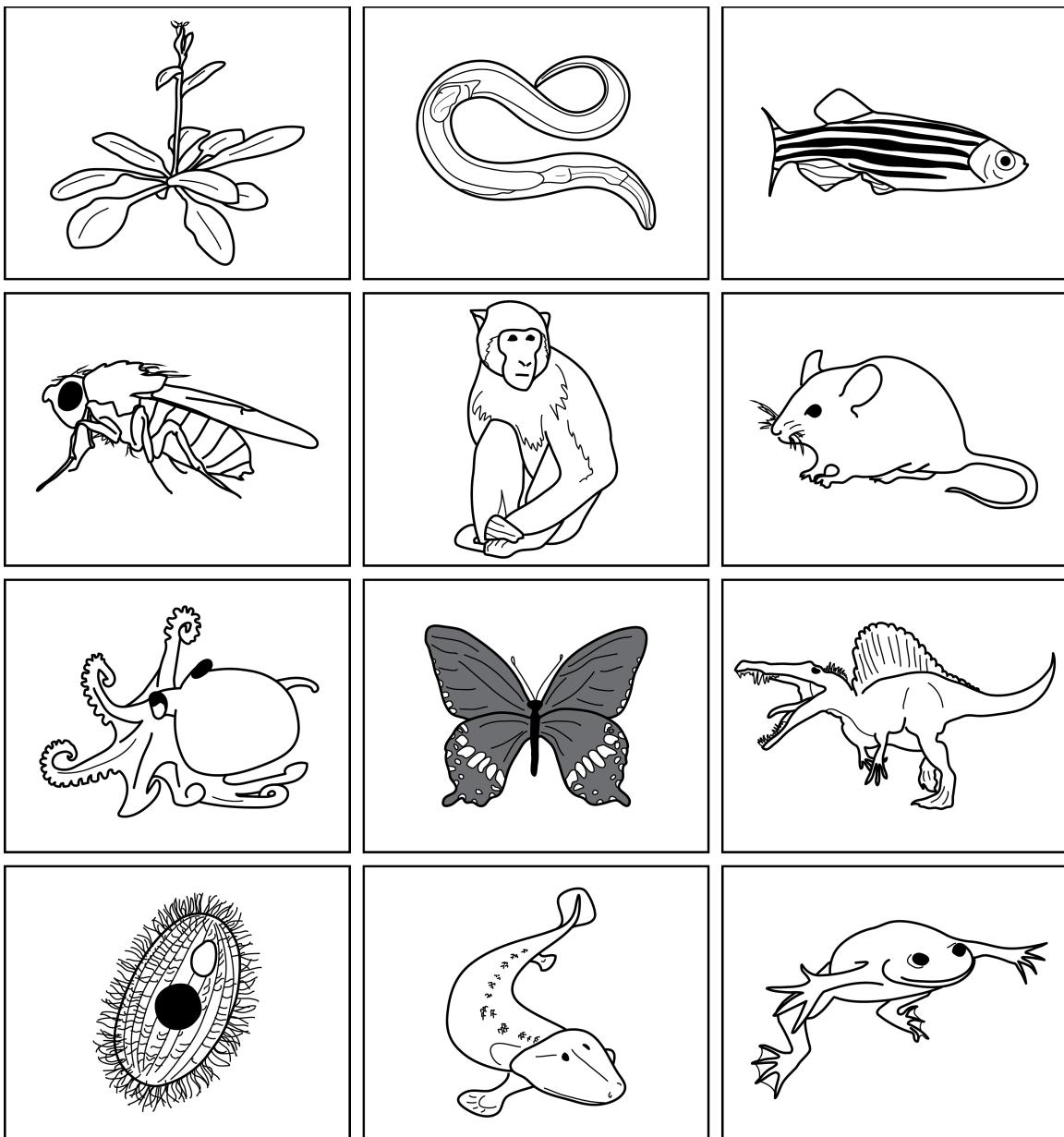


# BSD Quantitative Biology Boot Camp



*September 6 – 14, 2016  
Marine Biological Laboratories  
Woods Hole, MA*



# MBL Village Campus



- |           |  |
|-----------|--|
| <b>1</b>  | Candle House (Administration)                                      |
| <b>2</b>  | 100 Water Street (Pierce Exhibit Center, Satellite Club, MBL Club) |
| <b>3</b>  | Marine Resources Center  |
| <b>4</b>  | Collection Support Facility  |
| <b>5</b>  | Crane Wing (Labs, Shipping)  |
| <b>6</b>  | Lillie Laboratory (Labs, Service Shops, MBLWHOI Library)           |
| <b>7</b>  | Rowe Laboratory (Labs, Speck Auditorium)                           |
| <b>8</b>  | Environmental Sciences Laboratory                                  |
| <b>8a</b> | Homestead Administration (Human Resources, Education Dept.)        |
| <b>9</b>  | Loeb Laboratory (Research and Teaching Labs, Lecture Rooms)        |
| <b>10</b> | Brick Apartment House  |
| <b>11</b> | Veeder House Dormitory   |
| <b>12</b> | David House Dormitory  |
| <b>13</b> | Broderick House (IT)   |
| <b>14</b> | Crane House  |
| <b>15</b> | Swope Center (Registration, Cafeteria, Dormitory, Meigs Room)      |
| <b>16</b> | Ebert Hall Dormitory   |
| <b>17</b> | Drew House Dormitory   |
| <b>18</b> | 15 North Street (Carpentry Shop)                                   |
| <b>19</b> | Smith Cottage and Barnecker Road Property                          |
| <b>20</b> | C.V. Starr Environmental Sciences Laboratory                       |
| <b>21</b> | 11 North Street  |
|           | MBL Parking  |

## INFORMATION SHEET

**MBL**



<http://www.mbl.edu/>

### EMERGENCY

**Call boxes** can be found in the Brick Apartment Building, and outside of Swope.

**From cell:** 508-289-7911 calls MBL security who will respond and notify Falmouth police or fire. Put this on the favorites list of your cell while you are here

**MBL security** non-emergency: 508-289-7217

**AIRPORTS** Logan International Airport [www.massport.com/logan-airport](http://www.massport.com/logan-airport) Boston, MA.) or TF Green Airport [www.pvdairport.com](http://www.pvdairport.com) (Providence, RI.)

**ALCOHOL** No one under the age of 21 is allowed to consume alcoholic beverages. This is strictly enforced.

### ATM MACHINE

22 Luscombe Ave  
Woods Hole, MA. OR Bank of Woods Hole, 2 Water Street, Woods Hole, MA. There is also an ATM next to the Woods Hole Market.

**BADGES and LANYARDS** ID Cards and *MBL* lanyards are for registered guests and employees on the MBL campus. They must be worn at all times for identification purposes, entry into buildings and to attend meals. Please swipe them slowly as you are entering the building as the machines are sensitive.

**BIKE PATH** Shining Sea Bikeway is 10.7 miles which hugs the coast from Falmouth to Woods Hole and then beyond to North Falmouth. It is ideal for casual biking, which passes through Sippewissett Marsh, Cranberry bogs and overlooks Chapoquoit Beach crisscrossing through the Salt Pond Bird Sanctuary. For bike rental information try Corner Cycle Cape Cod (508-540-4195)

<http://www.cornercycle.com/about/rental-info-pg60.htm>. An indoor bicycle storage area is available in the basement of Ebert Hall for a nominal fee. Bicycles may not be stored in dormitories or laboratory buildings. For more information about bikes visit <http://www.mbl.edu/parking-program/>

**BIOLOGICAL SAFETY:** All needles, syringes, razor blades should be placed in "sharp" boxes prior to disposal. The safety department supplies these boxes to your lab x 7641 Safety Office.

**CAMPUS MAPS** Copies of the campus map are provided within your orientation packet. They can also be found at the Swope Front Desk.

**CHEMICAL SAFETY** safety data sheets (SDSs) can be found on the website:  
<http://www.mbl.edu/services/safety/> . Please contact x 7424 for further assistance.

**FALMOUTH TAXI** Falmouth Taxi may be contacted should you need transportation outside the MBL. They are often located at the ferry terminal as well. Please call 508-548-3100 for service.

**FIRST AID KITS** are available in hallways adjacent to emergency showers in Loeb classrooms and MBL Laboratories. AEDs are found at the entrance to most of the main buildings.

**FOOD ALLERGIES** On your registration form or program application and during orientation you will have opportunities to notify the MBL about food allergies. It is also recommended to inform your servers in the dining hall so that proper accommodations can be made. If severe allergies exist, please set up a meeting with our dining services by contacting Erin Hummetoglu at 508-457-9084.

**GIFT SHOP HOURS** - The MBL gift shop is open in July-August 10:00 am - 4:30 pm, MWF. Hours vary in other months. See <http://giftshop.mbl.edu>

**HISTORY OF MBL** The Marine Biological Laboratory (MBL) history project seeks to document the wonderfully rich history of the MBL. Historians from across America (<http://history.archives.mbl.edu/info/about#team>) are pouring over the archive to bring us more information on the significant contributions that the people of the MBL have made to scientific discovery See <http://history.archives.mbl.edu>

**LAUNDRY** Laundry facilities are located in Swope and in the basement of Ebert Hall and the Brick Apartment dorms. Those staying in the cottages may use the Pilot House in Devil's Lane, access is your cottage key. All laundry facilities have credit card-operated washers and dryers

**LOCAL TRANSPORTATION** Options for local transportation to and from the Marine Biological Laboratory can use

**Peter Pan Bus** [peterpanbus.com](http://peterpanbus.com), and the schedule

<http://peterpanbus.com/commuter/commuter-bus-schedule-Woods-Hole-Boston.pdf>

**Green Shuttle** [gogreenshuttle.com/#](http://gogreenshuttle.com/#) or, **White Tie Limo** [whitetie.imadevsrv.com](http://whitetie.imadevsrv.com) or Falmouth Taxi is available 24hrs a day and located at the ferry terminal - call 508-548-3100

**LUGGAGE** Upon check out (10am) there is a small hut outside Swope for your luggage. Please do not bring luggage with you into classrooms, labs or other indoor areas.

**MAIL** Please register with the MBL Mail Room upon arrival located in the basement of Lillie. Your address will be (Your Name) Marine Biological Laboratory, 7 MBL Street, Woods Hole, MA 02543.

**MBL ALERT** [emergency.mbl.edu/mblalert](http://emergency.mbl.edu/mblalert)

a high speed, automated emergency notification system to all registered scientists, staff, visitors and students conveying vital safety and security information quickly and accurately.

**MBL LIBRARY** The MBL library is located in the Lillie Building, on the 2nd floor. It is staffed and open to the public 8 am-5pm Monday-Friday. The Library is closed from 11 pm - 7 am everyday. See: <http://www.mblwholibRARY.org> to locate journal titles, databases, and the library catalog. Call: 508-289-7002.

**NOISE ORDINANCE:** Loud noises/music/ conversation that carry more than 150 feet are prohibited after 10 pm. This will be enforced by MBL security.

**PARKING** Parking passes must be obtained from the Swope Check-In desk. Parking is off campus in Devil's Lane Lot. Special permission for passes to Lillie, Loeb, and Swope lots must be obtained through [education@mbl.edu](mailto:education@mbl.edu) . Street parking is metered and cars will be towed from meters between 2 am-7 am. MBL parking lots are available after hours (between 4 pm-7 am) and on weekends.

**PIERCE EXHIBIT CENTER** located at 100 Water Street gives the history and story of the MBL and its impact of scientific discovery. Hours of operation 11am - 4pm (Spring), 10am - 4pm (Summer).

**POISON CONTROL** from campus phone 9-1-800-222-1222 (cell: 800-222-1222).

**PRINTING AND PHOTOCOPYING**

Students have 24/7 access to two photocopying machines located on the 2<sup>nd</sup> floor lobby of the Lillie library. You must use your student ID card to access this building. Printing is functional only from library workstations. Faculty may obtain a copy code from program organizer: 0.10 black and white and 0.30. color copies.

**RECYCLING** The MBL has an active recycling program that includes electronic equipment, paper, packaging material, glass, and plastic. Recycling containers are positioned and recycling notices are posted throughout the campus. Contact Bob Kaski (508-289-7326) for information.

**RESPONSIBLE CONDUCT RESEARCH** (<http://www.mbl.edu/ethics/>) and the MBL Code of Conduct (<http://www.mbl.edu/hr/staff-toolbox/marine-biological-laboratory-policy-manual/code-of-conduct/>)

**SMOKING** The MBL provides a smoke-free environment for all investigators, faculty, students and visitors. As such, smoking is prohibited in the interiors of all MBL facilities including housing, vehicles and all exterior doorways.

**SPORTS EQUIPMENT** Tennis rackets & balls, volleyball, football and other sporting equipment available to sign-out at the Swope front desk.

**STONEY BEACH** Stoney beach is located near the intersection of Millfield street and Albatross (map), a short walk from the MBL campus. Beach hours are 8 am – 10 pm.

**SWOPE DINING HALL HOURS: SUMMER**

Breakfast: 7:00 am - 9:30 am

Lunch: 11:30 am - 1:30 pm

Dinner: 5:00 pm - 7:30 pm

**TECHNOLOGY SUPPORT**

Information regarding common technology questions can generally be found online at [www.mbl.edu/it](http://www.mbl.edu/it). However, if you are unable to find answers to your questions online or if you wish to report a technical problem or request technical assistance please call the IT Help Desk at 508-289-7654 or x7654 from an internal phone, or submit a request to: [helpdesk@mbl.edu](mailto:helpdesk@mbl.edu).

**WIRELESS ACCESS** The *MBL-GUEST* (<https://intranet.mbl.edu/display/IN/MBL-Guest>) wireless network is an easy access encrypted wireless network designed to provide access to the internet. Please refer to wireless access signs posted throughout the campus for additional connectivity information.

# BSD qBio<sup>2</sup> @ MBL

## General Schedule

### Tuesday, September 6

<b>2:45-5:00</b>	Check-in
<b>5:00-6:30</b>	Dinner
<b>8:15-9:00</b>	Introduction – Allesina and Palmer

### Wednesday, September 7

<b>8:30-10:00</b>	Free time ( <i>M.mulatta M.musculus O.bimaculoides P.polytes</i> ) Imaging ( <i>S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Trip on the Gemma ( <i>A.thaliana C.elegans</i> ) Visit Marine Resources Center ( <i>D.melanogaster D.rerio</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Free time ( <i>M.mulatta M.musculus O.bimaculoides P.polytes</i> ) Imaging ( <i>S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Trip on the Gemma ( <i>D.melanogaster D.rerio</i> ) Visit Marine Resources Center ( <i>A.thaliana C.elegans</i> )
<b>12:00-12:45</b>	Lunch
<b>12:45-1:00</b>	MBL Orientation
<b>1:00-2:30</b>	Basic/Advanced computing I ( <i>A.thaliana C.elegans D.melanogaster D.rerio M.mulatta M.musculus</i> ) Math. Foundations I ( <i>O.bimaculoides P.polytes S.aegyptiacus T.roseae T.thermophila X.laevis</i> )
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Basic/Advanced computing I ( <i>A.thaliana C.elegans D.melanogaster D.rerio M.mulatta M.musculus</i> ) Math. Foundations I ( <i>O.bimaculoides P.polytes S.aegyptiacus T.roseae T.thermophila X.laevis</i> )
<b>5:00-6:30</b>	Dinner
<b>7:15-8:00</b>	Welcome – Nishi; Intro to Grad School – Prince

### Thursday, September 8

<b>8:30-10:00</b>	Free time ( <i>S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Imaging ( <i>A.thaliana C.elegans D.melanogaster D.rerio</i> ) Trip on the Gemma ( <i>M.mulatta M.musculus</i> ) Visit Marine Resources Center ( <i>O.bimaculoides P.polytes</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Free time ( <i>S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Imaging ( <i>A.thaliana C.elegans D.melanogaster D.rerio</i> ) Trip on the Gemma ( <i>O.bimaculoides P.polytes</i> )

	Visit Marine Resources Center ( <i>M.mulatta M.musculus</i> )
<b>12:00-1:00</b>	Lunch
<b>1:00-2:30</b>	Basic/Advanced computing I ( <i>O.bimaculoides P.polytes S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Math. Foundations I ( <i>A.thaliana C.elegans D.melanogaster D.rerio M.mulatta M.musculus</i> )
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Basic/Advanced computing I ( <i>O.bimaculoides P.polytes S.aegyptiacus T.roseae T.thermophila X.laevis</i> ) Math. Foundations I ( <i>A.thaliana C.elegans D.melanogaster D.rerio M.mulatta M.musculus</i> )
<b>5:00-6:30</b>	Dinner
<b>7:15-8:00</b>	MBL Talk – Jennifer Morgan

