Toward Reproducibility with R on Biowulf

Alexandra L Emmons, PhD & Joe Wu, PhD
BTEP/GAU/CCR/NCI/NIH - email ncibtep@mail.nih.gov
Bioinformatics Training and Education Program

Table of Contents

Course Overview

Lesson 1: Introduction to Biowulf, Unix, and R

Lesson 2: Getting Started with R on Biowulf

Lesson 3: R Project Management and renv

Lesson 4: Submitting R Scripts via command line

Additional Resources

Course Overview

Welcome to *Toward Reproducibility with R on Biowulf*

This course includes a series of four lessons designed for beginner to intermediate R users interested in working with R on Biowulf. The purpose of this course is to introduce the various ways to use R on Biowulf, while emphasizing reproducible practices such as project organization and R package dependency management. This course is not designed for advanced R users.

Course Expectations

This course will include a series of four, 1-hour lessons taught over four weeks. Lessons will be on Thursdays at 1 PM.

Course participants should have **beginner level knowledge** of working on the **Unix command line, Biowulf, and R**. While Lesson 1 will provide a refresher in these areas, this course is not recommended for novices.

> **Biowulf account required**
>
> In order to follow along with course lessons, participants are expected to have a Biowulf account. Instructions for obtaining an account can be found here *(https://hpc.nih.gov/docs/accounts.html)*. If you have an account but it has been inactive for more than 60 days, you will need to unlock your account. See instructions to unlock your account here *(https://hpc.nih.gov/docs/how_to.html)*. Email us at ncibtep@nih.gov *(mailto:ncibtep@nih.gov)* if you experience any issues.

Course topics

Lesson 1: Introduction to Biowulf, Unix, and R

Lesson 1 will serve as a course introduction and refresher on Unix, Biowulf, and R.

Lesson 2: Getting Started with R on Biowulf

In Lesson 2, partipants will learn about ways to use R on Biowulf. The focus will be on interactively working with R on Biowulf. Two different ways of accessing RStudio will be demonstrated. In addition, there will be a discussion on R modules and setting up custom R libraries.

Lesson 3: R Project Management and renv

Lesson 3 will focus on enhancing reproducibility as you get started using R. In particular, participants will learn how to set up and organize an R project and use the `renv` package for R dependency management.

Lesson 4: Submitting R Scripts via command line

Lesson 4 will focus on using R from the command line and submitting R scripts using `sbatch` on Biowulf. There will also be a brief discussion on paralellizing R code.

Lesson 1: Introduction to Biowulf, Unix, and R

Learning Objectives

1. Learn about why you may want to use R on Biowulf.
2. Refresh Unix and R skills.

This lesson will not be hands on.

Why use R for bioinformatics?

R is both a computational language and environment for statitical computing and graphics. It is open-source and widely used, not just by bioinformaticians. R is a particularly great resource for statistical analysis, plotting, and report generation, and it has become a powerhouse for biological assay data analysis (e.g., RNA-Seq, sc-RNAseq, ChIP-seq, population genomics). Package repositories like Bioconductor have influenced the rise of R programming in the -omics fields.

What is Bioconductor?

Bioconductor is an R package repository for free open-source software that "facilitates rigorous and reproducible analysis of data from current and emerging biological assays" *(https://www.bioconductor.org/)*. Bioconductor is released semi-annually, with two working Bioconductor releases per every release of R. Packages in Bioconductor undergo rigorous testing to ensure the interoperability of included software.

Bioconductor not only provides methodologically based software packages, packages focused on offering new methods for the analysis of specific data types, but also software focused on core infrastructure. Package developers are encouraged to use existing Bioconductor infrastructure, for the storage and accession of data, to increase the usability of packages by minimizing the time spent learning new data structures for different workflows. This emphasis on common infrastructure classes makes the use of Bioconductor software scalable to emerging data types and methods. Developers can build off of existing infrastructure and methods to rapidly deploy new packages with technological advancements in the molecular sciences. Beyond software, Bioconductor offers other types of packages including those that focus on annotation, providing access to well known databases such as Entrez genes, Ensembl, UCSC, the Gene Ontology Consortium, KEGG, etc. In addtiion, there are experimental data packages that provide datasets for package validation or package tutorials, and workflow packages focused on combining aspects of multiple Bioconductor packages to complete a particular type of analysis.

The latest version of Bioconductor (v 3.17, compatible with R v.4.3) includes 2,230 software packages, 419 experiment data packages, 912 annotation packages, 27 workflows, and 3 books. The Bioconductor project strives to "further scientific understanding" through extensive documentation and training opportunities. Each package includes one or more quality vignettes outlining the use of included functions.

What is Biowulf, and why use R on Biowulf?

Biological datasets can be massive. Often our local computers (laptops, desktops) do not have the storage space or computational power to analyze these datasets. Biowulf is the NIH high performance compute cluster. It has greater than 90k processors, and can easily perform large numbers of simultaneous jobs. Biowulf also includes greater than 600 preinstalled scientific software and databases.

You should use Biowulf when: software is unavailable or difficult to install on your local computer and is available on Biowulf, you are working with large amounts of data that can be parallelized to shorten computational time, or you are performing computational tasks that are memory intensive.

Many of the initial data processing steps for most data types will be performed with unix-based bioinformatics software, often requiring one to use Biowulf, especially in the case of Window's users. Users may want to further analyze data output from these inital workflows, which can still include "large data", using Bioconductor or other R packages. Instead of transferring data from Biowulf to your local computer, it may be easier to use R directly on Biowulf compute nodes.

Warning

Never run computational tasks on the login node. Computational tasks on Biowulf should be submitted as a job (`sbatch`, `swarm`) or run through an interactive session (`sinteractive`).

Danger

Do not put data with PII (personally identifiable information), patient data for example, on Biowulf.

Getting a Biowulf account

If you do not already have a Biowulf account, you can obtain one by following the instructions here *(https://hpc.nih.gov/docs/accounts.html)*. NIH HPC accounts are available to all NIH employees and contractors listed in the NIH Enterprise Directory. Obtaining an account requires PI approval and a nominal fee of $35 per month. Accounts are renewed annually contigent upon PI approval.

When you apply for a Biowulf account you will be issued two primary storage spaces:
`/home/$USER` (16 GB)
`/data/$USER` (100 GB)

You may request more space in `/data/$USER` by filing an online storage request (https://hpcnihapps.cit.nih.gov/auth/dashboard/storage_request.php).

NIH HPC Documentation

The NIH HPC systems are well-documented at hpc.nih.gov *(https://hpc.nih.gov/)*. The User guides *(https://hpc.nih.gov/docs/user_guides.html)* , Training documentation *(https://hpc.nih.gov/training/)*, and How To *(https://hpc.nih.gov/docs/how_to.html)* docs are fantastic resources for getting help with most HPC tasks.

Additional help

- Contact staff@hpc.nih.gov *(mailto:staff@hpc.nih.gov)*
  The HPC team welcomes questions and is happy to offer guidance to address your concerns.

