- Tutorial

Organização de repositórios

Atualmente se fala muito em ciência aberta, um tema que inclui não apenas os dados e as publicações científicas, mas também os códigos e programas usados para gerar resultados e produtos científicos. Mas não podemos ter ciência aberta de qualidade se a ciência não é repetível e reprodutível. Isto quer dizer que devemos conduzir estudos científicos de modo que qualquer pessoa possa reproduzi-los (i.e. reprodutibilidade), incluindo nós mesmo no futuro (i.e. repetibilidade).

A reprodutibilidade é um pilar essencial da ciência que nem sempre é respeitado pelos cientistas. Principalmente, no que se refere aos arquivos, códigos e programas usados para gerar os resultados científicos. Contudo, para que a ciência seja realmente reprodutível, precisamos abrir não apenas os dados e publicações, mas também nossos scripts, funções e programas necessários para reproduzir um estudo qualquer. Podemos chamar isso de reprodutibilidade analítica. O assunto é tão importante que ganhou o seu próprio CRAN Task View.

O primeiro passo para atingir a reprodutibilidade analítica é manter diretórios de trabalhos organizados, compreensíveis e bem documentados. Isso garante que qualquer pessoa (incluindo outros pesquisadores, os seus colaboradores e o seu Eu futuro) possa facilmente navegar e entender como o seu diretório está organizado, o que ele faz e como.

Não existe uma fórmula ou receita mágica de como organizar o seu diretório. O que existem são recomendações gerais do que fazer e também do que evitar, independente se você está trabalhando no R ou em qualquer outra linguagem de programação. Assim, neste tutorial iremos apresentar algumas boas práticas de organização do seu diretório de trabalho:

Organização em pastas

Pasta raiz

Para garantir que seu projeto científico seja reprodutível, é importante que ele tenha o seu próprio repositório (ou pasta), separado dos seus demais projetos. Para esse tutorial, foi criada previamente uma pasta chamada 'projetoX' e nela foi salvo um `.Rdata` a partir do qual uma sessão do R foi iniciado para esse tutorial. Já vimos como fazer isso no início do curso no nosso tutorial 1b. Para executar esse tutorial você deve fazer o mesmo no seu computador.

Essa é a pasta raiz do seu projeto e ela deve ser nomeada de uma maneira que faça sentido para o seu projeto (e.g. nome do projeto, seu acrônimo ou assunto). Salve essa pasta em um local que ajude

você a localizá-lo facilmente.

Para garantir a reprodutibilidade do seu projeto essa pasta deve conter todos os arquivos (i.e. dados, scripts, funções, etc.) necessários para a sua execução do início ao fim. Podemos criar rotinas para obter dados e funções/programas que estão em outros locais, mas esses locais devem ser acessíveis a qualquer pessoa (e.g. um site na Internet). Exceto para caso onde há restrições de acesso por qualquer motivo, evite o uso no seu projeto de arquivos que estão em outras pastas do seu computador, para os quais os demais pesquisadores não terão acesso.

Para saber qual é o caminho para o diretório que estamos atualmente trabalhando usamos a função `getwd()`:

```
getwd()
[1] "C:/Users/renat/Documents/Ensino/disciplinas/pos/R/projetoX"
```

O caminho acima é válido para o computador no qual o tutorial foi preparado.

Note que por enquanto não há nada nesse diretório. Podemos ver isso com usando a função `dir()` ou o seu alias `list.files()`:

```
dir()
list.files()
```

Como iniciamos uma sessão do R a partir do `.Rdata` previamente salvo dentro do diretório 'projetoX', `dir()` e `list.files()` assumem que nós queremos saber qual é o conteúdo dessa mesma pasta 'projetoX'. Mas você pode usar o argumento `path` dessas funções para saber o conteúdo de qualquer diretório no seu computador ou apenas de subpastas dentro da nossa pasta 'projetoX', se existirem.

Organização hierárquica

Para aumentar a organização e facilidade de compreensão do seu diretório, podemos organizá-lo de maneira hierárquica em pastas e subpastas.

Essas pastas devem ter nomes intuitivos, refletindo o seu conteúdo. Além disso, devemos evitar ter muitas pastas ou arquivos na raiz do projeto ou pastas com nomes parecidos, que possam gerar confusão. Isso vale também para as subpastas dentro de outra(s) pasta(s).

Em repositórios para projetos de pesquisa, em geral recomenda-se ter ao menos quatro pastas principais que irão conter os (1) dados, (2) códigos, (3) funções e (4) resultados. Claro que outras pastas podem e devem ser criadas de acordo com as necessidades de cada projeto.