**Friday, September 9**

<b>8:30-10:00</b>	Free time ( <i>A.thaliana C.elegans D.melanogaster D.rerio</i> ) Imaging ( <i>M.mulatta M.musculus O.bimaculoides P.polytes</i> ) Trip on the Gemma ( <i>S.aegyptiacus T.thermophila</i> ) Visit Marine Resources Center ( <i>T.roseae X.laevis</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Free time ( <i>A.thaliana C.elegans D.melanogaster D.rerio</i> ) Imaging ( <i>M.mulatta M.musculus O.bimaculoides P.polytes</i> ) Trip on the Gemma ( <i>T.roseae X.laevis</i> ) Visit Marine Resources Center ( <i>S.aegyptiacus T.thermophila</i> )
<b>12:00-1:00</b>	Lunch
<b>1:00-2:30</b>	Basic/Advanced computing II ( <i>A.thaliana D.rerio M.mulatta O.bimaculoides S.aegyptiacus T.roseae</i> ) Math. Foundations II ( <i>C.elegans D.melanogaster M.musculus P.polytes T.thermophila X.laevis</i> )
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Basic/Advanced computing II ( <i>A.thaliana D.rerio M.mulatta O.bimaculoides S.aegyptiacus T.roseae</i> ) Math. Foundations II ( <i>C.elegans D.melanogaster M.musculus P.polytes T.thermophila X.laevis</i> )
<b>5:00-6:30</b>	Dinner
<b>7:15-8:00</b>	MBL Talk – Roger Hanlon

**Saturday, September 10**

<b>8:30-10:00</b>	Basic/Advanced computing II ( <i>C.elegans D.melanogaster M.musculus P.polytes T.thermophila X.laevis</i> ) Math. Foundations II ( <i>A.thaliana D.rerio M.mulatta O.bimaculoides S.aegyptiacus T.roseae</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Basic/Advanced computing II ( <i>C.elegans D.melanogaster M.musculus P.polytes T.thermophila X.laevis</i> ) Math. Foundations II ( <i>A.thaliana D.rerio M.mulatta O.bimaculoides S.aegyptiacus T.roseae</i> )
<b>12:00-8:00</b>	Free time

**Sunday, September 11**

<b>8:30-10:00</b>	Workshop Cobey ( <i>A.thaliana C.elegans D.rerio</i> ) Workshop Novembre ( <i>O.bimaculoides P.polytes S.aegyptiacus</i> ) Workshop Osborne ( <i>T.roseae T.thermophila X.laevis</i> ) Workshop Vander Griend ( <i>D.melanogaster M.mulatta M.musculus</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Workshop Cobey ( <i>A.thaliana C.elegans D.rerio</i> ) Workshop Novembre ( <i>O.bimaculoides P.polytes S.aegyptiacus</i> ) Workshop Osborne ( <i>T.roseae T.thermophila X.laevis</i> ) Workshop Vander Griend ( <i>D.melanogaster M.mulatta M.musculus</i> )
<b>12:00-1:00</b>	Lunch
<b>1:00-2:30</b>	Workshop Cobey ( <i>D.melanogaster O.bimaculoides T.thermophila</i> ) Workshop Novembre ( <i>C.elegans M.mulatta X.laevis</i> ) Workshop Osborne ( <i>D.rerio M.musculus S.aegyptiacus</i> ) Workshop Vander Griend ( <i>A.thaliana P.polytes T.roseae</i> )
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Workshop Cobey ( <i>D.melanogaster O.bimaculoides T.thermophila</i> ) Workshop Novembre ( <i>C.elegans M.mulatta X.laevis</i> ) Workshop Osborne ( <i>D.rerio M.musculus S.aegyptiacus</i> ) Workshop Vander Griend ( <i>A.thaliana P.polytes T.roseae</i> )
<b>5:00-6:30</b>	Dinner
<b>6:30-7:10</b>	Life after PhD a) ( <i>D.rerio T.thermophila</i> ) Life after PhD b) ( <i>D.melanogaster S.aegyptiacus</i> ) Navigating first year a) ( <i>A.thaliana X.laevis</i> ) Navigating first year b) ( <i>C.elegans T.roseae</i> ) Publishing primer a) ( <i>M.mulatta P.polytes</i> ) Publishing primer b) ( <i>M.musculus O.bimaculoides</i> )
<b>7:20-8:00</b>	Life after PhD a) ( <i>A.thaliana X.laevis</i> ) Life after PhD b) ( <i>C.elegans T.roseae</i> ) Navigating first year a) ( <i>M.mulatta P.polytes</i> ) Navigating first year b) ( <i>M.musculus O.bimaculoides</i> ) Publishing primer a) ( <i>D.rerio T.thermophila</i> ) Publishing primer b) ( <i>D.melanogaster S.aegyptiacus</i> )
<b>8:10-8:50</b>	Life after PhD a) ( <i>M.mulatta P.polytes</i> ) Life after PhD b) ( <i>M.musculus O.bimaculoides</i> ) Navigating first year a) ( <i>D.rerio T.thermophila</i> ) Navigating first year b) ( <i>D.melanogaster S.aegyptiacus</i> ) Publishing primer a) ( <i>A.thaliana X.laevis</i> ) Publishing primer b) ( <i>C.elegans T.roseae</i> )

**Monday, September 12**

<b>8:30-10:00</b>	Workshop Cobey ( <i>M.mulatta S.aegyptiacus T.roseae</i> ) Workshop Novembre ( <i>A.thaliana M.musculus T.thermophila</i> ) Workshop Osborne ( <i>C.elegans D.melanogaster P.polytes</i> ) Workshop Vander Griend ( <i>D.rerio O.bimaculoides X.laevis</i> )
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Workshop Cobey ( <i>M.mulatta S.aegyptiacus T.roseae</i> ) Workshop Novembre ( <i>A.thaliana M.musculus T.thermophila</i> ) Workshop Osborne ( <i>C.elegans D.melanogaster P.polytes</i> ) Workshop Vander Griend ( <i>D.rerio O.bimaculoides X.laevis</i> )
<b>12:00-1:00</b>	Lunch
<b>1:00-2:30</b>	Workshop Cobey ( <i>M.musculus P.polytes X.laevis</i> ) Workshop Novembre ( <i>D.melanogaster D.rerio T.roseae</i> ) Workshop Osborne ( <i>A.thaliana M.mulatta O.bimaculoides</i> ) Workshop Vander Griend ( <i>C.elegans S.aegyptiacus T.thermophila</i> )
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Workshop Cobey ( <i>M.musculus P.polytes X.laevis</i> ) Workshop Novembre ( <i>D.melanogaster D.rerio T.roseae</i> ) Workshop Osborne ( <i>A.thaliana M.mulatta O.bimaculoides</i> ) Workshop Vander Griend ( <i>C.elegans S.aegyptiacus T.thermophila</i> )
<b>5:00-6:30</b>	Dinner
<b>7:15-8:00</b>	MBL Talk – David Remsen

**Tuesday, September 13**

<b>8:30-10:00</b>	Workshop Munro/Rust
<b>10:00-10:30</b>	Coffee break
<b>10:30-12:00</b>	Workshop Munro/Rust
<b>12:00-1:00</b>	Lunch
<b>1:00-2:30</b>	Workshop Munro/Rust
<b>2:30-2:45</b>	Coffee break
<b>2:45-4:30</b>	Workshop Munro/Rust
<b>5:00-6:30</b>	Dinner
<b>5:00-6:00</b>	Wrapping up
<b>6:00-8:00</b>	BBQ

**Wednesday, September 14****10:00-10:30** Departure for BOS

# Tutorials

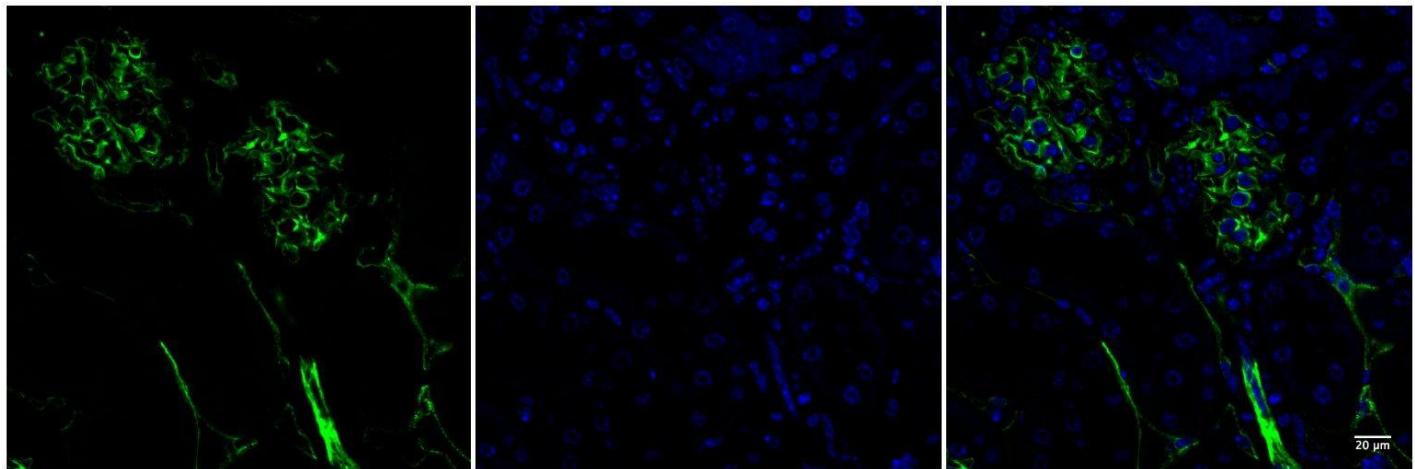
**Imaging**  
**Mathematical Foundations I**  
**Mathematical Foundations II**  
**Basic Computing I**  
**Basic Computing II**  
**Advanced Computing I**  
**Advanced Computing II**

## Image Processing with ImageJ Exercises

Using the information in the ImageJ Tutorials and the built-in ImageJ Command Finder tool (open ImageJ / Fiji and type L to get it to pop up), complete the following exercises. You may not be able to complete these exercises in the time you have, but do your best.

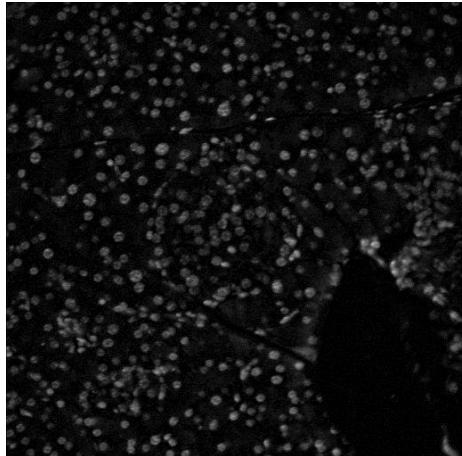
### Exercise 1

Use the cd31 glomeruli.tif file to create the montage shown below. The image comes as two images grouped together in a stack, so your first job is to figure out how to separate them. The pixel size for the image is 0.25 microns. Don't forget the 20um scale bar in the lower right hand corner!

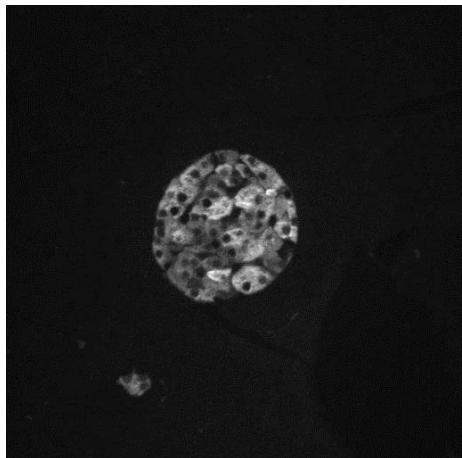


## Exercise 2

Use the pancreatic\_islet.tif to answer the following questions:



- a) How many cells are in the entire image? One nucleus per cell is marked by the DAPI stain in the first image.
- b) What is the average nuclear area for all the cells in the image? Readout should be in  $\mu\text{m}^2$  NOT  $\text{cm}^2$ !
- c) How would you create a data table that includes the mean gray value for every nucleus in the image? They are not all 255!
- d) How many beta cells are in the image? Beta cells are represented by the insulin (green) stain in the second image. Hint: you can still count the nuclei if you find a way to restrict your count to nuclei in an islet.



### Bonus Questions (if you have time)

Bonus 1: if there were multiple islets in the pancreatic\_islet.tif image, how could you count the number of cells PER islet?

Bonus 2: Create a macro to automate the creation of the cd31 glomeruli montage.

# Basic Computing 1 – Introduction to R

*Stefano Allesina*

## Basic Computing 1

- **Goal:** Introduce the statistical software R, and show how it can be used to analyze biological data in an automated, replicable way. Showcase the RStudio development environment, illustrate the notion of assignment, present the main data structures available in R. Show how to read and write data, how to execute simple programs, and how to modify the stream of execution of a program through conditional branching and looping.
- **Audience:** Biologists with little or no background in programming.
- **Installation:** To complete the tutorial, we will need to install R and RStudio. To install R, follow instructions at [cran.rstudio.com](http://cran.rstudio.com). Then install Rstudio following the instructions at [goo.gl/a42jYE](http://goo.gl/a42jYE).

## Motivation

When it comes to analyzing data, there are two competing paradigms. First, one could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; second, one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is to be preferred, because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a lab mate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce R, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

## What is R?

R is a statistical software that is completely programmable. This means that one can write a program (script) containing a series of commands for the analysis of data, and execute them automatically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is free software: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. You can find

a list of official packages (which have been vetted by R core developers) at [ggplot2.org](http://ggplot2.org); many more are available on GitHub and other websites.

The main hurdle new users face when approaching R is that it is based on a command line interface: when you launch R, you simply open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example,

```
1 + 1
```

```
## [1] 2
```

A little history: R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

## RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. The main advantage is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

Typically, an RStudio window contains four panels:

- **Console** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
- **Source code** In this panel, you can write a program, save it to a file pressing Ctrl + S and then execute it by pressing Ctrl + Shift + S.
- **Environment** This panel lists all the variables you created (more on this later); another tab shows you the history of the commands you typed.
- **Plots** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (just type `help(name_of_command)` in the Console) and packages.

## How to write an R program

An R program is simply a list of commands, which are executed one after the other. The commands are written in a text file (with extension .R). When R executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time R encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. Writing a program is advantageous, however, because it can be automated and shared with other researchers. Moreover, after a while you will have a large code base, so that you can recycle much of your code in several programs.

## The most basic operation: assignment

The most basic operation in any programming language is the assignment. In R, assignment is marked by the operator `<-`. When you type a command in R, it is executed, and the output is printed in the **Console**. For example:

```
sqrt(9)
```

```
## [1] 3
```

If we want to save the result of this operation, we can assign it to a variable. For example:

```
x <- sqrt(9)  
x
```

```
## [1] 3
```

What has happened? We wrote a command containing an assignment operator (`<-`). R has evaluated the right-hand-side of the command (`sqrt(9)`), and has stored the result (3) in a newly created variable called `x`. Now we can use `x` in our commands: every time the command needs to be evaluated, the program will look up which value is associated with the variable `x`, and substitute it. For example:

```
x * 2
```

```
## [1] 6
```

## Types of data

R provides different types of data that can be used in your programs. The basic data types are:

- `logical`, taking only two possible values: `TRUE` and `FALSE`

```
v <- TRUE  
class(v)
```

```
## [1] "logical"
```

- `numeric`, storing real numbers (actually, their approximations, as computers have limited memory and thus cannot store numbers like  $\pi$ , or even 0.2)

```
v <- 3.77  
class(v)
```

```
## [1] "numeric"
```

- `integer`, storing whole numbers

```
v <- 23L # the L signals that this should be stored as integer  
class(v)
```

```
## [1] "integer"
```

- **complex**, storing complex numbers (i.e., with a real and an imaginary part)

```
v <- 23 + 5i # the i marks the imaginary part  
class(v)
```

```
## [1] "complex"
```

- **character**, for strings, characters and text

```
v <- 'a string' # you can use single or double quotes  
class(v)
```

```
## [1] "character"
```

In R, the type of a variable is evaluated at runtime. This means that you can recycle the names of variables. This is very handy, but can make your programs more difficult to read and to debug (i.e., find mistakes). For example:

```
x <- '2.3' # this is a string  
x
```

```
## [1] "2.3"
```

```
x <- 2.3 # this is numeric  
x
```

```
## [1] 2.3
```

## Operators and functions

Each data type supports a certain number of operators and functions. For example, numeric variables can be combined with `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation). A possibly unfamiliar operator is the modulo (`%%`), calculating the remainder of an integer division:

```
5 %% 3
```

```
## [1] 2
```

meaning that `5 %% 3` (5 integer divided by three) is 1 with a remainder of 2. The modulo operator is useful to determine whether a number is divisible for another: if  $y$  is divisible by  $x$ , then  $y \% x$  is 0.