- Monthly Zoom consult sessions
  The HPC team offers montly zoom consult sessions. []"All problems and concerns are welcome, from scripting problems to node allocation, to strategies for a particular project, to anything that is affecting your use of the HPC systems. The Zoom details are emailed to all Biowulf users the week of the consult."](https://hpc.nih.gov/training/){target=_blank}

- Bioinformatics Training and Education Program
  BTEP is here to help with all training needs. We are happy to help you get started with Biowulf and begin analyzing your data. If you experience any difficulties or challenges, especially with different bioinformatics applications, please do not hesitate to email us *(mailto:ncibtep@nih.gov)*.

Unix Refresher

Biowulf computational nodes use a Unix-like (Linux) operating system (distributions RHEL8/ Rocky8). Unix is a proprietary operating system like Windows or MacOS (Unix based). There are many Unix and Unix-like operating systems, including open source Linux and its multiple distributions. Biowulf requires knowledge and use of the command line interface (shell) to direct computational functionality. To work on the command line we need to be able to issue Unix commands to tell the computer what we want it to do.

A basic foundation of Unix is advantageous for most scientists, as many bioinformatics open-source tools are available or accessible by command line on Unix-like systems.

How much Unix do I need to know to work on Biowulf?

As with any language, the learning curve for Unix can be quite steep. However, to work on Biowulf you really need to understand the following:

1. Directory navigation: what the directory tree is, how to navigate and move around with cd
2. Absolute and relative paths: how to access files located in directories
3. What simple Unix commands do: ls, mv, rm, mkdir, cat, man
4. Getting help: how to find out more on what a unix command does
5. What are “flags”: how to customize typical unix programs ls vs ls -l
6. Shell redirection: what is the standard input and output, how to “pipe” or redirect the output of one program into the input of the other --- Biostar Handbook *(https://www.biostarhandbook.com/introduction-to-unix.html)*

Accessing your local terminal or command prompt

Mac OS

- Type `cmd + spacebar` and search for "terminal". Once open, right click on the app logo in the dock. Select `Options` and `Keep in Dock`.

Windows 10 or greater

> You can start an SSH session in your command prompt by executing ssh user@machine and you will be prompted to enter your password. ---Windows documentation *(https://docs.microsoft.com/en-us/windows/terminal/tutorials/ssh?source=recommendations)*

To find the Command Prompt, type `cmd` in the search box (lower left), then press `Enter` to open the highlighted Command Prompt shortcut.

If this yields any major issues, try installing PuTTY *(https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html)*, Solar-PuTTY *(https://www.solarwinds.com/free-tools/solar-putty?a_aid=BIZ-PAP-CMPRTCH&a_bid=1bd20791&CMP=BIZ-PAP-CMPR_PCW-SolarPutty-FSPTY-LM&data1=&data2=list)*, or MobaXterm *(https://mobaxterm.mobatek.net/)*.

Unix commands to know

The following list is not comprehensive. Only select commands are included.

Navigating the file system

- `pwd` (print working directory)
- `ls` (list)
- `cd` (change directory), by itself will take you home, `cd ..` (will take you up one directory), `cd /results_dir/exp1` (go directly to this directory)

File management

- `touch` creates an empty file
- `nano` basic editor for creating small text files
- `rm` remove files or directories. Be careful!
- `mkdir` make a directory and `rmdir` (remove a directory with NO files)
- `mv` rename or move files and directories
- `less` and `more` view files; `less` can also be used to view zipped files on Biowulf. Use `q` to escape.
- `cp` copy files or directories
- `cat`, `head`, and `tail` - print to screen, print first few lines to the screen, print last few lines to the screen
- `zcat` viewing zipped files
- `chmod`,`chown` modify file / directory permissions
- `wc` number of lines (`-l`), words (`-w`), and bytes (`-c`, usually one byte per character); for number of characters use `-m`.
- `grep` search files using regular expressions
- `cut` cuts selected portions of a file (e.g., column selection)
- `sed` and `awk` - file editing (find and replace, column selection, filtering, etc.)

Obtaining help

- `help` display information about builtin commands
- `man` access online manual pages
- `-h`,`--help` flags for obtaining help

Useful information

- Flags and command options (`-`) are used to alter program functions
- Wildcards (e.g., `*`)
- Tab complete for less typing
- Accessing user history with the "up" and "down" arrows on the keyboard
- Working with file content (`<`, `>`, `>>`)
- Combining commands with pipe (`|`). Where the heck is pipe anyway?
- Performing repetitive actions with Unix (`for loop`), GNU `parallel`

File download

- `wget` The non-interactive network downloader
- `curl` transfer a URL

Remote connection

- `ssh` secure shell protocol for remote login to Biowulf / Helix

Biowulf

- `batchlim` show cpu and job limits for batch jobs
- `freen` show free and total nodes and cores
- `jobdata` show lots of info for a single jobid
- `sacct` select slurm jobs
- `sbatch` submit slurm job
- `scancel` delete slurm jobs
- `sinfo` view information about Slurm nodes and partitions
- `sinteractive` allocate an interactive session
- `sjobs` show brief summary of queued and running jobs
- `squeue` display status of slurm batch jobs
- `sstat` display various status information of a running job/step
- `swarm` submit a swarm of commands to cluster

Modules on Biowulf

- `module avail` list available applications on Biowulf
- `module load` load an application
- `module purge` purge applications

Resources for learning Unix

Learning Unix: Classes / Courses

- Introduction to Biowulf (May – Jun, 2023) *(https://bioinformatics.ccr.cancer.gov/docs/biowulf-introduction-summer-2023/index.html)*
- Introduction to Unix on Biowulf (Jan – Feb, 2023) *(https://bioinformatics.ccr.cancer.gov/docs/unix-on-biowulf-2023/index.html)*
- Bioinformatics for Beginners: Module 1 Unix/Biowulf *(https://bioinformatics.ccr.cancer.gov/docs/b4b/Module1_Unix_Biowulf/Lesson1/)*

Additional useful Unix resources

- BashScripting_LinuxCommands from the NIH HPC team *(https://hpc.nih.gov/training/handouts/BashScripting_LinuxCommands.pdf)*
- Fosswire linux reference sheet *(https://bioinformatics.ccr.cancer.gov/docs/b4b/fosswire_reference.pdf)*

R Refresher

R can be accessed from the command line using `R`, which opens the R console, or it can be accessed via and Integrated development environment (IDE) (e.g., RStudio, VSCode, etc.). R commands can be submitted together in a script or interactively in a console.

Navigating directories

`setwd()` Set working directory (equivalent to `cd`)
`getwd()` Get working directory (equivalent to `pwd`)

Getting help

`help()` and `?` "provide access to the documentation pages for R functions, data sets, and other objects".

`help.search()` "allows for searching the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries (or any combination thereof)"; equivalent to `??pattern`

`args()` returns information on function arguments including names and defaults

See more on getting help here *(https://www.r-project.org/help.html)*.

Installing and loading packages

To take full advantage of R, you need to install R packages. R packages are loadable extensions that contain code, data, documentation, and tests in a standardized shareable format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN) *(https://cran.r-project.org/)*. CRAN is a global network of servers that store identical versions of R code, packages, documentation, etc (cran.r-project.org).