Pasta de dados

Essa é a pasta que irá conter todos os dados necessários para executar o projeto. Como dito acima, no caso de arquivos que podem ser obtidos livremente online, não há necessidade de armazenar

localmente os arquivos nessa pasta. No caso de dados obtidos de terceiros (e.g. repositórios online), recomenda-se automatizar ao máximo a aquisição dos dados via páginas da internet ou uma API do repositório de informações.

Uma boa prática é conter ao menos duas subpastas na pasta de dados: uma subpasta que contém os dados brutos e outra contendo os dados editados ou processados do projeto.

Os dados brutos são apenas para leitura e devem ser armazenados como obtidos ou fornecidos. Nunca edite os dados brutos manualmente! A edição deve ser feita através de códigos para deixar documentada cada alteração realizada nos dados de origem.

Os dados processados são aqueles que resultam da edição dos dados originais ou, dependendo do projeto, dos resultados intermediários pós-processamento dos dados originais. Há uma linha tênue que separa dados processados dos resultados do projeto. Mas em geral, dados processados são aqueles que não pretendemos comunicar ao nosso público final.

Em geral, a pasta de dados é chamada de 'data', a subpasta dos dados brutos é chamada de 'raw' ou 'raw-data' e a subpasta dos dados processados é chamada de 'derived', 'derived-data', 'edited', 'edited-data', 'processed' ou 'processed-data'.

Dependendo dos tipos de dados usados pelo projeto, é essencial fornecer informações associadas às informações, os chamados metadados. Isso inclui os dicionários de dados (i.e. descrição de cada variável, sua natureza, unidade e exemplos), maneira de citação dos dados, licenças de uso, data de download de dados de terceiros, etc. Essas informações podem ser organizadas em uma terceira subpasta comumente chamada de 'metadata' ou na própria subpasta com os dados brutos.

Podemos usar a função `dir.create()`, que nos ajude na criação de pastas e subpastas no nosso diretório. A função `file.path` nos ajuda a criar caminhos para as nossas pastas e subpastas.

```
pasta <- "data"
dir.create(pasta)
dir()

subpastas <- c("raw-data", "derived-data")
caminhos <- file.path(pasta, subpastas)
dir.create(caminhos[1])
dir.create(caminhos[2])
dir(pasta)
```

Agora podemos obter e salvar dados disponíveis online na nossa pasta de dados brutos. Por exemplo, vamos baixar uma planilha disponível em nosso wiki contendo dados de árvores amostradas pela Andrea Vanini em três caixetais diferentes do litoral de SP:

```
url <- "http://ecor.ib.usp.br/lib/exe/fetch.php?media=dados:caixeta.csv"
caminho <- file.path(caminhos[1], "caixeta.csv")
download.file(url, destfile = caminho)
```

Agora vá no seu explorador de arquivos e deverá haver um arquivo na nossa subpasta 'raw-data' com o nome definido acima.

Pasta de funções

Se o projeto necessitar de funções próprias além daquelas contidas no R e seus pacotes contribuídos, essas funções deverão ser armazenadas em uma pasta específica. Em geral, essa pasta se chama 'R'.

```
dir.create("R")
dir()
```

Veja que agora já temos duas pastas no nosso projeto, `data/` e `R/`. Se criarmos uma função qualquer para o nosso projeto, podemos salvá-la nessa pasta:

```
eq_volume <- function(dap,
                      b0 = -8.889,
                      b1 = 1.881,
                      b2 = 0.875,
                      ff = 0.87) {

  volume <- ff * (exp( b0 + b1 * log(dap) + b2 * log(dap)))
  return(volume)
}
dump("eq_volume", file="R/eq_volume.R")
```

A função que acabamos de criar deveria ser documentada, permitindo que qualquer pessoa entenda seus objetivos, escopo e uso. Também é importante incluir exemplos de como usá-la e, se possível, testes para confirmar se ela retorna o que esperaríamos sob diferentes tipos de objetos ou opções de parâmetros. Mas, para os fins deste tutorial, vamos deixá-la assim bem simples.

O importante é essa função (e quaisquer outras funções relevantes ao projeto) podem ser disponibilizadas na sua área de trabalho usando a função `source()`:

```
rm(list = "eq_volume")
ls()
source(file = "R/eq_volume.R")
ls()
```

Se você tiver várias funções internas ao seu projeto, você pode salvar todas essas funções em apenas um script (em geral chamado de 'functions.R' ou 'internals.R') ou ter um script separado por função (mais indicado para projetos maiores). Nesse último caso você terá que usar a função `list.files()` e aplicar a função `source()` para cada arquivo usando `sapply()`:

```
rm(list = ls())
arquivos <- list.files("R", full.names = TRUE)
sapply(arquivos, source)
eq_volume(10)
```

Pasta de códigos

Nesta pasta são armazenados todos os códigos ou scripts necessários para executar o projeto. Esses códigos podem ser para obter dados online, fazer edições ou análises preliminares, gerar gráficos... o

que o seu projeto precisar!