Numeric types also support many built-in functions, such as:

- `abs(x)` absolute value
- `sqrt(x)` square root
- `round(x, digits = 3)` round  $x$  to three decimal digits
- `cos(x)` cosinus (also supported are all the usual trigonometric functions)
- `log(x)` natural logarithm (use `log10` for base 10 logarithms)
- `exp(x)` calculating  $e^x$

Similarly, `character` variables have their own set of functions, such as

- `toupper(x)` make uppercase
- `nchar(x)` count the number of characters in the string
- `paste(x, y, sep = "_")` concatenate strings, joining them using the separator `_`
- `strsplit(x, "_")` separate the string using the separator `_`

Calling a function meant for a certain data type on another will cause errors. If sensible, you can convert a type into another. For example:

```
v <- "2.13"
class(v)

## [1] "character"

# if we call v * 2, we get an error.
# to avoid it, we can convert v to numeric:
as.numeric(v) * 2

## [1] 4.26
```

If sensible, you can use the comparison operators `>` (greater), `<` (lower), `==` (equals), `!=` (differs), `>=` and `<=`, returning a logical value:

```
2 == sqrt(4)
```

```
## [1] TRUE
```

```
2 < sqrt(4)
```

```
## [1] FALSE
```

```
2 <= sqrt(4)
```

```
## [1] TRUE
```

Similarly, you can concatenate several comparison and logical variables using `&` (and), `|` (or), and `!` (not):

```
(2 > 3) & (3 > 1)
```

```
## [1] FALSE
```

```
(2 > 3) | (3 > 1)
```

```
## [1] TRUE
```

## Data structures

Besides these simple types, R provides structured data types, meant to collect and organize multiple values.

### Vectors

The most basic data structure in R is the vector, which is an ordered collection of values of the same type. Vectors can be created by concatenating different values with the function `c()` (concatenate):

```
x <- c(2, 3, 5, 27, 31, 13, 17, 19)
x
```

```
## [1] 2 3 5 27 31 13 17 19
```

You can access the elements of a vector by their index: the first element is indexed at 1, the second at 2, etc.

```
x[3]
```

```
## [1] 5
```

```
x[8]
```

```
## [1] 19
```

```
x[9] # what if the element does not exist?
```

```
## [1] NA
```

NA stands for Not Available. Other special values are `NaN` (Not a Number, e.g., `0/0`), `Inf` (Infinity, e.g., `1/0`), and `NULL` (variable not set). You can test for special values using `is.na(x)`, `is.infinite(x)`, etc.

Note that in R a single number (string, logical) is a vector of length 1 by default. That's why if you type `3` in the console you see `[1] 3` in the output.

You can extract several elements at once (i.e., create another vector), using the colon (`:`) command, or by concatenating the indices:

```
x[1:3]
```

```
## [1] 2 3 5
```

```
x[4:7]
```

```
## [1] 27 31 13 17
```

```
x[c(1,3,5)]
```

```
## [1] 2 5 31
```

You can also use a vector of logical variables to extract values from vectors. For example, suppose we have two vectors:

```
sex <- c("M", "M", "F", "M", "F") # sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # weight in mg
```

and that we want to extract only the weights for the males.

```
sex == "M"
```

```
## [1] TRUE TRUE FALSE TRUE FALSE
```

returns a vector of logical values, which we can use to subset the data:

```
weight[sex == "M"]
```

```
## [1] 0.230 0.281 0.260
```

Given that R was born for statistics, there are many statistical functions you can perform on vectors:

```
length(x)
```

```
## [1] 8
```

```
min(x)
```

```
## [1] 2
```

```
max(x)
```

```
## [1] 31
```

```
sum(x) # sum all elements
```

```
## [1] 117
```

```
prod(x) # multiply all elements
```

```
## [1] 105436890
```

```
median(x) # median value
```

```
## [1] 15
```

```

mean(x) # arithmetic mean

## [1] 14.625

var(x) # unbiased sample variance

## [1] 119.4107

mean(x ^ 2) - mean(x) ^ 2 # population variance

## [1] 104.4844

summary(x) # print a summary

##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
##      2.00    4.50   15.00    14.62   21.00    31.00

```

You can generate vectors of sequential numbers using the colon command:

```

x <- 1:10
x

## [1] 1 2 3 4 5 6 7 8 9 10

```

For more complex sequences, use `seq`:

```

seq(from = 1, to = 5, by = 0.5)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

```

To repeat a value or a sequence several times, use `rep`:

```

rep("abc", 3)

## [1] "abc" "abc" "abc"

rep(c(1,2,3), 3)

## [1] 1 2 3 1 2 3 1 2 3

```

### Exercise:

- Create a vector containing all the even numbers between 2 and 100 (inclusive) and store it in variable `z`.
- Extract all the elements of `z` that are divisible by 12. How many elements match this criterion?
- What is the sum of all the elements of `z`?
- Is it equal to  $51 \cdot 50$ ?
- What is the product of elements 5, 10 and 15 of `z`?
- Create a vector `y` that contains all numbers between 0 and 30 that are divisible by 3. Find the five elements of `y` that are also elements of `z`.
- Does `seq(2, 100, by = 2)` produce the same vector as `(1:50) * 2`?
- What happens if you type `z ^ 2`?

## Matrices

A matrix is a two-dimensional table of values. In case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, etc.):

```
A <- matrix(c(1, 2, 3, 4), 2, 2) # values, nrows, ncols  
A
```

```
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4
```

```
A %*% A # matrix product
```

```
##      [,1] [,2]  
## [1,]    7   15  
## [2,]   10   22
```

```
solve(A) # matrix inverse
```

```
##      [,1] [,2]  
## [1,]   -2  1.5  
## [2,]    1 -0.5
```

```
A %*% solve(A) # this should return the identity matrix
```

```
##      [,1] [,2]  
## [1,]    1    0  
## [2,]    0    1
```

```
B <- matrix(1, 3, 2) # you can fill the whole matrix with a single number (1)  
B
```

```
##      [,1] [,2]  
## [1,]    1    1  
## [2,]    1    1  
## [3,]    1    1
```

```
B %*% t(B) # transpose
```

```
##      [,1] [,2] [,3]  
## [1,]    2    2    2  
## [2,]    2    2    2  
## [3,]    2    2    2
```

```
Z <- matrix(1:9, 3, 3) # by default, matrices are filled by column  
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

To determine the dimensions of a matrix, use `dim`:

```
dim(B)
```

```
## [1] 3 2
```

```
dim(B)[1]
```

```
## [1] 3
```

```
nrow(B)
```

```
## [1] 3
```

```
dim(B)[2]
```

```
## [1] 2
```

```
ncol(B)
```

```
## [1] 2
```

Use indices to access a particular row/column of a matrix:

```
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
Z[1, ] # first row
```

```
## [1] 1 4 7
```

```
Z[, 2] # second column
```

```
## [1] 4 5 6
```

```
Z [1:2, 2:3] # submatrix with coefficients in first two rows, and second and third column
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
```

```
Z[c(1,3), c(1,3)] # indexing non-adjacent rows/columns
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    3    9
```

Some operations use all the elements of the matrix:

```
sum(Z)
```

```
## [1] 45
```

```
mean(Z)
```

```
## [1] 5
```

## Arrays

If you need tables with more than two dimensions, use arrays:

```
M <- array(1:24, c(4, 3, 2))
M
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

You can still determine the dimensions using:

```
dim(M)
```

```
## [1] 4 3 2
```

and access the elements as done for matrices. One thing you should be paying attention to: R drops dimensions that are not needed. So, if you access a “slice” of a 3-dimensional array:

```
M[, , 1]
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

you obtain a matrix:

```
dim(M[, , 1])
```

```
## [1] 4 3
```

This can be problematic, for example, when your code expects an array and R turns your data into a matrix (or you expect a matrix but find a vector). To avoid this behavior, add `drop = FALSE` when subsetting:

```
dim(M[, , 1, drop = FALSE])
```

```
## [1] 4 3 1
```

## Lists

Vectors are good if each element is of the same type (e.g., numbers, strings). Lists are used when we want to store elements of different types, or more complex objects (e.g., vectors, matrices, even lists of lists). Each element of the list can be referenced either by its index, or by a label:

```
mylist <- list(Names = c("a", "b", "c", "d"), Values = c(1, 2, 3))
mylist
```

```
## $Names
## [1] "a" "b" "c" "d"
##
## $Values
## [1] 1 2 3
```

```
mylist[[1]] # access first element using index
```

```
## [1] "a" "b" "c" "d"
```

```
mylist[[2]] # access second element by index
```

```
## [1] 1 2 3
```

```
mylist$Names # access second element by label
```

```
## [1] "a" "b" "c" "d"
```

```

mylist[["Names"]] # another way to access by label

## [1] "a" "b" "c" "d"

mylist[["Values"]][3] # access third element in second vector

## [1] 3

```

## Data frames

Data frames contain data organized like in a spreadsheet. The columns (typically representing different measurements) can be of different types (e.g., a column could be the date of measurement, another the weight of the individual, or the volume of the cell, or the treatment of the sample), while the rows typically represent different samples.

When you read a spreadsheet file in R, it is automatically stored as a data frame. The difference between a matrix and a data frame is that in a matrix all the values are of the same type (e.g., all numeric), while in a data frame each column can be of a different type.

Because typing a data frame by hand would be tedious, let's use a data set that is already available in R:

```

data(trees) # Girth, height and volume of cherry trees
str(trees) # structure of data frame

## 'data.frame': 31 obs. of 3 variables:
## $ Girth : num 8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
## $ Height: num 70 65 63 72 81 83 66 75 80 75 ...
## $ Volume: num 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...

ncol(trees)

## [1] 3

nrow(trees)

## [1] 31

head(trees) # print the first few rows

##   Girth Height Volume
## 1    8.3     70   10.3
## 2    8.6     65   10.3
## 3    8.8     63   10.2
## 4   10.5     72   16.4
## 5   10.7     81   18.8
## 6   10.8     83   19.7

```

```

trees$Girth # select column by name

## [1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3 11.4 11.4 11.7
## [15] 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2 14.5 16.0 16.3 17.3 17.5 17.9
## [29] 18.0 18.0 20.6

trees$Height[1:5] # select column by name; return first five elements

## [1] 70 65 63 72 81

trees[1:3, ] #select rows 1 through 3

##   Girth Height Volume
## 1    8.3     70   10.3
## 2    8.6     65   10.3
## 3    8.8     63   10.2

trees[1:3, ]$Volume # select rows 1 through 3; return column Volume

## [1] 10.3 10.3 10.2

trees <- rbind(trees, c(13.25, 76, 30.17)) # add a row
trees_double <- cbind(trees, trees) # combine columns
colnames(trees) <- c("Circumference", "Height", "Volume") # change column names

```

### Exercise:

- What is the average height of the cherry trees?
- What is the average girth of those that are more than 75 ft tall?
- What is the maximum height of trees with a volume between 15 and 35 ft<sup>3</sup>?

## Reading and writing data

In most cases, you will not generate your data in R, but import it from a file. By far, the best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data. The syntax of the functions is as follows:

```

read.csv("MyFile.csv") # read the file MyFile.csv
read.csv("MyFile.csv", header = TRUE) # The file has a header.
read.csv("MyFile.csv", sep = ';') # Specify the column separator.
read.csv("MyFile.csv", skip = 5) # Skip the first 5 lines.

```

Note that columns containing strings are typically converted to *factors* (categorical values, useful when performing regressions). To avoid this behavior, you can specify `stringsAsFactors = FALSE` when calling the function.

Similarly, you can save your data frames using `write.table` or `write.csv`. Suppose you want to save the data frame `MyDF`:

```

write.csv(MyDF, "MyFile.csv")
write.csv(MyDF, "MyFile.csv", append = TRUE) # Append to the end of the file.
write.csv(MyDF, "MyFile.csv", row.names = TRUE) # Include the row names.
write.csv(MyDF, "MyFile.csv", col.names = FALSE) # Do not include column names.

```

Let's look at an example: Read a file containing data on the 6th chromosome for a number of Europeans (Data adapted from Stanford HGDP SNP Genotyping Data by John Novembre):

```
ch6 <- read.table("../data/H938_Euro_chr6.geno", header = TRUE)
```

where `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
dim(ch6)
```

```
## [1] 43141      7
```

we have 7 columns, but more than 40k rows! Let's see the first few:

```
head(ch6)
```

```

##   CHR      SNP A1 A2 nA1A1 nA1A2 nA2A2
## 1  6 rs4959515 A  G    0    17   107
## 2  6 rs719065  A  G    0    26    98
## 3  6 rs6596790 C  T    0     4   119
## 4  6 rs6596796 A  G    0    22   102
## 5  6 rs1535053 G  A    5    39    80
## 6  6 rs12660307 C  T    0     3   121

```

and the last few:

```
tail(ch6)
```

```

##      CHR      SNP A1 A2 nA1A1 nA1A2 nA2A2
## 43136  6 rs10946282 C  T    0    16   108
## 43137  6 rs3734763 C  T    19    56    48
## 43138  6 rs960744  T  C    32    60    32
## 43139  6 rs4428484 A  G    1    11   112
## 43140  6 rs7775031 T  C    26    56    42
## 43141  6 rs12213906 C  T    1    11   112

```

The data contains the number of homozygotes (`nA1A1`, `nA2A2`) and heterozygotes (`nA1A2`), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals:

- `CHR` The chromosome (6 in this case)
- `SNP` The identifier of the Single Nucleotide Polymorphism
- `A1` One of the alleles
- `A2` The other allele
- `nA1A1` The number of individuals with the particular combination of alleles.

### Exercise:

- How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. Note: you can access the columns by index (e.g., `ch6[,5]`), or by name (e.g., `ch6$nA1A1`, or also `ch6[,"nA1A1"]`).
- Try using the function `rowSums` to obtain the same result.
- For how many SNPs do we have that all sampled individuals are homozygotes (i.e., all `A1A1` or all `A2A2`)?
- For how many SNPs, are more than 99% of the sampled individuals homozygous?

## Conditional branching

Now we turn to writing actual programs in the **Source code** panel. To start a new R program, press **Ctrl + Shift + N**. This will open an **Untitled** script. Save the script by pressing **Ctrl + S**: save it as `conditional.R` in the directory `basic_computing_1/code/`. To make sure you're working in the directory where the script is contained, on the menu on the top choose **Session -> Set Working Directory -> To Source File Location**.

Now type the following script:

```
print("Hello world!")
x <- 4
print(x)
```

and execute the script by pressing **Ctrl + Shift + S**. You should see `Hello World!` and `4` printed in your console.

As you saw in this simple example, when R executes the program, it starts from the top and proceeds toward the end of the file. Every time it encounters a command (for example, `print(x)`, printing the value of `x` into the console), it executes it.

When we want a certain block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met){
  # Execute this block of code
} else {
  # Execute this other block of code
}
```

For example, add these lines to the script `conditional.R`, and run it again:

```
print("Hello world!")
x <- 4
print(x)
if (x %% 2 == 0){
  my_message <- paste(x, "is even")
} else {
  my_message <- paste(x, "is odd")
}
print(my_message)
```

We have created a conditional branching point, so that the value of `my_message` changes depending on whether `x` is even (and thus the remainder of the integer division by 2 is 0), or odd. Change the line `x <- 4` to `x <- 131` and run it again.

**Exercise:** What does this do?

```
x <- 36
if (x > 20){
  x <- sqrt(x)
} else {
  x <- x ^ 2
}
if (x > 7) {
  print(x)
} else if (x %% 2 == 1){
  print(x + 1)
}
```

## Looping

Another way to change the flow of the program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

R provides you with two ways to loop over blocks of commands: the `for` loop, and the `while` loop. Let's start with the `for` loop, which is used to iterate over a vector (or a list): for each value of the vector, a series of commands will be run, as shown by the following example, which you can type in a new script called `forloop.R`.