> An R library is, effectively, a directory of installed R packages which can be loaded and used within an R session. ---renv *(https://rstudio.github.io/renv/articles/renv.html)*

`install.packages()` install packages from CRAN
`library()` load packages in R session

You will need to install and use the ***`BiocManager`*** *to install and use Bioconductor packages*:

```
if (!require("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install(version = "3.17")
```

`.libPaths()` reports the directory where your installed R packages are located.

`devtools::install_github()` to install an R package from Github

Commenting

You can annotate your code by starting annotations with #. Comments to the right of # will be ignored by R.

Use `# ----` to create navigable code sections.

Assignment operators

Anything that you want assigned to memory must be assigned to an R object.

`<-` the primary assignment operator, assigning values on the right to objects on the left.
= can also be used to assign values to objects, but is usually reserved for other purposes (e.g., function arguments)

Use `ls()` to list objects created in R. `rm()` can be used to remove an object from memory.

Object naming conventions

There are rules regarding the naming of objects:

1. Avoid spaces or special characters EXCEPT '_' and '.'
2. No numbers or underscores at the beginning of an object name.
3. Avoid common names with special meanings (See ?Reserved) or assigned to existing functions (These will auto complete).

> **Note**
>
> R is case sensitive, so an object with the name "FOO" is not the same as "foo".

Object data types

There are many functions in R to understand the types of objects you are working with. For example:

`class()` returns the class of an object
`typeof()` returns type or storage mode of object
`mode()` returns object storage mode

Importing and exporting data

Use the `read` functions to import data (e.g., `read.csv`, `read.delim`, etc.). Use `write` functions to export data (e.g., `write.table`).

Using functions

An R function is like a unix command. Functions perform specific tasks. R has a ton of built-in functions and functions available through additional packages. You can also create your own functions.

The general syntax for a function is the name followed by parantheses, `function_name()` (e.g., `round()`).

To create a function:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Vectors

> A vector is a collection of values that are all of the same type (numbers, characters, etc.) --- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html)*

`c()` - used to combine elements of a vector

When you combine elements of different types in the same vector, they are forced into the same type via "coercion" (logical < numeric < character).

`length()` - returns the number of elements in a vector

Use brackets to extract elements of a vector:

```
a <- 1:10
a[2]
```

Lists

Unlike vectors, lists can hold values of different types.

```
list(1, "apple", 3)
```

Data frames

Data frames hold tabular data comprised of rows and columns; they can be created using `data.frame()`.

To understand more about the structure of an object and data frame, consider the following functions:

`str()` displays the structure of an object, not just data frames
`dplyr::glimpse()` similar to `str` but applies to data frames and produces cleaner output
`summary()` produces result summaries of the results of various model fitting functions
`ncol()` returns number of columns in data frame
`nrow()` returns number of rows of data frame
`dim()` returns row and column numbers
`unique()` returns a vector of with duplicates removed; also see `dplyr::distinct()`

We can subset data frames using bracket notation:

```
```

```
df<- data.frame(Counts=seq(1,5), animals=c("racoon","squirrel","bird'
#to return just the animals column
df[,"animals"]
```

We can also use functions from `dplyr` such as `filter()` for subsetting by row and `select()` for subsetting by column.

Plotting

There are 3 primary plotting systems with R: base R, `ggplot2`, and `lattice`.

Check out the R Graph Gallery *(https://r-graph-gallery.com/)* for data visualization examples and code.

Getting info on R Session

`sessionInfo()` Print version information about R, the OS and attached or loaded packages.

Resources for learning R

Base R cheat sheet *(https://iqss.github.io/dss-workshops/R/Rintro/base-r-cheat-sheet.pdf)*

Other cheat sheets can be here *(https://posit.co/resources/cheatsheets/)*.

There is also a nice review here *(https://cosima.nceas.ucsb.edu/r-self-assessment/#section-r-overview)*.

BTEP courses

- R Introductory Series *(https://bioinformatics.ccr.cancer.gov/docs/rintro/index.html)*
- Data Wrangling with R *(https://bioinformatics.ccr.cancer.gov/docs/data-wrangle-with-r/)*
- Data Visualization with R *(https://bioinformatics.ccr.cancer.gov/docs/data-visualization-with-r/index.html)*

Test your Knowledge

Are your Unix skills satisfactory?

Complete the scavenger hunt from https://sanderslab.github.io/code/ *(https://sanderslab.github.io/code/)*.

Are your R skills ready?

Use this assessment *(https://cosima.nceas.ucsb.edu/r-self-assessment/)* to determine whether you need to further brush up on your R skills.

Do you need a Biowulf refresher?

So you think you know Biowulf? Quiz yourself using the hpc.nih.gov biowulf-quiz *(https://hpc.nih.gov/training/intro_biowulf/biowulf-quiz/)*.

Lesson 2: Getting Started with R on Biowulf

Learning objectives

1. Understand how R can be deployed on Biowulf
2. Understand how to access and use R modules
3. Learn to create a custom R library on Biowulf

Deploying R on Biowulf

There are multiple ways to use R on Biowulf. See the HPC documentation *(https://hpc.nih.gov/apps/R.html)*.

> **Note**
>
> R sessions are not allowed on Helix or the login node. All R sessions must use computational nodes.

1. Interactively

   Your workflow may require some element of interactivity (e.g., modifying code based on graphical output). In such cases, users generally like to use an IDE (Integrated development environment). The preferred IDE for R programming is generally RStudio. However, if you are expeirencing significant lag, there are other options including Jupyter Lab and VSCode.

   - RStudio *(https://hpc.nih.gov/apps/RStudio.html)*

     There are currently 2 ways to run RStudio on Biowulf.

     1. Using NoMachine (To be demoed)

        - To get started, you will need to install NoMachine (NX) *(https://hpc.nih.gov/docs/nx.html)*, "a graphical client that presents a full virtual Linux desktop to a window on the user's local machine".
        - Once NoMachine is installed, follow these instructions *(https://hpc.nih.gov/apps/RStudio.html)* to start RStudio.

> **Warning**

> NoMachine uses X11 forwarding and will experience lags.

1. Using RStudio Server *(https://hpc.nih.gov/apps/rstudio-server.html)* (**Warning: Under development**) (To be demoed)

- Jupyter Lab *(https://hpc.nih.gov/apps/jupyter.html)*
- VSCode *(https://hpc.nih.gov/apps/vscode.html)*
  - To use the VSCode R extension, use these instructions.
- Connecting and using just an R console (Lesson 1)

  This is how we will use R in today's lesson.

2. Submitting R scripts via sbatch (Lesson 4)

   To submit an R script from command line, you can use the command `Rscript` or `R CMD BATCH`. `Rscript` is preferred and prints output to stdout. `R CMD BATCH` prints R commands and output to a `.Rout` file. See more information here *(https://support.posit.co/hc/en-us/articles/218012917-How-to-run-R-scripts-from-the-command-line)*.

Remove your hands from your keyboard, sit back, and enjoy a demo on how to use the RStudio IDE on Biowulf. If you intend to use an IDE to interact with R on Biowulf and you experience difficulties in the future, please email us at ncibtep@nih.gov *(mailto:ncibtep@nih.gov)*.