Para esses códigos algumas boas práticas são:

1. documente o código (titulo, objetivo, autor data, etc.);
2. comente abundantemente e detalhadamente cada passo do código;
3. no fim do código, salve os objetos que serão necessários posteriormente e remova os objetos intermediários gerados;
4. sempre usar caminhos relativos para ler e salvar arquivos;
5. seja constante no estilo de programação ao longo de todos o código e entre códigos (e.g. uso de ← ao invés de =);
6. inclua testes de erro para as diferentes etapas do código, visando detectar problemas o mais cedo possível (veja a função `stopifnot()`);
7. defina fora do código as variáveis que são necessárias, mas que podem variar se você quiser alterar algo feito no código (e.g. número de simulações usadas);
8. se o resultado depende do estado atribuído a essas variáveis globais, inclua o estado no nome do arquivo salvo pelo código.

Essa pasta pode se chamar 'codes', 'scripts' ou 'analyses':

```
  dir.create("scripts")
```

Para simplificar o tutorial, vamos usar um script já pronto no qual são feitos: a leitura dos dados dos caixetais, sua edição, uma figura exploratória, uma tabela com as médias do volume por caixetal e a tabela com o resultado da análise de variância sobre o modelo testando se há diferença entre as áreas na média do logaritmo do volume de suas árvores. Nesse script usamos a função `eq_volume()` que armazenamos na nossa pasta `R/`.

```
  url <-
"http://ecor.ib.usp.br/lib/exe/fetch.php?media=dados:script_tutorial_organiz
a_diretorio.R"
  caminho <- file.path("scripts", "analisa_dados.R")
  download.file(url, destfile = caminho)
```

Pasta de resultados

Por fim, a pasta de resultados conterá as figuras, tabelas e quaisquer outros resultados gerados durante o projeto e que serão importantes para comunicar os seus resultados. Você pode colocar nessa pasta ou em uma pasta separada o resultado de compiladores de texto do tipo '.md', '.tex', '.pdf', etc.

Como dito acima, use nomes intuitivos e informativos para saber o que cada arquivo de resultado contém sem ter que abri-lo para inspeção (e.g. ‘tabela1_comparacao_media_volume_500_simulacoes.csv’)

```
  dir.create("output")
  dir()
```

Agora temos todas as subpastas criadas e com todos os arquivos que precisamos dentro delas para podermos executar o nosso 'projetoX'. Mas antes disso, precisamos documentar um pouco melhor o

nosso repositório para garantir que essa execução seja facilmente reproduzida.

Nome	Data de modificação	Tipo	Tamanho
data	15/09/2024 22:01	Pasta de arquivos	
output	15/09/2024 22:01	Pasta de arquivos	
R	15/09/2024 22:01	Pasta de arquivos	
scripts	15/09/2024 22:01	Pasta de arquivos	
.RData	13/09/2024 11:15	R Workspace	1 KB
make.R	15/09/2024 22:01	Arquivo R	1 KB
README.txt	15/09/2024 22:01	Text Document	1 KB

Documentação do repositório

Na raiz do repositório do nosso projeto é importante incluir uma documentação mínima sobre o projeto e como executá-lo.

Arquivo 'leia-me'

Convencionalmente usados no desenvolvimento de softwares, os arquivos do tipo README são arquivos de texto que contém informações sobre o diretório, os demais arquivos e, em nosso caso, sobre o projeto em si.

Esses arquivos contém uma visão geral do projeto, instruções sobre instalação/execução/operação (dependendo da finalidade do projeto), requisição de sistema e programas, uma descrição de como os arquivos estão organizados no diretório, como e onde reportar problemas, mudanças no projeto (i.e. log report), licenças de uso, contato e afiliação do(s) autor(es), créditos, agradecimentos, financiamentos, etc. Se necessário, informações como programas, licenças e log report podem ser arquivos à parte do README.

Em projetos no R atualmente os arquivos README são mantidos no formato 'Rmarkdown' (i.e. README.rmd) e vários pacotes contribuídos possuem funcionalidade para gerá-lo automaticamente quando estamos em um projeto ou pacote (veja os pacotes `devtools` e `usethis`).