```
myvec <- 1:10 # vector with numbers from 1 to 10

for (i in myvec) {
  a <- i ^ 2
  print(a)
}
```

In the code above, the variable `i` takes the value of each element of `myvec` in sequence. Inside the block defined by the `for` loop, you can use the variable `i` to perform operations.

The anatomy of the `for` statement:

```
for (variable in list_or_vector) {
  execute these commands
} # automatically moves to the next value
```

For loops are used when you know that you want to perform the analysis using a given set of values (e.g., run over all files of a directory, all samples in your data, all sequences of a fasta file, etc.).

The `while` loop is used when the commands need to be repeated while a certain condition is true, as shown by the following example, which you can type in a script called `whileloop.R`:

```
i <- 1

while (i <= 10) {
  a <- i ^ 2
  print(a)
```

```
i <- i + 1  
}
```

The script performs exactly the same operations we wrote for the `for` loop above. Note that you need to update the value of `i`, (using `i <- i + 1`), otherwise the loop will run forever (infinite loop—to terminate click on the stop button in the top-right corner of the console). The anatomy of the `while` statement:

```
while (condition is met) {  
  execute these commands  
} # beware of infinite loops: remember to update the condition!
```

You can break a loop using the command `break`. For example:

```
i <- 1  
  
while (i <= 10) {  
  if (i > 5) {  
    break  
  }  
  a <- i ^ 2  
  print(a)  
  i <- i + 1  
}
```

**Exercise:** What does this do? Try to guess what each loop does, and then create and run a script to confirm your intuition.

```
z <- seq(1, 1000, by = 3)  
for (k in z) {  
  if (k %% 4 == 0) {  
    print(k)  
  }  
}  
  
z <- readline(prompt = "Enter a number: ")  
z <- as.numeric(z)  
isthisspecial <- TRUE  
i <- 2  
while (i < z) {  
  if (z %% i == 0) {  
    isthisspecial <- FALSE  
    break  
  }  
  i <- i + 1  
}  
if (isthisspecial == TRUE) {  
  print(z)  
}
```

## Useful Functions

We conclude with a list of useful functions that will help you write your programs:

- `range(x)`: minimum and maximum of a vector  $x$
- `sort(x)`: sort a vector  $x$
- `unique(x)`: remove duplicate entries from vector  $x$
- `which(x == a)`: returns a vector of the indices of  $x$  having value  $a$
- `list.files("path_to_directory")`: list the files in a directory (current directory if not specified)
- `table(x)` build a table of frequencies

**Exercises:** What does this code do? For each snippet of code, first try to guess what will happen. Then, write a script and run it to confirm your intuition.

```
v <- c(1,3,5,5,3,1,2,4,6,4,2)
v <- sort(unique(v))
for (i in v){
  if (i > 2){
    print(i)
  }
  if (i > 4){
    break
  }
}
```

```
x <- 1:100
x <- x[which(x %% 7 == 0)]
```

```
my_files <- sort(list.files("../data/Saavedra2013/", full.names = TRUE))
for (f in my_files){
  M <- read.table(f)
  print(paste("The file", basename(f), "contains a matrix with", nrow(M),
  "rows and ", ncol(M), "columns. There are", sum(M == 1),
  "coefficients that are 1 and", sum(M == 0), "that are 0."))
}
```

```
my_amount <- 10
while (my_amount > 0){
  my_color <- NA
  while(is.na(my_color)){
    tmp <- readline(prompt="Do you want to bet on black or red? ")
    tmp <- tolower(tmp)
    if (tmp == "black") my_color <- "black"
    if (tmp == "red") my_color <- "red"
    if (is.na(my_color)) print("Please enter either red or black")
  }
  my_bet <- NA
  while(is.na(my_bet)){
    tmp <- readline(prompt="How much do you want to bet? ")
    tmp <- as.numeric(tmp)
    if (is.numeric(tmp) == FALSE){
      print("Please enter a number")
    } else {
      if (tmp > my_amount){
        print("You don't have enough money!")
      } else {
        my_bet <- tmp
      }
    }
  }
}
```

```

        my_amount <- my_amount - tmp
    }
}
}
lady_luck <- sample(c("red", "black"), 1)
if (lady_luck == my_color){
  my_amount <- my_amount + 2 * my_bet
  print(paste("You won!! Now you have", my_amount, "gold dobloons"))
} else {
  print(paste("You lost!! Now you have", my_amount, "gold dobloons"))
}
}
print("I told you this was not a good idea...")

```

## Exercises in groups

### Nobel nominations

The file `../data/nobel_nominations.csv` contains the nominations to the Nobel prize from 1901 to 1964. There are three columns (the file has no header): a) the field (e.g. `Phy` for physics), b) the year of the nomination, c) the id and name of the nominee.

- Take your favorite field. Who received most nominations?
- Who received nominations in more than one field?
- Take the field of physics. Which year had the largest number of nominees?
- What is the average number of nominees for each field? Calculate the average number of nominee for each field across years.

### Hints

- You will need to subset the data. To make operations clearer, you can give names to the columns. For example, suppose you stored the data in in a data frame called `nobel`. Then `colnames(nobel) <- c("Field", "Year", "Nominee")` will do the trick.
- The simplest way to obtain a count from a vector is to use the command `table`. The command `sort(table(my_vector))` produces a table of the occurrences in `my_vector` sorted from few to many counts.
- You can build a table using more than one vector. For example, store the Nobel nominations in the data frame `nobel`, and name the columns as suggested above. The command `head(table(nobel$Nominee, nobel$Field))` will build a table with the number of nominations in each field (column) for each nominee (rows).

# Basic Computing 2 – Packages, Functions, Documenting code

*Stefano Allesina*

## Basic Computing 2

- **Goal:** Show how to install, load and use the many freely available R packages. Illustrate how to write user-defined functions and how to organize code. Showcase basic plotting functions. Introduce the package `knitr` for writing beautiful reports.
- **Audience:** Biologists with basic knowledge of R.
- **Installation:** To produce well-documented code, we need to instal the package `knitr`. We will also use the package `MASS` for statistics.

## Packages

R is the most popular statistical computing software among biologists due to its highly specialized packages, often written from biologists for biologists. You can contribute a package too! The RStudio support ([goo.gl/harVqF](#)) provides guidance on how to start developing R packages and Hadley Wickham's free online book ([r-pkgs.had.co.nz](#)) will make you a pro.

You can find highly specialized packages to address your research questions. Here are some suggestions for finding an appropriate package. The Comprehensive R Archive Network (CRAN) offers several ways to find specific packages for your task. You can either browse packages ([goo.gl/7oVyKC](#)) and their short description or select a scientific field of interest ([goo.gl/0WdIcu](#)) to browse through a compilation of packages related to each discipline.

From within your R terminal or RStudio you can also call the function `RSiteSearch("KEYWORD")`, which submits a search query to the website [search.r-project.org](#). The website [rseek.org](#) casts an even wider net, as it not only includes package names and their documentation but also blogs and mailing lists related to R. If your research interests relate to high-throughput genomic data, you should have a look the packages provided by Bioconductor ([goo.gl/7dwQlq](#)).

### Installing a package

To install a package type

```
install.packages("name_of_package")
```

in the Console, or choose the panel **Packages** and then click on *Install* in RStudio.

### Loading a package

To load a package type

```
library(name_of_package)
```

or call the command into your script. If you want your script to automatically install a package in case it's missing, use this boilerplate:

```
if (!require(needed_package, character.only = TRUE, quietly = TRUE)) {  
  install.packages(needed_package)  
  library(needed_package, character.only = TRUE)  
}
```

## Example

For example, say we want to access the dataset `bacteria`, which reports the incidence of *H. influenzae* in Australian children. The dataset is contained in the package `MASS`.

First, we need to load the package:

```
library(MASS)
```

Now we can load the data:

```
data(bacteria)  
bacteria[1:3,]
```

```
##   y ap hilo week  ID      trt  
## 1 y  p    hi     0 X01 placebo  
## 2 y  p    hi     2 X01 placebo  
## 3 y  p    hi     4 X01 placebo
```

## Do shorter titles lead to more citations?

To keep learning about R, we study a simple problem: do papers with shorter titles have more citations? This is what claimed by Letchford *et al.*, who in 2015 analyzed 140,000 papers ([dx.doi.org/10.1098/rsos.150266](https://doi.org/10.1098/rsos.150266)) finding that shorter titles correlated with a larger number of citations.

In the `data/citations` folder, you find information on all the articles published between 2004 and 2013 by three top disciplinary journals (*Nature Neuroscience*, *Nature Genetics*, and *Ecology Letters*), which we are going to use to explore the robustness of these findings.

We start by reading the data in. This is a simple `csv` file, so that we can use

```
papers <- read.csv("../data/citations/nature_neuroscience.csv")
```

to load the data. However, running `str(papers)` shows that all the columns containing text have been automatically converted to `Factor` (categorical values, which is good when performing regressions). Because we want to manipulate these columns (for example, count how many characters are in a title), we want to avoid this automatic behavior. We can accomplish this by calling the function `read.csv` with an extra argument:

```
papers <- read.csv("../data/citations/nature_neuroscience.csv", stringsAsFactors = FALSE)
```

Next, we want to take a peek at the data. How large is it?

```
dim(papers)
```

```
## [1] 2000    7
```

Let's see the first few rows:

```
head(papers, 3)
```

```
##          Authors                  Title Year
## 1 Logothetis, N.K.      Francis crick 1916-2004. 2004
## 2 Narain, C. Object-specific unconscious processing. 2005
## 3 Narain, C.           Going down BOLDly. 2006
##          Source.title Page.start Page.end Cited.by
## 1  Nature neuroscience       1027      1028        0
## 2  Nature neuroscience.     1288       NA        0
## 3  Nature neuroscience       474       NA        0
```

Now, we want to test whether papers with longer titles do accrue fewer (or more) citations than those with shorter titles. The first step is therefore to add another column to the data, containing the length of the title for each paper:

```
papers$TitleLength <- nchar(papers>Title)
```

## Basic statistics in R

In the original paper, Letchford *et al.* used rank-correlation: rank all the papers according to their title length and the number of citations. If the Kendall's Tau (rank correlation) is positive, then longer titles are associated with more citations; if Tau is negative, longer titles are associated with fewer citations. In R you can compute rank correlation using:

```
kendall_cor <- cor(papers>TitleLength, papers$Cited.by, method = "kendall")
kendall_cor
```

```
## [1] 0.04528715
```

To perform a significance test, use

```
cor.test(papers>TitleLength, papers$Cited.by, method = "kendall")
```

```
##
##  Kendall's rank correlation tau
##
##  data:  papers>TitleLength and papers$Cited.by
##  z = 3.0023, p-value = 0.002679
##  alternative hypothesis: true tau is not equal to 0
##  sample estimates:
##          tau
##  0.04528715
```

showing that the correlation between the ranks is positive and significant. We have found the opposite effect than Letchford *et al.*—longer titles are associated with **more** citations!

Now we are going to examine the data in a different way, to test whether these results are robust.

## Basic plotting in R

To plot the title length vs. number of citations, we need to learn about plotting in R. To produce a simple scatterplot using the base functions for plotting, simply type:

```
plot(papers>TitleLength, papers$Cited.by)
```

It is hard to detect any trend in this plot, as there are a few papers with many more citations than the rest. We can transform the data by plotting on the y-axis the  $\log_{10}$  of citations + 1 (so that papers with zero citations do not cause errors):

```
plot(papers>TitleLength, log10(papers$Cited.by + 1))
```

Again, it's hard to see any trend in here. Maybe we should plot the best fitting line and overlay it on top of the graph. To do so, we first need to learn about regressions in R.

## Regressions in R

R was born for statistics — the fact that it's very easy to fit a linear regression is not surprising! To build a linear model, simply write

```
# model y = a + bx + error  
my_model <- lm(y ~ x)
```

Because it's more convenient to call the code in this way, let's add a new column to the data frame with the log of citations:

```
papers$LogCits <- log10(papers$Cited.by + 1)
```

And perform a linear regression:

```
model_cits <- lm(papers$LogCits ~ papers>TitleLength)  
# This is the best fitting line  
model_cits
```

```
##  
## Call:  
## lm(formula = papers$LogCits ~ papers>TitleLength)  
##  
## Coefficients:  
## (Intercept)  papers>TitleLength  
##           1.358195          0.006193
```

```

# This is a summary of all the statistics
summary(model_cits)

## 
## Call:
## lm(formula = papers$LogCits ~ papers>TitleLength)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -1.68022 -0.25261  0.02803  0.29188  1.82838 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 1.3581953  0.0445997 30.45   <2e-16 ***
## papers>TitleLength 0.0061927  0.0005339 11.60   <2e-16 ***  
## --- 
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
## 
## Residual standard error: 0.4522 on 1998 degrees of freedom
## Multiple R-squared:  0.06309,    Adjusted R-squared:  0.06263 
## F-statistic: 134.6 on 1 and 1998 DF,  p-value: < 2.2e-16

```

And plotting

```

# plot the points
plot(papers>TitleLength, log10(papers$Cited.by + 1))
# add the best fitting line
abline(model_cits, col = "red")

```

Again, we find a positive trend. One thing to consider, is that in the database we have papers spanning a decade. Naturally, older papers have had more time to accrue citations. In our models, we would like to control for this effect. First, let's plot the distribution of citations for a few years. To produce an histogram in R, use

```

hist(papers$LogCits)
# increase the number of breaks
hist(papers$LogCits, breaks = 15)

```

Alternatively, estimate the density using

```
plot(density(papers$LogCits))
```

Let's plot the density for years 2004, 2009, 2013:

```

# plot the density for the older papers:
plot(density(papers$LogCits[papers$Year == 2004]))
lines(density(papers$LogCits[papers$Year == 2009]), col = "red")
lines(density(papers$LogCits[papers$Year == 2013]), col = "blue")

```

As expected, younger papers have fewer citations. We can account for this fact in our regression model:

```

model_year_length <- lm(papers$LogCits ~ as.factor(papers$Year) + papers>TitleLength)
summary(model_year_length)

##
## Call:
## lm(formula = papers$LogCits ~ as.factor(papers$Year) + papers>TitleLength)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -1.88880 -0.21178  0.01232  0.25186  1.51362 
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)    
## (Intercept)           1.6953101  0.0499837 33.917 < 2e-16 ***
## as.factor(papers$Year)2005 -0.0277435  0.0424764 -0.653 0.513735  
## as.factor(papers$Year)2006 -0.1137399  0.0431189 -2.638 0.008409 ** 
## as.factor(papers$Year)2007 -0.1603601  0.0435023 -3.686 0.000234 *** 
## as.factor(papers$Year)2008 -0.1630806  0.0442859 -3.682 0.000237 *** 
## as.factor(papers$Year)2009 -0.2373747  0.0430316 -5.516 3.91e-08 *** 
## as.factor(papers$Year)2010 -0.2824792  0.0433906 -6.510 9.48e-11 *** 
## as.factor(papers$Year)2011 -0.4927523  0.0425956 -11.568 < 2e-16 *** 
## as.factor(papers$Year)2012 -0.6278381  0.0423156 -14.837 < 2e-16 *** 
## as.factor(papers$Year)2013 -0.6539580  0.0419966 -15.572 < 2e-16 *** 
## papers>TitleLength        0.0056728  0.0004645 12.213 < 2e-16 *** 
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
##
## Residual standard error: 0.3912 on 1989 degrees of freedom
## Multiple R-squared:  0.302, Adjusted R-squared:  0.2985 
## F-statistic: 86.06 on 10 and 1989 DF, p-value: < 2.2e-16

```

Using `as.factor(papers$Year)` we have fitted a model in which each year has a different baseline, and the title length influences this baseline. Again, we find that longer titles are associated with more citations.

## Random numbers

As a reminder, the Kendall's  $\tau$  takes as input two rankings  $x$  and  $y$ , both of length  $n$ . It calculates the number of "concordant pairs", in which if  $x_i > x_j$  then  $y_i > y_j$  and "discordant pairs". Then,

$$\tau = \frac{\text{num. concordant} - \text{num. discordant}}{\frac{n(n-1)}{2}}$$

If  $x$  and  $y$  were completely independent, we would expect  $\tau$  to be distributed with a mean of 0. The variance of the null distribution of  $\tau$  (and hence the p-value calculated above) depends on the data, and is typically approximated as a normal distribution. If you want to have a stronger result, you can use randomizations to approximate the p-value. Simply, compute  $\tau$  for the actual data, and for many "fake" datasets obtained randomizing the data. Your p-value is well approximated by the proportion of  $\tau$  values for the randomized sets that exceed the  $\tau$  value for the actual data.

To perform this randomization, or any simulation, we typically need to draw random numbers. R has functions to sample random numbers from very many different statistical distributions. For example:

```
runif(5) # sample 5 numbers from the uniform distribution between 0 and 1
```

```
## [1] 0.92304189 0.18625552 0.46269236 0.78057537 0.09205984
```

```
runif(5, min = 1, max = 9) # set the limits of the uniform distribution
```

```
## [1] 2.261791 6.451318 5.701103 5.655175 1.777831
```

```
rnorm(3) # three values from standard normal
```

```
## [1] 0.4255381 0.9207260 0.4026744
```

```
rnorm(3, mean = 5, sd = 4) # specify mean and standard deviation
```

```
## [1] 1.924565 4.859652 10.629033
```

To sample from a set of values, use `sample`:

```
v <- c("a", "b", "c", "d")
sample(v, 2) # without replacement
```

```
## [1] "b" "a"
```

```
sample(v, 6, replace = TRUE) # with replacement
```

```
## [1] "d" "b" "a" "b" "a" "a"
```

```
sample(v) # simply shuffle the elements
```

```
## [1] "b" "d" "c" "a"
```

Let's try to write a randomization to calculate p-value associated with the  $\tau$  observed for year 2006.