Connect to Biowulf (Hands-on)

To connect to Biowulf, you must be on the NIH network, either on campus or via VPN.

We will then connect using an `ssh` protocol.

Open your terminal if on a mac or the command prompt if using a Windows and type the following:

```
ssh username@biowulf.nih.gov
```

Replace `username` with your NIH user name. You will then be prompted for your NIH password.

> **Note**
>
> The cursor will not move nor will you be able to see what you type when entering your password.

Getting started with R

We will be working with R from our `/data/$USER` directory. There is not much space in ~ (16 GB), so it is good practice to always `cd` to `/data/$USER`.

```
cd /data/$USER
```

Info

$USER is an environment variable. You can read more about environment variables here *(https://www.geeksforgeeks.org/environment-variables-in-linux-unix/)*.

The default R installation on Biowulf is `R/4.3.0` as of May 2023. R is available on Biowulf via environment modules *(https://hpc.nih.gov/apps/modules.html)*.

To see the available modules use:

```
module -r avail '^R$'
```

Here we using the `module` command with the option to use regular expression matching (`-r`) and `avail` to return a list of available modules.

Before loading the R module and running R, we first need an interactive session. **R cannot be used on the login node or on helix.**

```
sinteractive --gres=lscratch:5
```

sinteractive default allocations

The default sinteractive allocation is 1 core (2 CPUs) and 0.768 GB/CPU (1.536 GB but rounded to 2 GB in the terminal) of memory and a walltime of 8 hours.

Note: lscratch

"R will automatically use lscratch for temporary files if it has been allocated" (HPC Biowulf docs). `lscratch` space can be requested using `--gres=lscratch:#`, where `gres` stands for "generic resources" and `#` is the number of GB you would like allocated. This will be code dependent.

Info: more memory and CPUs?

> You may want to also include more memory and more CPUs (for multi-threaded) (e.g., `sinteractive --cpus-per-task=2 --mem=6g --gres=lscratch:20`). However, often more memory is not needed and most R code is single threaded, unless written specifically to be multi-threaded. Track memory and CPU usage using `jobload` *(https://hpc.nih.gov/docs/biowulf_tools.html#jobload)* or the user dashboard *(https://hpcnihapps.cit.nih.gov/auth/dashboard/)*.

Loading modules

Load the R module and begin the R session.

```
# Load the module
module load R/4.2.2
# Begin the R session
R
```

Setting up local libraries

Each version of R loaded as a module includes a number of installed packages *(https://hpc.nih.gov/apps/R.html#packages)*. However, you may want to install additional packages, which will by default be stored in "`~/R/%v/library` where `%v` is the major.minor version of R (e.g. 4.2)".

Due to the space constraints associated with biowulf home directories (16GB), it is safer to save installed packages to `/data/$USER`.

First, make a new package directory.

```
#replace %v with the major.minor version of R you plan to use (e.g.,
mkdir -p /data/$USER/R/%v
```

Next, set this location using `$R_LIBS_USER` in your `~/.bashrc` file.

```
nano ~/.bashrc
```

Copy and paste `export R_LIBS_USER="/data/$USER/R/%v"` to the file. Replace `$USER` with your username and `%v` with the correct version number. Use `Ctrl + O` to write the file, press `return`, and `Ctrl + X` to exit.

Open R and check your library path.

```
R
.libPaths()
```

You should see the new path to your personal library listed first followed by the library established my `module load R`.

Let's quit R and end the interactive session.

```
q() # quit R
exit # end interactive session
```

Next time

Lesson 3 will feature R project management and using `renv` to manage package dependencies.

Lesson 3: R Project Management and renv

Learning objectives

1. Discuss the importance of reproducibility
2. Learn ways to make R analyses more reproducible
3. Learn how to set up and organize an R project
4. Learn how to use `renv` for R package management

What is the 2023 NIH Data Management and Sharing Policy?

Effective January 25, 2023, the NIH released the 2023 NIH Data Management and Sharing Policy *(https://oir.nih.gov/sourcebook/intramural-program-oversight/intramural-data-sharing/2023-nih-data-management-sharing-policy)*. This policy requires that NIH intramural researchers plan for data management and sharing prior to conducting scientific research. To do this, scientists are required to submit a Data Management and Sharing plan and comply with the approved plan. While the policy highlights types of data that should be managed and shared and provides links to further resources *(https://sharing.nih.gov)*, it does not provide any guidance on the management and sharing of code needed to truly replicate an analysis.

Sharing data and reporting on analysis steps is not enough to reproduce scientific results.

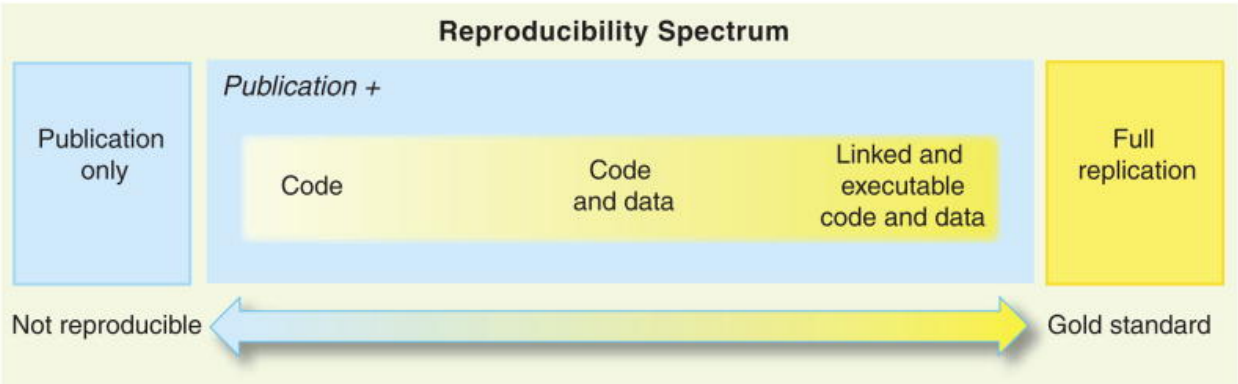

Figure from Peng 2012, *Science*, doi: 10.1126/science.1213847 *(doi: 10.1126/science.1213847)*.

On the reproducibility spectrum, we should strive for "Full replication". Ultimately, this includes making an analysis executable with a fully functioning computational environment. We aren't going to get that far today. However, we will discuss some ways to organize data, code, and package dependencies to improve data analysis sharing and collaboration.

How can we make our R analyses more reproducible?

There are many ways to increase collaboration and document and share code and results using R.

Some examples include:

1. RMarkdown / Quarto (i.e., literate programming)

   > R Markdown provides an unified authoring framework for data science, combining your code, its results, and your prose commentary. R Markdown documents are fully reproducible and support dozens of output formats, like PDFs, Word files, slideshows, and more. --- R4DS *(https://r4ds.had.co.nz/r-markdown.html#r-markdown)*

   Note

   `Quarto` is the next generation of R Markdown with new and enhanced features.

