4Econ.

1 Objective

Document and generate sharable R package.

1. Use roxygen2 from Hadley to document package.
2. Use pkgdown to package file and publish to github pages.

2 File Structure and Naming Convention

Folders:

1. R function functions in the */R* folder
2. RData files in the */data* folder

Naming Conventions:

1. functions files and functions all should be *snake_case_names*
2. function name prefix:
   - *fs_*: for non-project specific script files
   - *ffs_*: for project specific functions script files
   - *fv_*: for non-project specific vignettes files, generally RMD
   - *ffv_*: for project specific functions vignettes files, generally RMD
   - *ff_*: for non-project specific functions files
   - *ffp_*: for project specific functions files
   - *ffy_*: for project specific utility files
   - Each function file could be prepared to have multiple functions inside, each file have the root which is the function name.

Additionally, follow these general structures for *functions* in */R* folder:

1. 3 letter/digit project name
2. 3 letter/digit current file name, in R4E, three letter func group name: *ffp_opt_lin.R*
3. 5 letter/digit function name within file: *function ffp_opt_lin_solum*

With this structure, we end up with potentially fairly long names, but hopefully also not too long and clear.

Additionally, follow these general structures for *vignettes* in */vignettes* folder:

1. Follow conventions for function name vignettes should be associated with functions
2. Append onto existing function name 5 letter/digit to describe what this vignettes is suppose to achieve: *ffv_opt_lin_solum_vign1.Rmd*.
   - note the file starts with *ffv*

3 Create R Project Directory

3.1 Folder does not exist yet

If the project/folder does not yet exist:

1. *devtools:create* the folder of interest
2. Move your files over to */R* and */data* folders.
    - Write files following conventions above and with r descriptions
3. *pkgdown::build_site()*

```
devtools::create("C:/Users/fan/R4Econ")
```

3.2 Folder already exists

If there is already a folder with a bunch of files including R and not R files, and the folder needs to be converted to become a R package and was previously not a R package.

The idea is to use a folder somewhere to generate a generic template folder. This folder will have files and structure we need for our actual folder. Each time, we will just copy that folder's contents into the folder that we want to turn into a R folder. And do a global search to replace the template folder's folder name with the actual project name. To avoid confusion, generate this folder outside of an existing R package.

If there is already a R folder in your existing project, delete that or rename that and move files back into the */R* folder after completion. Make sure there are no duplicate folder or file names in the old and the new project.

Search replace the word *rprjtemplate* inside your old project folder with the new files, replace that with your project name. Should appear in three different spots.

3.2.1 R project template

The [rprjtemplate][https://github.com/FanWangEcon/Tex4Econ/tree/master/nontex/rprjtemplate] serves this templating role.

```
# Create project
devtools::create("C:/Users/fan/Tex4Econ/nontex/rprjtemplate")
```

Running the *devtools::create()* command will create the core needed folder structure with:

1. NAMESPACE
2. DESCRIPTION
3. *.gitignore*
4. *.Rbuildignore*
5. *.Rproj*
6. Empty R folder

Now customize this folder for future use with

1. custom *.gitignore*
2. MIT LICENSE
3. etc.

For adding vignette later:

Add these to the DESCRIPTION file: - Suggests: knitr, rmarkdown - VignetteBuilder: knitr

4 Develop and Package R Project

Developing a package means write various vignettes and functions. My process is to write various vignettes first, to basically test out core functionalities. Then convert parts of those vignettes to functions.

As functions are developed, some vignettes can be invoked differently, taking advantage of newly written functions. Make sure keep the original vignette that is not dependent on any other functions for ease of going to the initial point and figuring out what is happening. Names could be written to indicate vignette vintage.

4.1 Development Process

1. Write original vignette with dependencies, minimize dependencies, for example, do not import tidyverse, since package will not allow for all tidyverse components to be added, too much space needed.
   - this is not really a vignette, it is a function testing sandbox that does not call any package functions but provides line by line descriptions for the core functions with print outputs.
2. Test vignette inside Rstudio/VSCode etc. (These vignette do not depend on the function to be written below, they do not exist yet)
3. When raw vignettes work, start converting vignette components to functions, include minimal dependencies,
   - in function: *@import*
   - add to DESCRIPTION: **usethis::use_package("tidyr")** if tidyr is needed as *IMPORT*
   - *DEPENDS* vs *IMPORTS*
   - *important*: slowly check what does the function depend on, for example *R4Econ* has its own dependencies, when installing as packages, make those dependencies explicit in the new function if it calls *R4Econ*. Otherwise, *test()* might throw error. This is resolved if *R4Econ* add several things under depends.
4. Run the documentation tool:
   - **setwd('C:/Users/fan/PrjOptiAlloc')**
   - **devtools::document()**
   - this updates *NAMESPACE*
   - generates new *.Rd* files
5. Test @examples from Roxygen documentations
   - **devtools::run_examples()**
   - this tests if *@examples* are working.
6. Continue 3 to 4 - [ ] o convert various components of vignette.
7. Build package and load package using:
   - **devtools::check()**
   - this will run *run_examples* as well, but at the very end, will test everything for package.
8. Now install package
   - **devtools::install()**
   - *library(PrjOptiAlloc)* should work now
   - *ls("package:PrjOptiAlloc")* to list all objects in package
   - **devtools::reload()**
9. Publish webpage, review local, using *pkgdown*:
   - **pkgdown::build_site()**

```
# Step 3
rm(list = ls(all.names = TRUE))
setwd('C:/Users/fan/PrjOptiAlloc')
devtools::document()
# Step 4
devtools::run_examples()
# Step 6
```

```
devtools::check()
# Step 7
devtools::install()
library(PrjOptiAlloc)
ls("package:PrjOptiAlloc")
devtools::reload()
# Step 8
pkgdown::build_site()
```