```
# first, we subset the data
papers_year <- papers[papers$Year == 2006, ] # get all rows matching the year
# compute original tau
tau_original <- cor(papers_year>TitleLength, papers_year$Cited.by, method = "kendall")
tau_original
```

```
## [1] 0.1140872
```

Now we want to calculate it on the “fake” data sets. To have confidence in the first two decimal digits, we should perform about ten thousand randomizations. This and similar randomization techniques are known as “bootstrapping”.

```

num_randomizations <- 10000
pvalue <- 0 # initialize at 0
for (i in 1:num_randomizations){
  # calculate cor on shuffled data
  tau_shuffle <- cor(papers_year>TitleLength,
                      sample(papers_year$Cited.by), # scramble the citations at random
                      method = "kendall")
  if (tau_shuffle >= tau_original){
    pvalue <- pvalue + 1
  }
}
# calculate proportion
pvalue <- pvalue / num_randomizations
pvalue

## [1] 0.0097

```

Note that the p-value is different (and in fact smaller) than that calculated using the normal approximation:

```
cor.test(papers_year>TitleLength, papers_year$Cited.by, method = "kendall")
```

```

##
## Kendall's rank correlation tau
##
## data: papers_year>TitleLength and papers_year$Cited.by
## z = 2.3902, p-value = 0.01684
## alternative hypothesis: true tau is not equal to 0
## sample estimates:
##      tau
## 0.1140872

```

Whenever possible, use randomizations, rather than relying on classical tests. They are more computationally expensive, but they allow you to avoid making assumptions about your data.

## Writing functions

We have written code that analyzes one year of data. If we wanted to repeat the analysis on a different year, we would have to modify the code slightly. Instead of doing that, we can write a function that allows us to select a given year, and randomize the data. To do so, we need to learn about functions.

The R community provides about 7,000 packages. Still, sometimes there isn't an already made function capable of doing what you need. In these cases, you can write your own functions. In fact, it is in general a good idea to always divide your analysis into functions, and then write a small “master” program that calls the functions and performs the analysis. In this way, the code will be much more legible, and you will be able to recycle the functions for your other projects.

A function in R has this form:

```

my_function_name <- function(arguments of the function){
  # Body of the function
  # ...
  #
  return(return_value) # this is optional
}

```

A few examples:

```
sum_two_numbers <- function(a, b){  
  apb <- a + b  
  return(apb)  
}  
sum_two_numbers(5, 7.2)  
  
## [1] 12.2
```

You can set a default value for some of the arguments: if not specified by the user, the function will use these defaults:

```
sum_two_numbers <- function(a = 1, b = 2){  
  apb <- a + b  
  return(apb)  
}  
sum_two_numbers()
```

```
## [1] 3
```

```
sum_two_numbers(3)
```

```
## [1] 5
```

```
sum_two_numbers(b = 9)
```

```
## [1] 10
```

The return value is optional:

```
my_factorial <- function(a = 6){  
  if (as.integer(a) != a) {  
    print("Please enter an integer!")  
  } else {  
    tmp <- 1  
    for (i in 2:a){  
      tmp <- tmp * i  
    }  
    print(paste(a, "!" = "", tmp, sep = ""))  
  }  
}  
my_factorial()
```

```
## [1] "6! = 720"
```

```
my_factorial(10)
```

```
## [1] "10! = 3628800"
```

You can return **only one** object. If you need to return multiple values, organize them into a vector/matrix/list and return that.

```

order_two_numbers <- function(a, b){
  if (a > b) return(c(a, b))
  return(c(b,a))
}
order_two_numbers(runif(1), runif(1))

```

```
## [1] 0.4660082 0.1874920
```

Having learned a little about functions, we want to write one that takes as input a vector of citations, a vector of title lengths, and a number of randomizations to perform. The function returns the value of  $\tau$  as well as the associated p-value. In R, we can write:

```

tau_citations_titlelength <- function(citations, titlelength, num_randomizations = 1000){
  tau_original <- cor(titlelength, citations, method = "kendall")
  pvalue <- 0 # initialize at 0
  for (i in 1:num_randomizations){
    # calculate cor on shuffled data
    tau_shuffle <- cor(titlelength,
                         sample(citations), # scramble the citations at random
                         method = "kendall")
    if (tau_shuffle >= tau_original){
      pvalue <- pvalue + 1
    }
  }
  # calculate proportion
  pvalue <- pvalue / num_randomizations
  # return a list
  return(list(tau = tau_original,
             pvalue = pvalue))
}

```

We can write a loop that calls in turn the function for each year separately:

```

all_years <- sort(unique(papers$Year))
for (my_year in all_years){
  tmp <- tau_citations_titlelength(papers$Cited.by[papers$Year == my_year],
                                    papers>TitleLength[papers$Year == my_year],
                                    1000)
  print(paste(my_year, "-> Tau:", round(tmp$tau, 3), "pvalue:", tmp$pvalue))
}

```

```

## [1] "2004 -> Tau: 0.006 pvalue: 0.454"
## [1] "2005 -> Tau: 0.054 pvalue: 0.128"
## [1] "2006 -> Tau: 0.114 pvalue: 0.009"
## [1] "2007 -> Tau: 0.01 pvalue: 0.402"
## [1] "2008 -> Tau: 0.065 pvalue: 0.099"
## [1] "2009 -> Tau: 0.012 pvalue: 0.388"
## [1] "2010 -> Tau: -0.052 pvalue: 0.844"
## [1] "2011 -> Tau: 0.145 pvalue: 0.002"
## [1] "2012 -> Tau: 0.114 pvalue: 0.004"
## [1] "2013 -> Tau: 0.045 pvalue: 0.157"

```

## Organizing and running code

Now we would like to be able to automate the analysis, such that we can repeat it for each journal. This is a good place to pause and introduce how to go about writing programs that are well-organized, easy to write, and easy to debug.

1. Take the problem, and divide it into its basic building blocks
2. Write the code for each building block separately, and test it thoroughly.
3. Extensively document the code, so that you can understand what you did, how you did it, and why.
4. Combine the building blocks into a master program.

For example, let's say we want to write a program that takes as input the name of a file containing the data on titles, years and citations for a given journal. The program should first run the linear model:

```
log(citations + 1) ~ Year (categorical) + TitleLength
```

And output the coefficient associated with `TitleLength` as well as its p-value.

Then, the program should run the Kendall's test for each year separately, again outputting  $\tau$  and the p-value obtained with the normal approximation for each year.

Dividing it into blocks, we need to write:

- code to load the data, calculate title lengths and log citations
- a function to perform the linear model
- a function to perform the Kendall's test
- a master code putting it all together

Our first function:

```
load_data <- function(filename){  
  papers <- read.csv(filename, stringsAsFactors = FALSE)  
  papers$TitleLength <- nchar(papers$title)  
  papers$LogCits <- log10(papers$cited.by + 1)  
  return(papers)  
}
```

Make sure that everything is well by testing our function on the data:

```
for (my_file in list.files("../data/citations", full.names = TRUE)){  
  print(my_file)  
  print(basename(my_file))  
  papers <- load_data(my_file)  
}
```

```
## [1] "../data/citations/ecology_letters.csv"  
## [1] "ecology_letters.csv"  
## [1] "../data/citations/nature_genetics.csv"  
## [1] "nature_genetics.csv"  
## [1] "../data/citations/nature_neuroscience.csv"  
## [1] "nature_neuroscience.csv"
```

Now the function to fit the linear model:

```

linear_model_year_length <- function(papers){
  my_model <- summary(lm(LogCits ~ as.factor(Year) + TitleLength, data = papers))
  # Extract the coefficient and the pvalue
  estimate <- my_model$coefficients["TitleLength", "Estimate"]
  pvalue <- my_model$coefficients["TitleLength", "Pr(>|t|)"]
  return(list(estimate = estimate,
              pvalue = pvalue))
}

```

Let's run this on all files:

```

for (my_file in list.files("../data/citations", full.names = TRUE)){
  print(basename(my_file))
  papers <- load_data(my_file)
  linear_model <- linear_model_year_length(papers)
  print(paste("Linear model -> coefficient",
             round(linear_model$estimate, 5),
             "pvalue", round(linear_model$pvalue, 5)))
}

```

```

## [1] "ecology_letters.csv"
## [1] "Linear model -> coefficient -3e-04 pvalue 0.45814"
## [1] "nature_genetics.csv"
## [1] "Linear model -> coefficient 0.00276 pvalue 0"
## [1] "nature_neuroscience.csv"
## [1] "Linear model -> coefficient 0.00567 pvalue 0"

```

Now the function that calls the Kendall's test for each year: we write two functions. One subsets the data, and the other simply runs the test.

```

Kendall_test <- function(a, b){
  my_test <- cor.test(a, b, method = "kendall")
  return(list(estimate = as.numeric(my_test$estimate),
              pvalue = my_test$p.value))
}

call_Kendall_by_year <- function(papers){
  all_years <- sort(unique(papers$Year))
  for (yr in all_years){
    my_test <- Kendall_test(papers$TitleLength[papers$Year == yr],
                            papers$Cited.by[papers$Year == yr])
    print(paste("Year", yr, "-> estimate", my_test$estimate, "pvalue", my_test$pvalue))
  }
}

```

Now a master function to test that the program is working:

```

analyze_journal <- function(my_file){
  # First, the linear model
  print(basename(my_file))
  papers <- load_data(my_file)
  linear_model <- linear_model_year_length(papers)

```

```

print(paste("Linear model -> coefficient",
            round(linear_model$estimate, 5),
            "pvalue", round(linear_model$pvalue, 5)))
# Then, Kendall year by year
call_Kendall_by_year(papers)
}
analyze_journal("../data/citations/nature_genetics.csv")

```

```

## [1] "nature_genetics.csv"
## [1] "Linear model -> coefficient 0.00276 pvalue 0"
## [1] "Year 2004 -> estimate 0.045610559310653 pvalue 0.370327813824674"
## [1] "Year 2005 -> estimate -0.0550279205871904 pvalue 0.267581272430449"
## [1] "Year 2006 -> estimate 0.0596670158288517 pvalue 0.213567208478809"
## [1] "Year 2007 -> estimate -0.0390688441353209 pvalue 0.418061571895653"
## [1] "Year 2008 -> estimate 0.0674234973583218 pvalue 0.149431264169447"
## [1] "Year 2009 -> estimate 0.0297023932900025 pvalue 0.524960049608212"
## [1] "Year 2010 -> estimate -0.158944978390644 pvalue 0.00176578355758891"
## [1] "Year 2011 -> estimate 0.130352296718688 pvalue 0.00961339158713348"
## [1] "Year 2012 -> estimate 0.168763320248576 pvalue 0.000111079233389155"
## [1] "Year 2013 -> estimate 0.0792087074318355 pvalue 0.0880890674881281"

```

Finally, let's analyze all the journals!

```

for (my_file in list.files("../data/citations", full.names = TRUE)){
  analyze_journal(my_file)
}


```

```

## [1] "ecology_letters.csv"
## [1] "Linear model -> coefficient -3e-04 pvalue 0.45814"
## [1] "Year 2004 -> estimate -0.0330650126212166 pvalue 0.613150449110409"
## [1] "Year 2005 -> estimate 0.027544573550034 pvalue 0.671742234063287"
## [1] "Year 2006 -> estimate -0.0639728739951933 pvalue 0.334992423839896"
## [1] "Year 2007 -> estimate 0.130805775658087 pvalue 0.0792684296315682"
## [1] "Year 2008 -> estimate -0.0946065886996697 pvalue 0.185851799603582"
## [1] "Year 2009 -> estimate -0.00528034747952401 pvalue 0.932130040345834"
## [1] "Year 2010 -> estimate 0.0478717663229855 pvalue 0.429380872167305"
## [1] "Year 2011 -> estimate -0.103776346604215 pvalue 0.0989632669799205"
## [1] "Year 2012 -> estimate 0.0708774786316527 pvalue 0.205434298051716"
## [1] "Year 2013 -> estimate -0.0532355687009939 pvalue 0.326025326942473"
## [1] "nature_genetics.csv"
## [1] "Linear model -> coefficient 0.00276 pvalue 0"
## [1] "Year 2004 -> estimate 0.045610559310653 pvalue 0.370327813824674"
## [1] "Year 2005 -> estimate -0.0550279205871904 pvalue 0.267581272430449"
## [1] "Year 2006 -> estimate 0.0596670158288517 pvalue 0.213567208478809"
## [1] "Year 2007 -> estimate -0.0390688441353209 pvalue 0.418061571895653"
## [1] "Year 2008 -> estimate 0.0674234973583218 pvalue 0.149431264169447"
## [1] "Year 2009 -> estimate 0.0297023932900025 pvalue 0.524960049608212"
## [1] "Year 2010 -> estimate -0.158944978390644 pvalue 0.00176578355758891"
## [1] "Year 2011 -> estimate 0.130352296718688 pvalue 0.00961339158713348"
## [1] "Year 2012 -> estimate 0.168763320248576 pvalue 0.000111079233389155"
## [1] "Year 2013 -> estimate 0.0792087074318355 pvalue 0.0880890674881281"
## [1] "nature_neuroscience.csv"

```

```

## [1] "Linear model -> coefficient 0.00567 pvalue 0"
## [1] "Year 2004 -> estimate 0.00589142291037872 pvalue 0.918745470907139"
## [1] "Year 2005 -> estimate 0.0535867457573801 pvalue 0.243552261640925"
## [1] "Year 2006 -> estimate 0.114087173434659 pvalue 0.0168412989934459"
## [1] "Year 2007 -> estimate 0.00966171293776095 pvalue 0.843003041215137"
## [1] "Year 2008 -> estimate 0.0652257952013621 pvalue 0.200992557539214"
## [1] "Year 2009 -> estimate 0.012124492386758 pvalue 0.79906576984121"
## [1] "Year 2010 -> estimate -0.0518768931100408 pvalue 0.28616709714268"
## [1] "Year 2011 -> estimate 0.145346619264126 pvalue 0.00174156563082106"
## [1] "Year 2012 -> estimate 0.114380879075143 pvalue 0.0125050217924556"
## [1] "Year 2013 -> estimate 0.0446967410995934 pvalue 0.318724166924561"

```

**Discussion:** How many significant results we should expect, when citations and title lengths are completely independent?

## Documenting the code using knitr

*Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.*

Donald E. Knuth, Literate Programming, 1984

When doing experiments, we typically keep track of everything we do in a laboratory notebook, so that when writing the manuscript, or responding to queries, we can go back to our documentation to find exactly what we did, how we did it, and possibly why we did it. The same should be true for computational work.

RStudio makes it very easy to build a computational laboratory notebook. First, create a new R Markdown file (choose File -> New File -> R Markdown from the menu).

The gist of it is that you write a text file (.Rmd). The file is then read by an interpreter that transforms it into an .html or .pdf file, or even into a Word document. You can use special syntax to render the text in different ways. For example,

```

*****
*Test* **Test2**

# Very large header

## Large header

### Smaller header

## Unordered lists

* First
* Second
  + Second 1
  + Second 2

1. This is

```

## 2. A numbered list

You can insert `inline code`

-----

The code above yields:

---

*Test* **Test2**

## Very large header

### Large header

#### Smaller header

#### Unordered lists

- First
  - Second
    - Second 1
    - Second 2
1. This is
  2. A numbered list

You can insert **inline code**

---

The most important feature of R Markdown, however, is that you can include blocks of code, and they will be interpreted and executed by R. You can therefore combine effectively the code itself with the description of what you are doing.

For example, including