   RMarkdown and Quarto can be used to communicate analysis steps and results. They can specifically be used to:

   1. Create a data science lab notebook
   2. Share and report results to collaborators and others via specific output formats (e.g., html, pdf, etc.)
   3. Create a dashboard of results (via flexdashboard *(https://pkgs.rstudio.com/flexdashboard/)*)

   Tip

   Always include a code chunk calling `sessionInfo()` at the end of your RMarkdown file. This will yield crucial information about your R session, including your operating system requirements, R version, and package versions.

2. R Project

   RStudio projects are self-contained project directories that include the data, code, outputs, and other related files to reproduce an analysis. When you use relative file paths (relative to the project directory), it is fairly easy to reproduce any results within the project.

   We are going to leverage the benefits of an R project to enhance reproducibility.

3. Version Control (e.g. Git) (**Recommended**)

Version control is a great way to enhance data management and collaboration. When you use version control, you can easily track changes that you make to your code and eliminate the need for multiple copies of a script (e.g., Final, Finalv2, Final_final, etc.). Version control is easy to use with R packages and R projects.

> **Info**
>
> For more information on using Git with R, check out https://happygitwithr.com/index.html *(https://happygitwithr.com/index.html)* and https://raps-with-r.dev/git.html *(https://raps-with-r.dev/git.html)*.

4. R package

   R packages are loadable extensions that contain code, data, documentation, and tests in a standardized shareable format that can easily be installed by R users. While the primary repository for R packages is CRAN, you can also readily distribute R packages directly from GitHub.

   Here are some resources if interested in bundling your analysis in an R package:

   1. Put your Data Analysis in an R Package — Even if You Don't Publish it *(https://towardsdatascience.com/put-your-data-analysis-in-an-r-package-even-if-you-dont-publish-it-64f2bb8fd791)*
   2. How to turn your analysis project into a stand alone R package *(https://vimeo.com/427243128)*
   3. The fusen package: inflates a Rmarkdown file to magically create a package *(https://thinkr-open.github.io/fusen/)*

5. Containerization

   Cointanerizing a computational environment using **Docker** or **Singularity** freezes the computational environment, including operating system, so that results are truly reproducible.

> **Other tips for reproducible programming**
>
> 1. Incorporate functional programming
>   1. Elminate repetitive code with well-written functions, making code easier to test, document, and share.
>   2. Make code as independent from the global environment as possible.
> 2. Use literate programming
>   1. Rmarkdown, quarto, etc. to generate parameterized reports.

Today we will focus on organizing our data, code, documents in an R project.

R Project Management and `renv`

R projects allow us to easily share data, code, and other related information, but this only scratches the surface of what is required for true data analysis reproducibility. We won't take all steps to make our project reproducible today, but beyond basic project organization, it is fairly easy to document and manage package dependencies.

Too often an R script will fail simply due to a clash in package dependencies. Versions are important. R versions change over time; Bioconductor versions evolve, and R packages change. While we can include the `sessionInfo()` at the end of a script or markdown file, this in no way facilitates our ability to truly replicate the infrastructure surrounding our code. **Thankfully, there are R packages available that help us do just that.**

Check out this chapter from R 4 Data Science *(https://r4ds.had.co.nz/workflow-projects.html)*.

Introducing renv (reproducible environments)

> The renv package is a new effort to bring project-local R dependency management to your projects. The goal is for renv to be a robust, stable replacement for the Packrat package, with fewer surprises and better default behaviors.
>
> Underlying the philosophy of renv is that any of your existing workflows should just work as they did before – renv helps manage library paths (and other project-specific state) to help isolate your project's R dependencies, and the existing tools you've used for managing R packages (e.g. install.packages(), remove.packages()) should work as they did before.--- renv *(https://rstudio.github.io/renv/articles/renv.html)*

In a nut shell, `renv` will allow us to recreate our `sessionInfo()`. However, it is not perfect, and does require extra storage due to the creation of a per project library.

Note

`renv` does not manage R versions. You will need to make sure you are using an appropriate version of R to recreate an R project library. Because Biowulf uses module environments for R installations, this isn't a huge hurdle.

Main functions

Creates a local library of R packages copying what you used from your project.

The primary functions and workflow is as follows:

1. `renv::init()` initialize the project to be used with renv and creates a project library

   This is only required once. Once initialized, you work in the project as normal.

`renv::init()` will detect package dependencies based on `library()` and `require()` in R scripts found in the R project.

> **Note**
>
> You can intialize a project without dependency discovery and installation using `renv::init(bare=TRUE)`.

2. `renv::snapshot()` updates renv.lock file *(https://rstudio.github.io/renv/articles/lockfile.html)*, saving the state of the project library.
3. `renv::restore()` restores the state of R environment to replicate what is in lock file.

Getting Started: Setting up our R Project

Connect to Biowulf, obtain an interactive session, load R

Let's connect remotely to Biowulf.

```
ssh username@biowulf.nih.gov
```

Enter your password and hit `enter`.

Navigate to your `/data/$USER` directory.

```
cd /data/$USER
```

Get an interactive session.

```
sinteractive --gres=lscratch:5
```

Let's make a class directory.

```
mkdir R_on_Biowulf
cd R_on_Biowulf
```

Load R version 4.2.2.

```
module load R/4.2.2
```

> **Note**

R/4.3.0 became the default R installation as of May 2023.

Set up an R project

Things to consider:

- R Projects are generally created with intent to use with RStudio; you do not need to create an "R project" to organize a project directory.
- When creating a project directory:
  - Create a consistent directory structure with the top level as the project directory
  - All inputs and outputs (where possible) should be contained within a project directory
  - "never use absolute paths in your scripts, because they hinder sharing: no one else will have exactly the same directory configuration as you" R4ds *(https://r4ds.had.co.nz/workflow-projects.html)*

We will not be using an IDE but we will create an R project using the R package `usethis` *(https://usethis.r-lib.org/index.html)*, which is accessible via `devtools`. `usethis` is a "package that facilitates interactive workflows for R project creation and development" *(https://www.tidyverse.org/blog/2020/12/usethis-2-0-0/)*.

Create the R project

```
#open R
R
#create project
usethis::create_project(path = "MyNewProject", open = TRUE, rstudio =
#quit R
q()
```

```
v Creating 'MyNewProject/'
v Setting active project to '/vf/users/emmonsal/R_on_Biowulf/MyNewPr
v Creating 'R/'
v Writing a sentinel file '.here'
* Build robust paths within your project via `here::here()`
* Learn more at <https://here.r-lib.org>
```

When prompted:

```
Save workspace image? [y/n/c]: n
```

The arguments `open = TRUE` activates the new project and establishes a new working directory. `rstudio = FALSE` establishes a `.here` *(https://here.r-lib.org/)* file that allows the project directory to be recognized as the top level of a project.

```
ls
```

Now, we will see our new directory `MyNewProject`. Let's copy our R scripts to our new project directory.

```
cd MyNewProject
cp /data/classes/BTEP/R_on_Biowulf_2023/scripts/*.R ./R
```

Initialize and activate renv in the project

Cache directory set-up

First, let's set up our `renv` cache location. `renv` uses a global cache to reduce duplicate installs of packages across projects.