Para esse tutorial faremos algo mais simples, em '.txt' mesmo:

```
  texto <- c("ProjetoX: comparando o volume arbóreo em caixetais do estado
de São Paulo",
           "Autor: professores e monitores do curso de R do IB e ESALQ,
USP",
           "Bugs: enviar problemas e sugestões para raflima@usp.br",
           "Organização: Este repositório contém na pasta 'data/' os dados
do projeto, na pasta 'scripts/' os códigos para executar o projeto, na pasta
```

```
'R/' as funções internas usadas nos códigos e na pasta 'output/' os
resultados dos códigos.",
          "Uso: para executar o projeto use o seguinte comando
'source(file.path('scripts', 'analisa_dados.R'))'",
          "Agradecimentos: PPG em Ecologia (IB) e em Recursos Florestais
(ESALQ)")
  writeLines(texto, "README.txt", sep = "\n\n")
```

Arquivo 'makefile'

Idealmente, o projeto deve ter um arquivo 'makefile', que ajuda na automatização da execução do projeto e portanto a sua reprodutibilidade. Esse arquivo carrega todos os pacotes, dados e funções necessárias para a execução do projeto. Ele também executa na ordem necessária todos os scripts necessários para a execução do projeto. Todas essas operações devem ser executadas neste arquivo de modo que quando lemos o arquivo usando a função `source()` todo projeto é executado sem necessidade de intervenção.

Atingir esse nível de automatização de projetos nem sempre é simples, pois alguns passos intermediários podem depender da intervenção do usuário. Alguns passos podem demorar para ser executados. Além disso, é preciso planejamento e organização para que todos os scripts possam ser executados sem maiores intervenções e que se haja algum erro ele possa ser rapidamente detectado e facilmente isolado para correção. Mas, ele garante que você, seus colaboradores e outros usuários não percam tempo tentando compreender todo o projeto, seus detalhes e a ordem necessária de execução dos scripts.

Uma forma de facilitar a reprodução de um projeto é salvar cada script com um prefixo numérico ordenado (e.g. '00_download_dados.R'), que denota a ordem na qual cada script deve ser executado. Contudo, como isso já está incorporado ao 'makefile' e como ele faz mais coisas que apenas executar os scripts em ordem, é sempre melhor ter um 'makefile' na raiz do seu projeto.

No exemplo do nosso 'projetoX' o nosso 'makefile' é bem simples, visto que não usamos nenhum pacote contribuído (além dos pacotes básicos do R), temos apenas uma função e um código nas pastas `R/` e `scripts/`, respectivamente. Assim o nosso 'makefile' pode ser algo simples assim:

```
  make <- c("## EXECUTANDO O PROJETO X ##",
          "\n## Instalando/carregando os pacotes necessários",
          "### (Nenhum pacote sendo usado atualmente)\n",
          "\n## Carregando as funções internas",
          "source(file.path('R', 'eq_volume.R'))\n",
          "\n## Executando os códigos do projeto",
          "source(file.path('scripts', 'analisa_dados.R'))")
  writeLines(make, "make.R", sep = "\n")
```

Dando tudo certo, tudo o que você precisa para repetir o seu estudo e que os seus colaboradores e pares precisam para reproduzi-lo se resume a linha de comando abaixo:

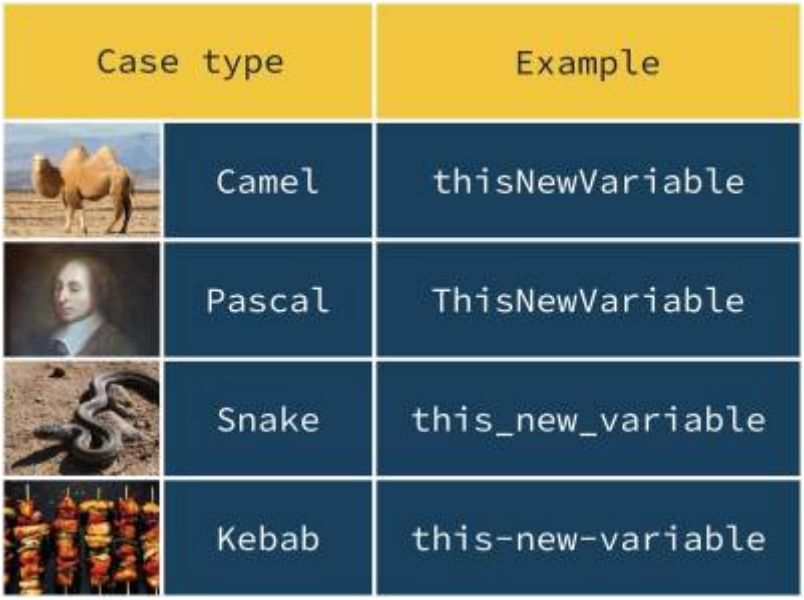
```
  source("make.R")
```

Nomeando arquivos

Tocamos brevemente nesse assunto ao longo desse tutorial, mas também existem boas práticas para a maneira com a qual nomeamos nossos arquivos e pastas visando aumentar a reprodutibilidade de nossos projetos.