```
```r  
print("hello world!")  
```
```

will become

```
print("hello world!")
```

```
## [1] "hello world!"
```

If you don't want to run the R code, but just display it, use `{r, eval = FALSE}`; if you want to show the output but not the code, use `{r, echo = FALSE}`.

You can include plots, tables, and even render equations using LaTeX. In summary, when exploring your data or writing the methods of your paper, give R Markdown a try!

You can find inspiration in the notes for the Boot Camp: both the notes for Basic and Advanced Computing are written in R Markdown.

## Exercises in groups

### Google Flu Trends

Preis & Moat (2014, [dx.doi.org/10.1098/rsos.140095](https://dx.doi.org/10.1098/rsos.140095)) showed that Google Flu Trends are correlated with outpatient visits due to influenza-like illness. You can find the data, along with its description, in `data/GoogleFlu`.

- Read the data
- Plot number of visits vs. `GoogleFluTrends`
- Calculate the correlation using `cor`
- The data spans 2010-2013. In Aug 2013 Google Flu changed their algorithm. Did this lead to improvements? Compare the data from Aug to Dec 2013 with the same months in 2010, 2011, and 2012. For each, calculate the correlation, and see whether the correlation is higher for 2013.

**Hints** You will need to extract the year from a string for each row. To do so, you can use `substr(google$WeekCommencing, 1,4)`.

# Advanced Computing 1 – Data wrangling and visualization

*Stefano Allesina*

## Data Wrangling and Visualization

- **Goal:** learn how to manipulate large data sets by writing efficient, consistent, and compact code. Introduce the use of `dplyr`, `tidyr`, and the “pipeline” operator `%>%`. Produce beautiful graphs and figures for scientific publications using `ggplot2`.
- **Audience:** experienced R users, familiar with the data type `data.frame`, loops, functions, and having some notions of data bases.
- **Installation:** the following packages need to be installed: `ggplot2`, `dplyr`, `tidyr`, `lubridate`, `gridthemes`

## Data wrangling

As biologists living in the XXI century, we are often faced with tons of data, possibly replicated over several organisms, treatments, or locations. We would like to streamline and automate our analysis as much as possible, writing scripts that are easy to read, fast to run, and easy to debug. Base R can get the job done, but often the code contains complicated operations (think of the cases in which you used `lapply` only because of its speed), and a lot of \$ signs and brackets.

We’re going to learn about the packages `dplyr` and `tidyr`, which can be used to manipulate large data frames in a simple and straightforward way. These tools are also much faster than the corresponding base R commands, are very compact, and can be concatenated into “pipelines”.

To start, we need to import the libraries:

```
library(dplyr)  
library(tidyr)
```

Then, we need a dataset to play with. We take a dataset containing all the Divvy bikes trips in Chicago in July 2014:

```
divvy <- read.csv("../data/Divvy_Trips_July_2014.csv")
```

### A new data type, `tbl`

This is now a data frame:

```
is.data.frame(divvy)
```

`dplyr` ships with a new data type, called a `tbl`. To convert from data frame, use

```
divvy <- tbl_df(divvy)
divvy
```

The nice feature of `tbl` objects is that they will print only what fits on the screen, and also give you useful information on the size of the data, as well as the type of data in each column. Other than that, a `tbl` object behaves very much like a `data.frame`. In some rare cases, you want to transform the `tbl` back into a `data.frame`. For this, use the function `as.data.frame(tbl_object)`.

We can take a look at the data using one of several functions:

- `head(divvy)` shows the first few (10 by default) rows
- `tail(divvy)` shows the last few (10 by default) rows
- `glimpse(divvy)` a summary of the data (similar to `str` in base R)
- `View(divvy)` open in spreadsheet-like window

## Selecting rows and columns

There are many ways to subset the data, either by row (subsetting the *observations*), or by column (subsetting the *variables*). For example, suppose we want to count how many trips (of the > 410k) are very short. The column `tripduration` contains the length of the trip in seconds. Let's select only the trips that lasted less than 3 minutes:

```
filter(divvy, tripduration < 180)
```

You can see that “only” 11,099 trips lasted less than three minutes. We have used the command `filter(tbl, conditions)` to select certain observations. We can combine several conditions, by listing them side by side, possibly using logical operators.

**Exercise:** what does this do?

```
filter(divvy, gender == "Male", tripduration > 60, tripduration < 180)
```

We can also select particular variables using the function `select(tbl, cols to select)`. For example, select `from_station_name` and `from_station_id`:

```
select(divvy, from_station_name, from_station_id)
```

How many stations are represented in the data set? We can use the function `distinct(tbl)` to retain only the rows that differ from each other:

```
distinct(select(divvy, from_station_name, from_station_id))
```

Showing that there are 300 stations, once we removed the duplicates.

Other ways to subset observations:

- `sample_n(tbl, howmany, replace = TRUE)` sample `howmany` rows at random with replacement
- `sample_frac(tbl, proportion, replace = FALSE)` sample a certain proportion (e.g. 0.2 for 20%) of rows at random without replacement

- `slice(tbl, 50:100)` extract the rows between 50 and 100
- `top_n(tbl, 10, tripduration)` extract the first 10 rows, once ordered by `tripduration`

More ways to select columns:

- `select(divvy, contains("station"))` select all columns containing the word `station`
- `select(divvy, -gender, -tripduration)` exclude the columns `gender` and `tripduration`
- `select(divvy, matches("year|time"))` select all columns whose names match a regular expression

## Creating pipelines using `%>%`

We've been calling nested functions, such as `distinct(select(divvy, ...))`. If you have to add another layer or two, the code would become unreadable. `dplyr` allows you to "un-nest" these functions and create a "pipeline", in which you concatenate commands separated by the special operator `%>%`. For example:

```
divvy %>% # take a data table
  select(from_station_name, from_station_id) %>% # select two columns
  distinct() # remove duplicates
```

does exactly the same as the command above, but is much more readable. By concatenating many commands, you can create incredibly complex pipelines while retaining readability.

## Producing summaries

Sometimes we need to calculate statistics on certain columns. For example, calculate the average trip duration. We can do this using `summarise`:

```
divvy %>% summarise(avg = mean(tripduration))
```

which returns a `tbl` object with just the average trip duration. You can combine multiple statistics (use `first`, `last`, `min`, `max`, `n` [count the number of rows], `n_distinct` [count the number of distinct rows], `mean`, `median`, `var`, `sd`, etc.):

```
divvy %>% summarise(avg = mean(tripduration),
                      sd = sd(tripduration),
                      median = median(tripduration))
```

## Summaries by group

One of the most useful features of `dplyr` is the ability to produce statistics for the data once subsetted by *groups*. For example, we would like to measure whether men take longer trips than women. We can then group the data by `gender`, and calculate the mean `tripduration` once the data is split into groups:

```
divvy %>% group_by(gender) %>% summarise(mean = mean(tripduration))
```

showing that women tend to take longer trips than men.

**Exercise:** count the number of trips for Male, Female, and unspecified gender.

## Ordering the data

To order the data according to one or more variables, use `arrange()`:

```
divvy %>% select(trip_id, tripduration) %>% arrange(tripduration)
divvy %>% select(trip_id, tripduration) %>% arrange(desc(tripduration))
```

## Renaming columns

To rename one or more columns, use `rename()`:

```
divvy %>% rename(tt = tripduration)
```

## Adding new variables using mutate

If you want to add one or more new columns, use the function `mutate`:

```
divvy %>% select(from_station_id, to_station_id) %>%
  mutate(mylink = paste0(from_station_id, " -> ", to_station_id))
```

use the function `transmute()` to create a new column and drop the original columns. You can also use `mutate` and `transmute` on grouped data:

```
# A more complex pipeline
divvy %>%
  select(trip_id, gender, tripduration) %>% # select only three columns
  rename(t = tripduration) %>% # rename a column
  group_by(gender) %>% # create a group for each gender value
  mutate(zscore = (t - mean(t)) / sd(t)) %>% # compute z-score for t, according to gender
  ungroup() %>% # remove group information
  arrange(desc(t), zscore, gender) %>% # order by t (decreasing), zscore, and gender
  head(20) # display first 20 rows
```

## Data visualization

The most salient feature of scientific graphs should be clarity. Each figure should make crystal-clear a) what is being plotted; b) what are the axes; c) what do colors, shapes, and sizes represent; d) the message the figure wants to convey. Each figure is accompanied by a (sometimes long) caption, where the details can be explained further, but the main message should be clear from glancing at the figure (often, figures are the first thing editors and referees look at).

Many scientific publications contain very poor graphics: labels are missing, scales are unintelligible, there is no explanation of some graphical elements. Moreover, some color graphs are impossible to understand if printed in black and white, or difficult to discern for color-blind people.

Given the effort that you put in your science, you want to ensure that it is well presented and accessible. The investment to master some plotting software will be rewarded by pleasing graphics that convey a clear message.

In this section, we introduce `ggplot2`, a plotting package for R. This package was developed by Hadley Wickham who contributed many important packages to R (including `dplyr`). Unlike many other plotting

systems, `ggplot2` is deeply rooted in a “philosophical” vision. The goal is to conceive a grammar for all graphical representation of data. Leland Wilkinson and collaborators proposed The Grammar of Graphics. It follows the idea of a well-formed sentence that is composed of a subject, a predicate, and an object. The Grammar of Graphics likewise aims at describing a well-formed graph by a grammar that captures a very wide range of statistical and scientific graphics. This might be more clear with an example – Take a simple two-dimensional scatterplot. How can we describe it? We have:

- **Data** The data we want to plot.
- **Mapping** What part of the data is associated with a particular visual feature? For example: Which column is associated with the x-axis? Which with the y-axis? Which column corresponds to the shape or the color of the points? In `ggplot2` lingo, these are called *aesthetic mappings* (`aes`).
- **Geometry** Do we want to draw points? Lines? In `ggplot2` we speak of *geometries* (`geom`).
- **Scale** Do we want the sizes and shapes of the points to scale according to some value? Linearly? Logarithmically? Which palette of colors do we want to use?
- **Coordinate** We need to choose a coordinate system (e.g., Cartesian, polar).
- **Faceting** Do we want to produce different panels, partitioning the data according to one (or more) of the variables?

This basic grammar can be extended by adding statistical transformations of the data (e.g., regression, smoothing), multiple layers, adjustment of position (e.g., stack bars instead of plotting them side-by-side), annotations, and so on.

Exactly like in the grammar of a natural language, we can easily change the meaning of a “sentence” by adding or removing parts. Also, it is very easy to completely change the type of geometry if we are moving from say a histogram to a boxplot or a violin plot, as these types of plots are meant to describe one-dimensional distributions. Similarly, we can go from points to lines, changing one “word” in our code. Finally, the look and feel of the graphs is controlled by a theming system, separating the content from the presentation.

## Basic `ggplot2`

`ggplot2` ships with a simplified graphing function, called `qplot`. In this introduction we are not going to use it, and we concentrate instead on the function `ggplot`, which gives you complete control over your plotting. First, we need to load the package. While we are at it, let’s also load a package extending its theming system:

```
library(ggplot2)
library(ggthemes)
```

And then, let’s get a small data set, containing the data on the Divvy stations:

```
divvy_stations <- read.csv("../data/Divvy_Stations_July_2014.csv")
```

A particularity of `ggplot2` is that it accepts exclusively data organized in tables (a `data.frame` or a `tbl` object). Thus, all of your data needs to be converted into a data frame format for plotting.

Let’s look at the data:

```
head(divvy_stations)
```

For our first plot, we’re going to plot the position of the stations, using the latitude (*y* axis) and longitude (*x* axis). First, we need to specify a dataset to use:

```
ggplot(data = divvy_stations)
```

As you can see, nothing is drawn: we need to specify what we would like to associate to the *x* axis, and what to the *y* axis (i.e., we want to set the *aesthetic mappings*):

```
ggplot(data = divvy_stations) + aes(x = longitude, y = latitude)
```

Note that we concatenate pieces of our “sentence” using the `+` sign! We’ve got the axes, but still no graph... we need to specify a geometry. Let’s use points:

```
ggplot(data = divvy_stations) + aes(x = longitude, y = latitude) + geom_point()
```

You can now see the outline of Chicago, with the lake on the right (east), the river separating the Loop from the West Loop, etc. As you can see, we wrote a well-formed sentence, composed of **data** + **mapping** + **geometry**. We can add other mappings, for example, showing the capacity of the station using different point sizes:

```
ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, size = dpcapacity) +
  geom_point()
```

or colors

```
ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = dpcapacity) +
  geom_point()
```

## Scatterplots

Using `ggplot2`, one can produce very many types of graphs. The package works very well for 2D graphs (or 3D rendered in two dimensions), while it lack capabilities to draw proper 3D graphs, or networks.

The main feature of `ggplot2` is that you can tinker with your graph fairly easily, and with a common grammar. You don’t have to settle on a certain presentation of the data until you’re ready, and it is very easy to switch from one type of graph to another.

For example, let’s calculate the median `tripduration` by `birthdate`, to see whether older people tend to take longer or shorter trips:

```
duration_byyr <- divvy %>%
  filter(is.na(birthyear) == FALSE) %>% # remove records without birthdate
  filter(birthyear > 1925) %>% # remove ultra centenarian people (probably, errors)
  group_by(birthyear) %>% # group by birth year
  summarise(median_duration = median(tripduration)) # calculate median for each group

pl <- ggplot(data = duration_byyr) + # data
  aes(x = birthyear, y = median_duration) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

We can add a smoother by typing

```
pl + geom_smooth() # spline by default  
pl + geom_smooth(method = "lm", se = FALSE) # linear model, no standard errors
```

**Exercise:** repeat the plot of the median, but grouping the data by `gender` as well as `birthyear`. Set the aesthetic mapping `colour` to plot the results by gender.

## Histograms, density and boxplots

How many trips did each bike take? We can plot a histogram showing the number of trips per bike:

```
ggplot(data = divvy, aes(x = bikeid)) + geom_histogram(binwidth = 50)
```

showing a quite uniform density. Speaking of which, we can draw a density plot:

```
ggplot(data = divvy, aes(x = bikeid)) + geom_density()
```

Similarly, we can produce boxplots, for example showing the tripduration for men and women (in `log10`, as the distribution is close to a lognormal):

```
ggplot(data = divvy, aes(x = gender, y = log10(tripduration))) + geom_boxplot()
```

It is very easy to change geometry, for example switching to a violin plot:

```
ggplot(data = divvy, aes(x = gender, y = log10(tripduration))) + geom_violin()
```

## Duration by weekday

Now we're going to test whether the trip duration varies considerably by weekday. To do so, we load the package `lubridate`, which contains many excellent functions for manipulating dates and times.

```
library(lubridate)
```

we then create a new variable, `tripday` specifying the day of the week when the trip was initiated. First, we want to transform the string `starttime` into a date:

```
head(divvy) %>% mutate(tripday = mdy_hm(starttime)) #mdy_hm specifies the date format
```

then we can call `wday` with `label = TRUE` to have a label specifying the day of the week:

```
head(divvy) %>% mutate(tripday = wday(mdy_hm(starttime), label = TRUE))
```

Looks good! Let's perform this operation on the whole set:

```
divvy <- divvy %>% mutate(tripday = wday(mdy_hm(starttime), label = TRUE))
```

### Exercises:

- Produce a barplot (`geom_bar`) showing the number of trips by day
- Calculate the median trip duration per weekday. Then plot it with the command:

```
ggplot(medianbyweekday, aes(x = tripday, y = mediantrip)) +
    geom_bar(stat = "identity")
```

the command `stat = "identity"` tells `ggplot2` to interpret the `y` aesthetic mapping as the height of the barplot.

## Scales

We can use scales to determine how the aesthetic mappings are displayed. For example, we could set the `x` axis to be in logarithmic scale, or we can choose how the colors, shapes and sizes are used. `ggplot2` uses two types of scales: `continuous` scales are used for continuous variables (e.g., real numbers); `discrete` scales for variables that can only take a certain number of values (e.g., colors, shapes, sizes).

For example, let's plot a histogram of `tripduration`:

```
ggplot(divvy, aes(x = tripduration)) + geom_histogram() # no transformation
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "log")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "log10")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "sqrt", name = "Duration in minutes")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() + scale_x_log10() # shorthand
```

We can use different color scales. We can convert the capacity to a factor, to use discrete scales:

```
pl <- ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = as.factor(dpcapacity)) +
  geom_point()
pl + scale_colour_brewer()
pl + scale_colour_brewer(palette = "Spectral")
pl + scale_colour_brewer(palette = "Blues")
pl + scale_colour_brewer("Station Capacity", palette = "Paired")
```

Or use the capacity as a continuous variable:

```
pl <- ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = dpcapacity) +
  geom_point()
pl + scale_colour_gradient()
pl + scale_colour_gradient(low = "red", high = "green")
pl + scale_colour_gradientn(colours = c("blue", "white", "red"))
```

Similarly, you can use scales to modify the display of the shapes of the points (`scale_shape_continuous`, `scale_shape_discrete`), their size (`scale_size_continuous`, `scale_size_discrete`), etc. To set values manually (useful typically for discrete scales of colors or shapes), use `scale_colour_manual`, `scale_shape_manual` etc.