> When using renv with the global package cache, the project library is instead formed as a directory of symlinks (or, on Windows, junction points) into the renv global package cache. --- renv *(https://rstudio.github.io/renv/articles/renv.html#cache)*

By default the `renv` cache will be created in your home directory, which can quickly fill up if using Bioconductor packages. We are going to instead create a cache in our `/data/$USER` directory.

```
mkdir -p /data/$USER/.cache/R/renv
```

Create a .Renviron file *(https://support.posit.co/hc/en-us/articles/360047157094-Managing-R-with-Rprofile-Renviron-Rprofile-site-Renviron-site-rsession-conf-and-repos-conf)* within your home directory.

```
nano ~/.Renviron
```

Add the following line:
`RENV_PATHS_ROOT=/data/$USER/.cache/R/renv`
Replace `$USER` with your actual username.

Use `ctrl+O` to save, `return`, and `ctrl+X` to exit.

> **Important**
>
> The renv cache only needs to be set up once regardless of the version of R you are using as long as you created a user level .Renviron file establishing its location.

Run renv::init()

Once we have done this, we can activate renv within our project. But, first, let's verify the location of our renv cache.

```
R
renv::paths$cache() # Check the cache location
renv::init(bioconductor = "3.16") #initialize renv in the project
```

Info

We can initialize renv with a specific version of Bioconductor. This eliminates later headaches as Bioconductor updates to newer versions. See here *(https://rstudio.github.io/renv/articles/bioconductor.html)* for more information.

Warning

This step takes about 5-10 minutes.

Now that we have initialized `renv` with this project. Let's check our R library paths.

```
.libPaths()
```

You should see the `renv` project library listed first, meaning it is prioritized over the module (site) libraries.

```
[1] "/vf/users/$USER/R_on_Biowulf/MyNewProject/renv/library/R-4.2/x8(
[2] "/usr/local/apps/R/4.2/site-library_4.2.2"
[3] "/usr/local/apps/R/4.2/4.2.2/lib64/R/library"
```

The library snapshot resulted in an error due to `GenomeInfoDb [installed 1.35.14 != latest 1.34.9]`.

Let's update the installation of `GenomeInfoDb` as suggested by the prompt.

> Packages from Bioconductor can be installed by using the bioc:: prefix. ---renv vignette *(https://rstudio.github.io/renv/reference/install.html)*

```
renv::install("bioc::GenomeInfoDb")
```

Note

> Without specifying a version of Bioconductor to be used with a project (e.g., `renv::init(bioconductor = "3.16")`), `install("bioc::GenomeInfoDb")` will attempt to install the latest-available version from Bioconductor (v.3.17).

Call `renv::snapshot()` to save the state of the project library to the lockfile.

```
renv::snapshot()
```

We see a long list of packages being written to the lockfile and the following message:

```
The version of R recorded in the lockfile will be updated:
- R                      [* -> 4.2.2]

Do you want to proceed? [y/N]:
```

Type `y`.

```
* Lockfile written to '/vf/users/$USER/R_on_Biowulf/MyNewProject/renv
```

This was successful and the lockfile was written. The lockfile is necessary to restore the project at a later date.

Establish a consistent project structure

Now that we have `renv` set up with our project, let's also establish a project structure.

Let's exit R and edit our `.Rprofile`.

> **Note**
>
> When we ran `renv::init()` a local `.Rprofile` file was created with the code `source("renv/activate.R")`. This code is necessary "to automatically load and use the private [renv] library for new R sessions launched from the project root directory" (renv *(https://rstudio.github.io/renv/articles/renv.html#workflow)*).

```
q()
Save workspace image? [y/n/c]: n
nano .Rprofile
```

Delete the single line in the .Rprofile file, and paste the following, which was borrowed from a blog on data management *(https://www.r-bloggers.com/2020/02/efficient-data-management-in-r/)*, into the Rproject .Rprofile:

```
.First <- function() {
  dir.create(paste0(getwd(), "/figures"), showWarnings = F)
  dir.create(paste0(getwd(), "/outputs"), showWarnings = F)
  dir.create(paste0(getwd(), "/data"), showWarnings = F)
  dir.create(paste0(getwd(), "/docs"), showWarnings = F)

  if (!("renv" %in% list.files())) {
    renv::init()
  } else {
    source("renv/activate.R")
  }

  cat("\nWelcome to your R-Project:", basename(getwd()), "\n")
}
```

`ctrl + O` to save, `return`, `ctrl+X` to exit.

This code creates several directories (i.e., figures, outputs, data, and docs) and initializes the project for use with the `renv` package using `renv::init` or if already initialized activates the renv project library using `renv/activate.R`. This will not overwrite directories that have already been created.

Note

You can change these directory names to whatever works best with your organization style. The key, however, is to stay as consistent as possible across projects.

Info

Using version control (git via GitHub) is an even better way to manage data and share inputs, code, and results. You can easily manage a Github repository or create a new repository using the `usethis` package. Also, check out this resource *(https://happygitwithr.com/index.html)* for understanding more regarding version control and R.

We will need to start a new R session for the .Rprofile to take effect.

```
R
q()
```

```
Save workspace image? [y/n/c]: n
```

Now we are ready to work with our project files.

Test it

Before we end today's lesson, let's test out `renv`.

We will create a new directory and transfer our R script and lock file. We will then restore the project and run our R script.

```
#change director to /data/$USER
cd ..
#make test directory
mkdir renv_test
#copy files to test directory
cp MyNewProject/renv.lock renv_test/
cp MyNewProject/R/DESeq2_airway.R renv_test/
#change directory to test directory
cd renv_test
#load R module if not already loaded and start R session
module load R/4.2.2
R
#Restore renv library
renv::restore()
```

Now, we can run our script.

```
source("DESeq2_airway.R")
```

Note

The R version used to create a new project with the MyNewProject renv.lock file must be the same.

Also, because the required packages were already in our renv cache, updating the test library was much faster.

Next Lesson

In the final lesson, we will learn more about submitting jobs using R on Biowulf.

Acknowledgements

- Making your analysis portable and reproducible *(https://diytranscriptomics.com/project/lecture-12)* from DIY transcriptomics.
- Efficient Data Management in R *(https://www.r-bloggers.com/2020/02/efficient-data-management-in-r/)*.
- R Packages with renv *(https://sites.google.com/nyu.edu/nyu-hpc/hpc-systems/greene/software/r-packages-with-renv)*

Lesson 4: Submitting R Scripts via command line

Learning Objectives

1. Learn how to use R with less interaction
2. Learn how to deploy `sbatch` R jobs, and learn about alternatives such as `swarm`.
3. Learn about R job parallelization in the context of Biowulf

We have organized our R project directory and have set up `renv` to make our R environment a bit more reproducible. Now, we need to learn how to submit an R script. Thus far, we have been using R interactively by first obtaining an interactive compute node (`sinteractive`). However, we can submit R scripts without interaction using `sbatch` and `swarm`. This is advantageous as we may want to include our R Script in a pipeline or process thousands of files.