4.2 Attach vs Import

Packages that are loaded in under *usethis::use_package** are imports, not attached.

According to Wickham: - *apkg::bfnc()*: when packages are loaded, its components can be accessed with the *::*. That is, if I want to use the *document()* from from *devtools()*, if I first attach, I can directly call document(). Without attaching, just loading, I have to do *devtools::document()*. - *attach(), library()* : when attaching, must first load. After attached, can directly call function names, for example, *filter*, it is on the search path.

See How R Searches and Finds Stuff:

> The better solution would have been to stuff reshape's cast() function into imports:ggplot2 using the Imports feature. In that case, we would have travelled from 2 to 3 and stopped. Now you can see why the choice between Imports and Depends is not arbitrary. With so many packages on CRAN and so many of us working in related disciplines its no surprise that same-named functions appear in multiple packages. Depends is less safe. Depends makes a package vulnerable to whatever other packages are loaded by the user.

This is question as well.

4.3 Dealing with Datasets

Datasets to be used with the project should be in the */data* folder. The name of the data file should appear in several spots and be consistent, suppose data is called *abc*

1. Rdata file name: */data/abc.Rdata*
2. Rdata file contains a dataframe inside that has to be called abc as well, so open the Rdata file inside Rstudio or R, what is the file called? is it abc? Look under environment
3. In the */R/ffp_abc.R*, the last line should be "abc", but the file name does not need to be.
4. If any functions uses a dataset, this could be done in two ways:
   - a dataset is a parameter for a function. In @example, can load in dataframes declared in dependencies.
   - inside a function, if direclty use a dataframe, that data frame should be declared following 1 to 3 here in */data*, make sure it has the same name, and write: *data(data_name)* at the beginning of the function to explicitly load the function in.

4.4 Logging and Printing

Logging and printing control is as usual important. See:

1. Why is message() a better choice than print() in R for writing a package?
2. Conditions

For simple package, for regression results, table outputs etc, use *print*, print statements can be suppressed by *invisible(capture.output(abc <- ffy_opt_dtgch_cbem4()))*.

4.5 Build Project

check to make sure file structures etc are all correct. Will also test the @examples inside r functions in ROxygen comments.

```
# will check the file structure, but also @example in functions.
devtools::check('C:/Users/fan/PrjOptiAlloc')
devtools::check('C:/Users/fan/PrjOptiAlloc', manual=FALSE)
devtools::run_examples('C:/Users/fan/PrjOptiAlloc')
devtools::build('C:/Users/fan/PrjOptiAlloc')
```

4.6 R project build and generate documentations

Assuming that we have used roxygen2 formats to write functions, now generate *.Rd* automatically with the *document()* function. This should create a */man* folder in which various *.Rd* files are stored. Note that their .Rd are for specific R functions, not for files that contain multiple functions.

```
# add dependencies:
setwd('C:/Users/fan/PrjOptiAlloc')
usethis::use_package("tidyr")
usethis::use_package("dplyr")
usethis::use_package("stringr")
usethis::use_package("broom")
usethis::use_package("ggplot2")
usethis::use_package("R4Econ")
# document will generate new NAMESPACE, so after dependencies added, need to document.
# funciton import must be consistent with these
devtools::document('C:/Users/fan/PrjOptiAlloc')
```

4.6.1 Vignettes Generation

To generate Vignettes, follow instructions here:

1. create a vignette directory
2. put RMD files in there, RMD front matter should look like below
3. add to DESCRIPTION file:
   - Suggests: knitr, rmarkdown
   - VignetteBuilder: knitr
4. run *devtools::build_vignettes()*. Running build should build all? automatically builds vignette

```
title: "Put the title of your vignette here"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Put the title of your vignette here}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
```

```
devtools::build_vignettes('C:/Users/fan/PrjOptiAlloc')
devtools::check()
```

4.7 R project build site with pkgdown

Once we have used *pkgdown::build_site()*, by default, all html and other package presentation files and reference files are saved in the */docs* folder, build_site is for building site, not for generating */man* documenation. If R files change, need to document first, before rebuilding site.

```
pkgdown::build_site("C:/Users/fan/PrjOptiAlloc")
pkgdown::build_site("C:/Users/fan/PrjOptiAlloc",build_vignettes=FALSE)
```