## Themes

Themes allow you to manipulate the look and feel of a graph with just one command. The package `ggthemes` extends the themes collection of `ggplot2` considerably. For example:

```

library(ggthemes)
pl <- ggplot(divvy, aes(x = tripduration)) +
  geom_histogram() +
  scale_x_continuous(trans = "log")
pl + theme_bw() # white background
pl + theme_economist() # like in the magazine "The Economist"
pl + theme_wsj() # like "The Wall Street Journal"

```

## Faceting

In many cases, we would like to produce a multi-panel graph, in which each panel shows the data for a certain combination of parameters. In `ggplot` this is called *faceting*: the command `facet_grid` is used when you want to produce a grid of panels, in which all the panels in the same row (column) have axis-ranges in common; `facet_wrap` is used when the different panels do not have axis-ranges in common.

For example:

```

pl <- ggplot(data = divvy, aes(x = log10(tripduration))) + geom_histogram(binwidth = 0.1)
show(pl)
ggplot(data = divvy, aes(x = log10(tripduration), group = gender)) +
  geom_histogram(binwidth = 0.1) + facet_grid(~gender)
ggplot(data = divvy, aes(x = log10(tripduration), group = gender)) +
  geom_histogram(binwidth = 0.1) + facet_grid(gender~.)

```

Now faceting by `tripday` and `gender`

```

ggplot(data = divvy, aes(x = log10(tripduration),
                         colour = gender,
                         fill = gender,
                         group = tripday)) +
  geom_histogram(binwidth = 0.1) +
  facet_grid(tripday~gender)

```

## Setting features

Often, you want to simply set a feature (e.g., the color of the points, or their shape), rather than using it to display information (i.e., mapping some aesthetic). In such cases, simply declare the feature outside the `aes`:

```

pl <- ggplot(data = divvy_stations, aes(x = longitude, y = latitude))
pl + geom_point()
pl + geom_point(colour = "red")
pl + geom_point(shape = 3)
pl + geom_point(alpha = 0.5)

```

## Saving graphs

You can either save graphs as done normally in R:

```

# save to pdf format
pdf("my_output.pdf", width = 6, height = 4)
print(my_plot)
dev.off()
# save to svg format
svg("my_output.svg", width = 6, height = 4)
print(my_plot)
dev.off()

```

or use the function `ggsave`

```

# save current graph
ggsave("my_output.pdf")
# save a graph stored in ggplot object
ggsave(plot = my_plot, filename = "my_output.svg")

```

## Multiple layers

You can overlay different plots. To do so, however, they must share some of the aesthetic mappings. The simplest case is that in which you have only one dataset:

```

ggplot(data = divvy_stations, aes(x = longitude, y = latitude)) +
  geom_density2d() +
  geom_point()

```

in this case, the `geom_density2d` and `geom_point` shared the `aes`, and were taken from the same dataset.

Let's build a more complicated example:

```

# Capacity of stations in Michigan Avenue
data1 <- divvy_stations %>%
  filter(grepl("Michigan", as.character(name))) %>%
  select(name, dpcapacity) %>%
  rename(value = dpcapacity)
data1

# Number of trips leaving stations in Michigan Ave
data2 <- divvy %>%
  filter(grepl("Michigan", as.character(from_station_name))) %>%
  mutate(name = from_station_name) %>%
  select(name) %>%
  group_by(name) %>%
  summarise(value = n())
data2

```

Now we want to plot the capacity:

```

pl <- ggplot(data = data1, aes(x = name, y = value)) +
  geom_point() +
  scale_y_log10() +
  theme(axis.text.x=element_text(angle=90, hjust=1)) # rotate labels

```

And overlay the other data set:

```
pl + geom_point(data = data2, colour = "red")
```

which is allowed, as the two datasets have the same `aes`. Note that Divvy should increase the capacity of the station at Michigan & Oak!

## Tidying up data

The best data to plot is the one in *tidy form*, meaning that a) each variable has its own column, and b) each observation has its own row. When data is not in tidy form, you can use the package `tidy` to reshape it.

For example, suppose we want to produce a table in which we have the number of trips departing a certain station by gender. First, we create a summary:

```
station_gender <- divvy %>%
  group_by(from_station_name, gender) %>%
  summarise(tot_trips = n()) %>%
  filter(gender == "Male" | gender == "Female") %>%
  ungroup()
```

Now we would like to create two columns (for `Male` and `Female`), containing the number of trips. To do so, we `spread` the column `gender`:

```
# Syntax:
# my_tbl %>% spread(COL_TO_SPREAD, WHAT_TO_USE_AS_VALUE, OPTIONAL: fill = NA)
station_gender <- station_gender %>% spread(gender, tot_trips)
station_gender
```

Having reshaped the data, we can see that the station with the highest proportion of women is in Hyde Park:

```
station_gender %>%
  mutate(proportion_female = Female / (Male + Female)) %>%
  arrange(desc(proportion_female))
```

In the data, we have a column from the station of departure and one for that of arrival. Suppose that for our analysis we would need only one column for the station name, and a separate column detailing whether this is the start or the end of the trip:

```
all_stations <- divvy %>% select(from_station_name, to_station_name, tripduration)
```

We can `gather` the two columns creating a column specifying whether it's a from/to station, and one containing the name of the station:

```
# Syntax:
# my_tbl %>% gather(NAME_NEW_COL, NAME_CONTENT, COLS_TO_GATHER)
all_stations %>% gather("FromTo", "StationName", 1:2)
```

Finally, sometimes we need to split the content of a column into several columns. We can use `separate` to do this quickly:

```
divvy %>% select(starttime) %>% separate(starttime, into = c("Day", "Time"), sep = " ")
```

## Joining tables

If you have multiple data frames or `tbl` objects with columns in common, it is easy to join them (as in a database). To showcase this, we are going to create a map of all the trips in the data. First, we count the number of trips from/to each pair of stations:

```
num_trips <- divvy %>% group_by(from_station_id, to_station_id) %>% summarise(trips = n())
# remove trips starting and ending at the same point, for easier visualization
num_trips <- num_trips %>% filter(from_station_id != to_station_id)
```

Now we use `inner_join` to combine the data from `num_trips` and `divvy_stations`, creating the columns `x1` and `y1` containing the coordinates of the starting station. If we rename the columns so that their names match, the join is done automatically:

```
only_id_lat_long <- divvy_stations %>% select(id, latitude, longitude)

# Join the coordinates of the starting station
num_trips <- inner_join(num_trips,
                         only_id_lat_long %>%
                           rename(from_station_id = id,
                                  x1 = longitude,
                                  y1 = latitude))

# Join the coordinates of the ending station
num_trips <- inner_join(num_trips,
                         only_id_lat_long %>%
                           rename(to_station_id = id,
                                  x2 = longitude,
                                  y2 = latitude))

num_trips$trips <- as.numeric(num_trips$trips)

# Now we can plot all the trips!
ggplot(data = num_trips,
       aes(x = x1, y = y1, xend = x2, yend = y2,
           alpha = I(trips / max(trips)))) +
  geom_curve() + theme_minimal()
```

## Project: network analysis of Divvy data

Now that we have an overview of the methods available, we are going to perform some simple analysis on the data. First of all, we are going to create a matrix of station-to-station flows, where the rows are the starting stations, the columns the ending stations, and coefficients in the matrix measure the number of trips.

For this, we can use a combination of `dplyr` and `tidyverse`:

```
flows <- divvy %>%
  select(from_station_id, to_station_id) %>%
  group_by(from_station_id, to_station_id) %>%
```

```

summarise(trips = n())
# transform into a matrix
flows_mat <- flows %>% spread(to_station_id, trips, fill = 0) %>% as.matrix()
# remove the first col (use it for row name)
rownames(flows_mat) <- flows_mat[,1]
flows_mat <- flows_mat[,-1]
# see one corner of the matrix
flows_mat[1:10, 1:10]

```

Now we're going to rank stations according to their PageRank, the algorithm at the heart of Google's search engine. The idea of PageRank is to simulate a random walk on a set of web-pages: at each step, the random walker can follow a link (with a probability proportional its weight), or "teleport" to another page at random (with small probability). The walk therefore describes a Markov Chain, whose stationary distribution (Perron eigenvector) is the PageRank score for all the nodes. This value indicates how "central" and important a node in the network is.

Mathematically, we want to calculate the Perron eigenvector of the matrix:

$$M' = (1 - \epsilon)M + \epsilon U$$

Where  $M$  is a nonnegative matrix with columns summing to 1, and  $U$  is a matrix with all coefficients being 1.  $\epsilon$  is the teleport probability.

First, we construct the matrix  $M$ , by dividing each row for the corresponding row sum, and transposing:

```
M <- t(flows_mat / rowSums(flows_mat))
```

Then, we choose a "teleport probability" (here  $\epsilon = 0.01$ ), and build  $M'$ :

```

U <- matrix(1, nrow(M), ncol(M))
epsilon <- 0.01
M_prime <- (1 - epsilon) * M + epsilon * U

```

and calculate the PageRank

```

ev <- eigen(M_prime)$vectors[,1]
# normalize ev
ev <- ev / sum(ev)
page_rank <- data.frame(station_id = as.integer(rownames(M_prime)), pagerank = Re(ev))

```

Which stations are the most "central" Divvy stations in Chicago? Let's plot them out:

```

st_pr <- inner_join(divvy_stations, page_rank, by = c("id" = "station_id"))
st_pr <- st_pr %>% mutate(lab = replace(name, pagerank < 0.0055, NA))
ggplot(st_pr,
       aes(x = longitude, y = latitude, colour = pagerank,
           size = pagerank, label = lab)) +
  geom_point() + geom_text(colour = "black", hjust=0, vjust=0)

```

## Exercises in groups

The file `data/Chicago_Crimes_May2016.csv` contains a list of all the crimes reported in Chicago in May 2016. Form small groups and work on the following exercises:

- **Crime map** write a function that takes as input a crime's Primary Type (e.g., ASSAULT), and draws a map of all the occurrences. Mark a point for each occurrence using Latitude and Longitude. Set the alpha to something like 0.1 to show brighter colors in areas with many occurrences.
- **Crimes by community** write a function that takes as input a crime's Primary Type, and produces a barplot showing the number of crimes per Community area. The names of the community areas are found in the file `data/Chicago_Crimes_CommunityAreas.csv`. You will need to `join` the tables before plotting.
- **Violent crimes** add a new column to the dataset specifying whether the crime is considered violent (e.g., HOMICIDE, ASSAULT, KIDNAPPING, BATTERY, CRIM SEXUAL ASSAULT, etc.)
- **Crimes in time** plot the number of violent crimes against time, faceting by community areas.
- **Dangerous day** which day of the week is the most dangerous?
- **Dangerous time** which time of the day is the most dangerous (divide the data by hour of the day).
- **Correlation between crimes** which crimes tend to have the same pattern? Divide the crimes by day and type, and plot the correlation between crimes using `geom_tile` and colouring the cells according to the correlation (see `cor` for a function that computes the correlation between different columns).

# Advanced Computing 2 – UNIX shell and Regular Expressions

*Stefano Allesina*

## Advanced Computing 2

- **Goal:** Learn to write pipelines for data manipulation and analysis using the **UNIX** shell. Show how to interface the **UNIX** shell and **R**. Introduce the use of regular expressions.
- **Audience:** experienced **R** users, familiar with **dplyr** and **ggplot2**.
- **Installation:** Windows users need to install a **UNIX** shell emulator (e.g., **Git Bash**); for regular expressions, we are going to use the **R** package **stringr**.

## The **UNIX** Shell

**UNIX** is an operating system (i.e., the software that lets you interface with the computer) developed in the 1970s by a group of programmers at the AT&T Bell laboratories. Among them were Brian Kernighan and Dennis Ritchie, who also developed the programming language **C**. The new operating system was an immediate success in academic circles, with many scientists writing new programs to extend its features. This mix of commercial and academic interest led to the many variants of **UNIX** available today (e.g., OpenBSD, Sun Solaris, Apple OS X), collectively denoted as **\*nix** systems. Linux is the open source **UNIX** clone whose “engine” (kernel) was written from scratch by Linus Torvalds with the assistance of a loosely-knit team of hackers from across the internet.

All **\*nix** systems are multi-user, network-oriented, and store data as plain text files that can be exchanged between interconnected computer systems. Another characteristic is the use of a strictly hierarchical file system.

## Why use **UNIX**?

Many biologists are not familiar with coding in **\*nix** systems and, given that the learning curve is initially fairly steep, we start by listing the main advantages of these systems over possible alternatives.

First, **UNIX** is an operating system written by programmers for programmers. This means that it is an ideal environment for developing your code and storing your data.

Second, hundreds of small programs are available to perform simple tasks. These small programs can be strung together efficiently so that a single line of **UNIX** commands can perform complex operations, which otherwise would require writing a long and complex program. The possibility of creating these pipelines for data analysis is especially important for biologists, as modern research groups produce large and complex data sets, whose analysis requires a level of automation that would be hard to achieve otherwise. For instance, imagine working with millions of files by having to open each one of them manually to perform an identical task, or try opening your single 80Gb whole-genome sequencing file in a software with a graphical user interface! In **UNIX**, you can string a number of small programs together, each performing a simple task, and create a complex pipeline that can be stored in a script (a text file containing all the commands). Then, you can let the computer analyze all of your data while you’re having a cup of coffee.

Third, text is the rule: almost anything (including the screen, the mouse, etc.) in **UNIX** is represented as a text file. Using text files means that all of your data can be read and written by any machine, and without

the need for sophisticated (and expensive) proprietary software. Text files are (and always will be) supported by any operating system and you will still be able to access your data decades from today (while this is not the case for most commercial software). The text-based nature of **UNIX** might seem unusual at first, especially if you are used to graphical interfaces and proprietary software. However, remember that **UNIX** has been around since the early 1970s and will likely be around at the end of your career. Thus, the hard work you are putting into learning **UNIX** will pay off over a lifetime.

The long history of **UNIX** means that a large body of tutorials and support web sites are readily available online. Last but not least, **UNIX** is very stable, robust, secure, and—in the case of Linux—freely available.

In the end, entirely avoiding to work in a **UNIX** “shell” is almost impossible for a professional scientist: basically all resources for High-Performance Computing (computer clusters, large workstations, etc.) run a **UNIX** or Linux operating system. Similarly, the transfer of large files, websites, and data between machines is typically accomplished through command-line interfaces.

## Directory structure

In **UNIX** we speak of “directories”, while in a graphical environment the term “folder” is more common. These two terms are interchangeable and refer to a structure that may contain sub-directories and files. The **UNIX** directory structure is organized hierarchically in a tree. As a biologist, you can think of this structure as a phylogenetic tree. The common ancestor of all directories is also called the “root” directory and is denoted by an individual slash (/). From the root directory, several important directories branch:

- `/bin` contains several basic programs.
- `/etc` contains configuration files.
- `/dev` contains the files connecting to devices such as the keyboard, mouse and screen.
- `/home` contains the home directory of each user (e.g., `/home/yourname`; in OS X, your home directory is stored in `/Users/yourname`).
- `/tmp` contains temporary files.

You will typically work in your home directory. From there, you can access the Desktop, Downloads, Documents, and other directories you are likely familiar with. When you navigate the system you are in one directory and can move deeper in the tree or upward towards the root.

## Using the terminal

*Terminal* refers to the interface that you use to communicate with the kernel (the core of the operating system). The terminal is also called *shell*, or *command-line interface* (CLI). It processes the commands you type, translates them for the kernel, and shows you the results of your operations. There are several shells available. Here, we concentrate on the most popular one, the `bash` shell, which is the default shell in both Ubuntu and OS X.

In Ubuntu, you can open a shell by pressing `Ctrl + Alt + t` or by opening the dash (hold the `Meta` key) and typing `Terminal`. In OS X, you want to open the application `Terminal.app`, which is located in the folder *Utilities* within *Applications*. Alternatively, you can type `Terminal` in *Spotlight*. In either system, the shell will automatically start in your home directory. Windows users can launch `Git Bash` or another terminal emulator.

The command line prompt ends with a “dollar” (\$) sign. This means the terminal is ready to accept your commands. Here, a \$ sign at the beginning of a line of code signals that the command has to be executed in your terminal. You do not need to type the \$ sign in your terminal, just copy the command that follows it.