> Running R scripts from the command line can be a powerful way to:
>
> - Automate your R scripts
> - Integrate R into production
> - Call R through other tools or systems--- Nathan Stephens, Posit Support *(https://support.posit.co/hc/en-us/articles/218012917-How-to-run-R-scripts-from-the-command-line)*

Example scripts

We will use a couple of example scripts in this section (`DESeq2_airway.R`, `Volcano.R`). The first script uses the R package `airway`, which contains data from Himes et al. 2014 *(https://pubmed.ncbi.nlm.nih.gov/24926665/)*, a bulk RNA-Seq study, as a Ranged SummarizedExperiment. The Bioconductor package `DESeq2` is then used to produce differential expression results. This R script largely follows a Bioconductor workflow on RNA-seq *(https://bioconductor.org/packages/release/workflows/vignettes/rnaseqGene/inst/doc/rnaseqGene.html)*. The second script (`Volcano.R`) takes output from the first script and makes a volcano plot using the package `EnhancedVolcano`.

Warning

These scripts are for example only. You should not use them to apply to your own data.

Running R from command line

Before jumping into submitting scripts in job files, let's first focus on how to run R from the command line.

The primary way to run R from the command line is to call `Rscript`. `Rscript` is a binary front-end to R, to use for scripting applications; basically, it is a convenience function.

Let's see this in action first in an interactive session:

```
sinteractive --gres=lscratch:5
module load R/4.2.2
```

Let's use our `renv_test` directory to see how this works. The syntax is `Rscript [options] file [args]` .

```
cd /data/$USER/R_on_Biowulf/renv_test
Rscript DESeq2_airway.R > DESeq2_airway.out
```

The default is `--no-echo`, which makes R run as quietly as possible, and `--no-restore`, which indicates that we do not want anything restored (e.g., objects, history, etc.) also imply `--no-save`, meaning the workspace will not be saved. Here, we have no additional `options` or `args`.

As a convenience function, `Rscript` is the same as calling

```
R --no-echo --no-restore --no-save --file=DESeq2_airway.R  > DESeq2_a
```

Note

We have been using > to direct stdout to a file. We can also use < to direct the input file. See below.

```
R --no-echo --no-restore --no-save < DESeq2_airway.R > DESeq2_airway3.out
```

Info: Rscript --help

You can learn more about `Rscript` using `Rscript --help` and `R --help`. Notice from `R --help` that you can also use `R CMD BATCH` to run an R script from command line. To run a script from the R console, use `source()`.

Saving R output

Notice that we can easily save R output directed to standard output using >. However, this will exclude messages, warnings, and errors, which are directed to standard error. For stdout you can specify `1> or >`; for stderr you can specify `2>`, and for both in a single file you can specify `&>`. See here *(https://www.r-bloggers.com/2020/04/where-does-the-output-of-rscript-go/)* and here *(https://tldp.org/LDP/abs/html/io-redirection.html)* for more information.

Adding command line arguments

R scripts can be run from the command line with command line arguments. Here *(http://swcarpentry.github.io/r-novice-inflammation/05-cmdline.html)* is a great resource from software carpentry explaining command line arguments.

To use command line arguments with an R script, we leverage `commandArgs()`. This function creates a vector of command line arguments. When using `trailingOnly = TRUE`, `commandArgs()` only returns arguments after `R -no-echo --no-restore --file --args`.

Let's see how this works in a simple script that returns a volcano plot of our differential expression results. First, let's copy over the `Volcano.R` script to our test directory, `renv_test`.

```
cp /data/classes/BTEP/R_on_Biowulf_2023/scripts/Volcano.R .
```

The contents of `Volcano.R`:

```
# Create a Volcano Plot from DESeq2 differential expression results ·
library(EnhancedVolcano)
library(dplyr)

## set command line arguments ----
args <- commandArgs(trailingOnly = TRUE)

#stop the script if no command line argument
if(length(args)==0){
  print("Please include differential expression results!")
  stop("Requires command line argument.")
}

## Read in data ----
data<-read.csv(args[1],row.names=1) %>% filter(!is.na(padj))

labs<-head(row.names(data),5)

## Plot ----
EnhancedVolcano(data,
```

```
                title = "Enhanced Volcano with Airways",
                lab = rownames(data),
                selectLab=labs,
                labSize=3,
                drawConnectors = TRUE,
                x = 'log2FoldChange',
                y = 'padj')


ggsave("./figures/Volcano.png",width=5.5,height=3.5,units="in",dpi=3(
```

This script requires a single argument, a .csv file containing our differential expression results.

How can we run this from the command line?

```
Rscript Volcano.R ./outputs/deseq2_DEGs.csv
```

The easiest way to checkout the output of this function (`Volcano.png`) is to mount our HPC system directories locally *(https://hpc.nih.gov/docs/helixdrive.html)*.

Info: Packages used to parse command-line arguments

There are also several packages that can be used to parse command-line arguments such as `getopt` *(https://github.com/trevorld/r-getopt)*, `optparse` *(https://github.com/trevorld/r-optparse)* ,`optigrab` *(https://github.com/cran/optigrab)*, `argparse` *(https://github.com/trevorld/r-argparse)*, `docopt` *(https://github.com/docopt/docopt.R)*, `GetoptLong` *(https://github.com/jokergoo/GetoptLong)*.

Rendering Rmarkdown files from command line

In addition to R scripts, we can render Rmarkdown files directly from the command line by adding an R expression (an object that represents an action that can be performed by R *(https://adv-r.hadley.nz/expressions.html)*) directly to our `Rscript` command using the `-e` expression flag.

```
cp /data/classes/BTEP/R_on_Biowulf_2023/rmarkdown/Volcano.Rmd .
cp ./outputs/deseq2_DEGs.csv DEGs.csv

Rscript -e "rmarkdown::render('Volcano.Rmd',params=list(args = 'DEGs
```

To make this work, parameters had to be added to the yaml of the Rmarkdown.

Using sbatch

> R batch jobs are similar to any other batch job. A batch script ('rjob.sh') is created that sets up the environment and runs the R code. --- R/Bioconductor on Biowulf *(https://hpc.nih.gov/apps/R.html#sbatch)*

Default allocations for an `sbatch` job include:

2 CPUs with a default memory per CPU of 2 GB. Therefore, the default memory allocation is 4 GB.

More about `sbatch`

`sbatch` is used to submit batch jobs, which are resource provisions that run applications on compute nodes and do not require supervision or interaction *(https://curc.readthedocs.io/en/stable/running-jobs/batch-jobs.html)*. To submit a batch job, a job script containing a list of unix commands to be executed by the job is typically required. This script may also include resource requirements (job directives) telling the job scheduler what types of resources are needed for the job. While bash shell scripting is typically used to write these files. Other shells can also be used.

Features of job scripts:

- if using a bash shell, the file typically ends in `.sh`.
- File content starts with a shebang (`#!`) followed by the path to the interpreter (`/bin/bash`) on the first line.
- Content may include SLURM job directives denoted by `#SBATCH` at the beginning of the script directly following `#!/bin/bash`. These can provide information to the Biowulf batch system such as:
    - Partition (default = "norm", `--partition`)
    - Name of the job (`--job-name`)
    - What types of job status notifications to send (`--mail-type`)
    - Where to send job status notification (`--mail-user`)
    - Memory to allocate (`--mem`)
    - Time to allocate (`--time`)
    - cpus per tasks (# of threads if multithreaded) (`--cpus-per-task`)
- Following `#SBATCH` directives, you can include comments throughout your list of commands using #.