Em geral, esses nomes devem ser intuitivos de modo que qualquer pessoa possa inferir seu conteúdo sem ter que abrir o arquivo ou pasta. Além disso, pensando na fluidez do código e nos caminhos para ler e salvar arquivos, os nomes de arquivo devem ser curtos (em geral, menos de 50 caracteres), não devem conter caracteres especiais (e.g. ~, $, &, etc.) e, se possível, sem artigos e preposições (e.g. a, de, do, das, ...). Além disso, esses nomes não devem conter espaços, ou seja, use '-', 'snake_case' ou 'camelCase' para nomear seus arquivos!

Assim, exemplos de nomes de arquivos seguindo essas boas práticas seriam 'download_data.R', 'analise_dados.R', 'getCovariables.R', etc.



Se o projeto estiver bem organizado, documentado e estiver usando um programa para o controle de versão de arquivos (e.g. 'git' ou 'svn'), não é mais necessário incluir nos nomes dos arquivos coisas como: o nome do projeto, ou do programador/autor, localização, versão, datas, etc. Claro que isso nem sempre é válido para todos os projetos. Por exemplo, o uso de datas no nome do arquivo pode ser necessário para separar um conjunto de dados do outro ou dados coletados em um mesmo local, porém em momentos diferentes. Neste caso, use sempre o padrão 'YYYYMMDD' para as datas nos nomes dos arquivos.

Listando, copiando, movendo e apagando arquivos

Vimos acima que há funções para manipular pastas (i.e. funções `dir.create()` e `dir.exists()`). Mas há também uma família de funções para manipular arquivos. Essas funções são muito úteis para melhor organizarmos nossos diretórios de trabalho.

Por exemplo, vamos criar arquivos vazios na raiz do nosso diretório usando a função `file.create()`:

```
  file.create("simulação dos dados.R")
  file.create("funcao1.R")
  list.files()
```

Contudo, vimos acima que arquivos de códigos e funções devem ficar nas pastas 'scripts' e 'R', e não na raiz do nosso projeto. Então podemos copiar ou mover esses arquivos para as pastas certa usando as funções `file.copy()` and `file.rename()`:

```
  file.copy(from= "simulação dos dados.R",
    to= file.path("scripts", "simulação dos dados.R"))
  file.rename(from= "funcao1.R",
    to= file.path("R", "funcao1.R"))
  list.files()
```

Note que enquanto uma função 'copia e cola' a outra 'recorta e cola'. Então precisamos apagar o arquivo da raiz do projeto:

```
  file.remove("simulação dos dados.R")
  list.files()
```

Ok, tudo bem mais organizado agora! É claro que poderíamos ter feito isso diretamente no explorador de arquivos do computador. Contudo, essas funções permitem que você localize, renomeie e mova muitos arquivos de uma só vez.

E para seguir as boas práticas que listamos acima vamos renomear esses arquivos:

```
  arquivo1 <- list.files(path = 'scripts', pattern = 'ç', full.names = TRUE)
  arquivo1.edit <- sub('simulação dos dados', 'simula_dados_volume_medio',
arquivo1)
  file.rename(from= arquivo1, to= arquivo1.edit)


  arquivo2 <- list.files(path = 'R', pattern = '1', full.names = TRUE)
  arquivo2.edit <- sub('funcao1', 'simula_media', arquivo2)
  file.rename(from= arquivo2, to= arquivo2.edit)
```

Pronto! O diretório do nosso projeto está organizado e automatizado!

Para saber mais

Legal, você já sabe o básico para melhor organizar o seu diretório de trabalho e para aumentar o seu nível de documentação e automatização. Veja abaixo algumas sugestões para você se aprofundar no assunto:

- Artigo Wilson et al. (2017) 'Good enough practices in scientific computing' PLoS Comp Biol 13(6): e1005510.
- Capítulo 1 do livro Douglas et al. (2024) 'An introduction to R'. (https://intro2r.com/)

Além desse material, segue uma lista de pacotes contribuídos do R para ajudar ou automatizar a importante tarefa de aumentar a sua reprodutibilidade analítica:

- Pacote ''here''
- Pacote ''usethis''
- Pacote ''rcompendium''
- Pacote ''starter''
- Pacote ''worcs''