See important sbatch flags here *(https://hpc.nih.gov/docs/userguide.html#submit)* and complete options with `sbatch --help`.

Submitting the R script as a job using `sbatch`.

We will create and submit a job script using `sbatch` that will run the R scripts in the project we created in Lesson 3 (`MyNewProject`).

Example job script:

```
nano rjob.sh
```

Paste the following:

```
#!/bin/bash
#SBATCH --gres=lscratch:5
#SBATCH --mail-type=BEGIN,END

#Load the R module
module load R/4.2.2

#change to project directory
cd /data/$USER/R_on_Biowulf/MyNewProject

#Run R scripts using Rscript
Rscript ./R/DESeq2_airway.R
Rscript ./R/Volcano.R ./outputs/deseq2_DEGs.csv
```

`Ctrl+O`, `return`, `Ctrl+X`

The R script should be run in the project directory (`MyNewProject`) to take advantage of `renv`.

We included the job directives `--gres=lscratch:5` and `--mail-type=BEGIN,END`. `--gres=lscratch:5` ensures that we have 5 GB of lscratch space for temporary storage. `--mail-type=BEGIN,END` directs the job scheduler to send us an email when the job starts and ends. This email will by default go to your NIH email.

Note: stdout & stderr

For an sbatch job, a stdout and stderr file is automatically generated (by default, slurm######.out in the submitting directory). This can be modified using the following sbatch flags / directives (`--output=/path/to/dir/filename`, `--error=/path/to/dir/filename`).

Note: command line flags vs directives

You can also include job flags at the time of job submission. If these conflict with #SBATCH directives, the command line flags take priority.

Let's submit the script.

```
sbatch rjob.sh
```

This job script can be submitted from any location as long as the path to the script (`rjob.sh`) is correct.

Using swarm

Swarm *(https://hpc.nih.gov/apps/swarm.html)* is a way to submit multiple commands to the Biowulf batch system and each command will be run as an independent job with identical resources, allowing for parallelization.

- Swarm scripts have the extension *.swarm.
- Lines that start with #SWARM are not run as a part of the script but these are directives that tells the Biowulf batch system what resoures (ie. memory, time, temporary storage, modules) are needed.

See here *(https://hpc.nih.gov/apps/R.html#swarm)* for submitting R swarm jobs.

Rswarm

There is also a utility `Rswarm` that may interest you in specific cases.

> Rswarm is a utility to create a series of R input files from a single R (master) template file with different output filenames and with unique random number generator seeds. It will simultaneously create a swarm command file that can be used to submit the swarm of R jobs. Rswarm was originally developed by Lori Dodd and Trevor Reeve with modifications by the Biowulf staff.

`Rswarm` is great for simulations; see an example use case of rswarm here *(https://hpc.nih.gov/apps/R.html#rswarm)*.

Parallelizing code

Can you speed up your code with parallelization?

Considerations:

- levels of parallelization: multiprocessing vs multithreads

> The most common form of parallelism in R is multiprocessing. This is usually explicitly done by you or package you are using. There are are some parts of base R and the underlying math libraries that can multithread which is mostly

> implicit parallelism. You can check if your code can take advantage of that. You can allocate for example 4 CPUs and then run your script with different settings of the `$OMP_NUM_THREADS` or `$MKL_NUM_THREADS` environment variable. If you see a significant speed up and the dashboard data shows that it used multiple CPUs then it's worth using more than one CPU.
>
> It is important to always test parallel efficiency and monitor actual usage of CPUs and memory with the dashboard *(https://hpc.nih.gov/dashboard)* or using the dashboard_cli command. For running jobs there is also `jobload`. --- R on Biowulf, NIH HPC Team *(https://hpc.nih.gov/training/handouts/R_on_Biowulf.pdf)*

- Can the job be split into multiple independent processes? If yes, consider an R swarm job *(https://hpc.nih.gov/apps/R.html#swarm)*.
- Are there functions in the code that support multiple threads? If so, you can take advantage of multi-threading.
- Is there a `lapply/sapply` function? Consider replacing with `mclapply` *(https://www.rdocumentation.org/packages/parallel/versions/3.4.0/topics/mclapply)*.
- Is there 'for' loop? Consider using `foreach` *(https://cran.r-project.org/web/packages/foreach/vignettes/foreach.html)* for parallel execution.

You may find this resource *(https://bookdown.org/rdpeng/rprogdatascience/parallel-computation.html)* on parallelizing R code, helpful.

However, see tips from the NIH HPC R/Bioconductor documentation for specific considerations on:
1. Using the parallel package *(https://hpc.nih.gov/apps/R.html#parallel)*
2. Using the BiocParallel package *(https://hpc.nih.gov/apps/R.html#parallel)*
3. Implicit multi-threading *(https://hpc.nih.gov/apps/R.html#threading)*

> **Info: Pitfalls around parallelizing R Code**
>
> Some R packages will detect all cores on a node even if they are not allocated (e.g. `parallel::detectCores()`). You should use `parallelly::availableCores()` to detect allocated CPUs. --- R on Biowulf, HPC Team *(https://hpc.nih.gov/training/handouts/R_on_Biowulf.pdf)*

See specific examples regarding parallelization and troubleshooting in the NIH HPC training R on Biowulf *(https://hpc.nih.gov/training/handouts/R_on_Biowulf.pdf)*.

Need help running your R code on Biowulf?

If you experience difficulties with running R on Biowulf, you should:

1. Read the R docs on Biowulf *(https://hpc.nih.gov/apps/R.html)*.
2. Contact the HPC team at staff@hpc.nih.gov *(mailto:staff@hpc.nih.gov)*

3. Attend monthly HPC walk-in virtual consultations *(https://hpc.nih.gov/training/)*

Also, please feel free to email us at ncibtep@nih.gov *(mailto:ncibtep@nih.gov)*

Additional Resources

Additional Resources

HPC Biowulf Resources

1. Biowulf User Guide *(https://hpc.nih.gov/docs/userguide.html)*
2. R/Bioconductor on Biowulf *(https://hpc.nih.gov/apps/R.html)*
3. R on Biowulf Training by the HPC team *(https://hpc.nih.gov/training/handouts/R_on_Biowulf.pdf)*

Other Resources

1. What they forgot to teach you about R *(https://rstats.wtf/)*
2. `usethis` package reference *(https://usethis.r-lib.org/articles/usethis-setup.html)*
3. Controlling R Startup with .Rprofile, .Renviron, etc. *(https://support.posit.co/hc/en-us/articles/360047157094-Managing-R-with-Rprofile-Renviron-Rprofile-site-Renviron-site-rsession-conf-and-repos-conf)*
4. Workflow:projects from R 4 Data Science *(https://r4ds.had.co.nz/workflow-projects.html)*
5. Introduction to renv *(https://rstudio.github.io/renv/articles/renv.html)*
6. Happy Git and GitHub for the useR *(https://happygitwithr.com/index.html)*
7. Building reproducible analytical pipelines with R *(https://raps-with-r.dev/)*
8. RStudio Background jobs *(https://docs.posit.co/ide/user/ide/guide/tools/jobs.html)*