In **UNIX**, you can use the `Tab` key to reduce the amount you have to type, which in turn reduces errors caused by typos. When you press `Tab` in a (properly configured) shell, it will try to automatically complete your

command, directory or file name (if multiple completions are possible, you can display them all by hitting the **Tab** key twice). Additionally, you can navigate the history of commands you typed by using the up/down arrows (you do not need to re-type a command that you recently executed). There are also shortcuts that help pace through long lines of code:

- **Ctrl + A** Go to the beginning of the line
- **Ctrl + E** Go to the end of the line
- **Ctrl + L** Clear the screen
- **Ctrl + U** Clear the line before the cursor position
- **Ctrl + K** Clear the line after the cursor
- **Ctrl + C** Kill the command that is currently running
- **Ctrl + D** Exit the current shell
- **Alt + F** Move cursor forward one word (in OS X, **Esc + F**)
- **Alt + B** Move cursor backward one word (in OS X, **Esc + B**)

Mastering these and other keyboard shortcuts will save you a lot of time. You may want to print this list and keep it next to your keyboard—in a while you will have them all memorized and will start using them automatically.

## Basic UNIX commands

Here we introduce some of the most basic (and most useful) UNIX commands. We write the commands in **fixed-width font** and specific, user-provided input is capitalized in square brackets. Again, the brackets and special formatting are not required to execute a command in your terminal.

Many commands require some arguments (e.g., copy which file to where), and all can be modified using the several options available. Typically, options are either written as a dash followed by a single letter (older style, e.g., **-f**) or two dashes followed by words (newer style, e.g., **--full-name**). A command, its options and arguments are separated by a space.

## How to get help in UNIX

UNIX ships with hundreds of commands. As such, it is impossible to remember them all, let alone all their possible options. Fortunately, each command is described in detail in its manual page, which can be accessed directly from the shell by typing **man [COMMAND OF YOUR CHOICE]** (not available in **Git Bash**). Use arrows to scroll up and down and hit **q** to close the manual page. Checking the exact behavior of a command is especially important, given that the shell will execute any command you type without asking whether you know what you’re doing (so that it will promptly remove all of your files, if that’s the command you typed). You may be used to more forgiving (and slightly patronizing) operating systems in which a pop-up window will warn you whenever something you’re doing is considered dangerous. In **UNIX**, it is always better to consult the manual rather than improvising.

If you want to interrupt the execution of a command, press **Ctrl + C** to halt any command that is currently running in your shell.

## Navigating the directory system

You can navigate the hierarchical **UNIX** directory system using these commands:

- **pwd** print the path of the current working directory.
- **ls** list the files and sub-directories in the current directory. **ls -a** list all (including hidden) files. **ls -l** return the long list with detailed information. **ls -lh** provide file sizes with units (B, M, K, etc. ).}

- `cd [NAMEOFTDIR]` change directory. `cd ..` move one directory up; `cd /` move to the root directory; `cd ~` move to your home directory; `cd -` go back to the directory you visited previously (like “Back” in a browser).

## Handling directories and files

Create and delete files or directories using the following commands:

- `cp [FROM] [TO]` copy a file. The first argument is the file to copy. The second argument is where to copy it (either a directory or a file name).
- `mv [FROM] [TO]` move or rename a file. Move a file by specifying two arguments: the file, and the destination directory. Rename a file by specifying the old and the new file name in the same directory.
- `touch [FILENAME]` Update the date of last access to the file. Interestingly, if the file does not exist, this command will create an empty file.
- `rm [TOREMOVE]` remove a file. `rm -r` deletes the contents of a directory recursively (i.e., including all files and sub-directories in it; use with caution!). Similarly, `rm -f` removes the file and suppresses any prompt asking whether you are sure you want to remove the file.
- `mkdir [DIRECTORY]` make a directory. To create nested directories, use the option `-p` (e.g., `mkdir -p d1/d2/d3`).
- `rmdir [DIRECTORY]` remove an empty directory.

## Printing and modulating files

UNIX was especially designed to handle text files, which is apparent when considering the multitude of commands dealing with text. Here are a few popular ones:

- `less [FILENAME]` progressively print a file on the screen (press `q` to exit). Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, in UNIX, `less` is `more`.
- `cat [FILENAME]` concatenate and print files.
- `wc [FILENAME]` line, word, and byte (character) count of a file.
- `sort [FILENAME]` sort the lines of a file and print the result to the screen.
- `uniq [FILENAME]` show only unique lines of a file. The file needs to be sorted first for this to work properly.
- `file [FILENAME]` determine the type of a file.
- `head [FILENAME]` print the `head` (i.e., first few lines of a file).
- `tail [FILENAME]` print the `tail` (i.e., last few lines of a file).
- `diff [FILE1] [FILE2]` show the differences between two files.

**Exercise** To familiarize yourself with these commands, try the following:

- Go to the `data` directory for this tutorial.
- How many lines are in file `Marra2014_data.fasta`?
- Go back to the `code` directory.
- Create the empty file `toremove.txt`.
- List the content of the directory.
- Remove the file `toremove.txt`.

## Miscellaneous commands

- `echo "[A STRING]"` print the string `[A STRING]`.
- `time` time the execution of a command.

- `wget [URL]` download the webpage at `[URL]`. (Available in Ubuntu; for OS X look at `curl`, or install `wget`).
- `history` list the last commands you executed.

## Advanced UNIX commands

### Redirection and pipes

So far, we have printed the output of each command (e.g., `ls`) directly to the screen. However, it is easy to direct the output to a file (*redirect*) or use it as the input of another command (*pipe*). Stringing commands together in pipes is the real power of **UNIX**—the ability to perform complex processing of large amounts of data in a single line of commands. First, we show how to redirect the output of a command into a file:

```
$ [COMMAND] > filename
```

Note that if the file `filename` exists, it will be overwritten. If instead we want to append to an existing file, we can use the `>>` symbol as in the following line:

```
$ [COMMAND] >> filename
```

When the command is very long and complex, we might want to redirect the content of a file as input to a command, “reversing” the flow:

```
$ [COMMAND] < filename
```

To run a few examples, let’s start by moving to our `code` directory:

```
$ cd ~/BSD-QBio2/tutorials/advanced_computing2/code
```

The command `echo` can be used to print a string on the screen. Instead of printing to the screen, we redirect the output to a file, effectively creating a file containing the string we want to print:

```
$ echo "My first line" > test.txt
```

We can see the result of our operation by printing the file to the screen using the command `cat`:

```
$ cat test.txt
```

To append a second line to the file, we use `>>`:

```
$ echo "My second line" >> test.txt
$ cat test.txt
```

We can redirect the output of any command to a file. For example, it is quite common to have to determine how many files are in a directory. The files could have been created by an instrument or provided by a collaborator. Before analyzing the data, we want to get a sense of how many files will we need to process. If there are thousands of files, it is quite time consuming to count them by hand or even open a file browser that can do the counting for us. It is much simpler and faster to just type a command or two. To try this, let’s create a file listing all the files contained in `data/Saavedra2013`:

```
$ ls ..../data/Saavedra2013 >> filelist.txt
$ cat filelist.txt
```

Now we want to count how many lines are in the file. We can do so by calling the command `wc -l` (count only the lines):

```
$ wc -l filelist.txt
$ rm filelist.txt
```

However, we can skip the creation of the file by creating a short pipeline. The pipe symbol `|` tells the shell to take the output on the left of the pipe and use it as the input of the command on the right of the pipe. To take the output of the command `ls` and use it as the input of the command `wc` we can write:

```
$ ls ..../data/Saavedra2013 | wc -l
```

We have created our first, simple pipeline. In the following sections, we are going to build increasingly long and complex pipelines. The idea is always to start with a command and progressively add one piece after another to the pipeline, each time checking that the result is the desired one.

### Selecting columns using `cut`

When dealing with tabular data, you will often encounter the Comma Separated Values (CSV) Standard File Format. The CSV format is platform and software independent, making it the standard output format of many experimental devices. The versatility of the file format should also make it your preferred choice when manually entering and storing data.

The main UNIX command you want to master for comma-, space-, tab-, or character-delimited text files is `cut`. To showcase its features, we work with data on generation time of mammals published by Pacifici *et al.*. First, let's make sure we are in the right directory (`advanced_computing2/data`). Then, we can print the header (the first line, specifying the content of each column) of the CSV file using the command `head`, which prints the first few lines of a file on the screen, with the option `-n 1`, specifying that we want to output only the first line:

```
$ head -n 1 Pacifici2013_data.csv
TaxID;Order;Family;Genus;Scientific_name;...
```

We now pipe the header to `cut`, specify the character to be used as delimiter (`-d ','`), and use the `head` command to extract the name of the first column (`-f 1`), or the names of the first four columns (`-f 1-4`):

```
$ head -n 1 Pacifici2013_data.csv | cut -d ',' -f 1
TaxID

$ head -n 1 Pacifici2013_data.csv | cut -d ',' -f 1-4
TaxID;Order;Family;Genus
```

Remember to use the Tab key to auto-complete file names and the arrow keys to access your command history.

In the next example, we work with the file content. We specify a delimiter, extract specific columns, and pipe the result to the `head` command—to display only the first few elements:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | head -n 5
Order
Rodentia
Rodentia
Rodentia
Rodentia
Macroscelidea

$ cut -d ';' -f 2,8 Pacifici2013_data.csv | head -n 3
Order;Max_longevity_d
Rodentia;292
Rodentia;456.25
```

Now, we specify the delimiter, extract the second column, skip the first line (the header) using the `tail -n +2` command (i.e., return the whole file starting from the second line), and finally display the first five entries:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | tail -n +2 | head -n 5
Rodentia
Rodentia
Rodentia
Rodentia
Macroscelidea
Rodentia
```

We pipe the result of the previous command to the `sort` command (which sorts the lines), and then again to `uniq`, (which takes only the elements that are not repeated). Effectively, we have created a pipeline to extract the names of all the Orders in the database, from Afrosoricida to Tubulidentata (a remarkable Order, which today contains only the aardvark).

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | tail -n +2 | sort | uniq
Afrosoricida
Carnivora
Cetartiodactyla
...
```

This type of manipulation of character-delimited files is very fast and effective. It is an excellent idea to master the `cut` command in order to start exploring large data sets without the need to open files in specialized programs (if you don't want to modify the content of a file, you should not open it in an editor!).

### **Exercise:**

- If we order all species names (fifth column) of `Pacifici2013_data.csv` in alphabetical order, which is the first species? Which the last?
- How many families are represented in the database?

### **Substituting characters using tr**

We often want to substitute or remove a specific character in a text file (e.g., to convert a comma-separated file into a tab-separated file). Such a one-by-one substitution can be accomplished with the command `tr`. Let's look at some examples in which we use a pipe to pass a string to `tr`, which then processes the text input according to the search term and specific options.

Substitute all characters `a` with `b`:

```
$ echo 'aaaaabbb' | tr 'a' 'b'  
bbbbbbb
```

Substitute every number in the range 1 through 5 with 0:

```
$ echo '123456789' | tr 1-5 0  
000006789
```

Substitute lower-case letters with upper-case:

```
$ echo 'ACTGGcAATT' | tr actg ACTG  
ACTGGCAATT
```

We achieve the same result using bracket expressions that provide a predefined set of characters. Here, we use the set of all lower-case letters [:lower:] and translate into upper-case letters [:upper:]:

```
$ echo 'ACTGGcAATT' | tr [:lower:] [:upper:]  
ACTGGCAATT
```

We can also indicate ranges of characters to substitute:

```
$ echo 'aabbccddeee' | tr a-c 1-3  
112233ddee
```

Delete all occurrences of **a**:

```
$ echo 'aaaaabbbb' | tr -d a  
bbbb
```

“Squeeze” all consecutive occurrences of **a**:

```
$ echo 'aaaaabbbb' | tr -s a  
abbbb
```

Note that the command **tr** reads standard input, and does not operate on files directly. However, we can use pipes in conjunction with **cat**, **head**, **cut**, etc. to create input for **tr**:

```
$ tr ' ' '\t' < inputfile.csv > outputfile.csv
```

In this example we input **inputfile.csv** to the **tr** command to replace all spaces with tabs. Note the use of quotes to specify the space character. The tab is indicated by **\t** and is called a “meta-character”. We use the backslash to signal that the following character should not be interpreted literally, but rather is a special code referring to a character that is difficult to represent otherwise.

Now we can apply the command **tr** and the commands we have showcased earlier to create a new file containing a subset of the data contained in **Pacifici2013\_data.csv**, which we are going to use in the next section.

First, we change directory to the **code**:

```
$ cd ../code/
```

Now, we want to create a version of `Pacifici2013_data.csv` containing only the `Order`, `Family`, `Genus`, `Scientific_name`, and `AdultBodyMass_g` (columns 2-6). Moreover, we want to remove the header, sort the lines according to body mass (with larger critters first), and have the values separated by spaces. This sounds like an awful lot of work, but we're going to see how this can be accomplished piping a few commands together.

First, let's remove the header:

```
$ tail -n +2 ../data/Pacifici2013_data.csv
```

Then, take only the columns 2-6:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6
```

Now, substitute the current delimiter (`;`) with a space:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' ''
```

To sort the lines according to body size, we need to exploit a few of the options for the command `sort`. First, we want to sort numbers (option `-n`); second, we want larger values first (option `-r`, reverse order); finally, we want to sort the data according to the sixth column (option `-k 6`):

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' '' | sort -r -n -k 6
```

That's it. We have created our first complex pipeline. To complete the task, we redirect the output of our pipeline to a new file called `BodyM.csv`.

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' '' | sort -r -n -k 6 > BodyM.csv
```

You might object that the same operations could have been accomplished with a few clicks by opening the file in a spreadsheet editor. However, suppose you have to repeat this task many times, e.g., to reformat every file that is produced by a laboratory device. Then it is convenient to automate this task such that it can be run with a single command.

Similarly, suppose you need to download a large CSV file from a server, but many of the columns are not needed. With `cut`, you can extract only the relevant columns, reducing download time and storage.

## Selecting lines using `grep`

`grep` is a powerful command that finds all the lines of a file that match a given pattern. You can return or count all occurrences of the pattern in a large text file without ever opening it. `grep` is based on the concept of regular expressions, which we will cover just below (but use R, as the syntax is slightly different).

We will test the basic features of `grep` using the file we just created. The file contains data on thousands of species:

```
$ wc -l BodyM.csv
5426 BodyM.csv
```

Let's see how many wombats (family Vombatidae) are contained in the data. First we display the lines that contain the term "Vombatidae":

```
$ grep Vombatidae BodyM.csv
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus krefftii 31849.99
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus latifrons 26163.8
Diprotodontia Vombatidae Vombatus ursinus 26000
```

Now we add the option `-c` to count the lines:

```
$ grep -c Vombatidae BodyM.csv
3
```

Next, we have a look at the genus `Bos` in the data file:

```
$ grep Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
Cetartiodactyla Bovidae Boselaphus Boselaphus tragocamelus 182253
```

Besides all the members of the `Bos` genus, we also match one member of the genus `Boselaphus`. To exclude it, we can use the option `-w`, which prompts `grep` to match only full words:

```
$ grep -w Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
```

Using the option `-i` we can make the search case-insensitive (it will match both upper- and lower-case instances):

```
$ grep -i Bos BodyM.csv
Proboscidea Elephantidae Loxodonta Loxodonta africana 3824540
Proboscidea Elephantidae Elephas Elephas maximus 3269794
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
...
.
```

Sometimes, we want to know which lines precede or follow the one we want to match. For example, suppose we want to know which mammals have body weight most similar to the gorilla (*Gorilla gorilla*). The species are already ordered by size, thus we can simply print the two lines before the match using the option `-B 2` and the two lines after the match using `-A 2`:

```
$ grep -B 2 -A 2 "Gorilla gorilla" BodyM.csv
Cetartiodactyla Bovidae Ovis Ovis ammon 113998.7
Cetartiodactyla Delphinidae Lissodelphis Lissodelphis borealis 113000
Primates Hominidae Gorilla Gorilla gorilla 112589
Cetartiodactyla Cervidae Blastocerus Blastocerus dichotomus 112518.5
Cetartiodactyla Iniidae Lipotes Lipotes vexillifer 112138.3
```

Use option `-n` to show the line number of the match. For example, the gorilla is the 164th largest mammal in the database:

```
$ grep -n "Gorilla gorilla" BodyM.csv
164:Primates Hominidae Gorilla Gorilla gorilla 112589
```

To print all the lines that do not match a given pattern, use the option `-v`. For example, to get the other species of the genus *Gorilla* with the exception of *Gorilla gorilla*, we can use:

```
$ grep Gorilla BodyM.csv | grep -v gorilla
Primates Hominidae Gorilla Gorilla beringei 149325.2
```

To match one of several strings, use `grep "[STRING1]||[STRING2]"`

```
$ grep -w "Gorilla\\|Pan" BodyM.csv
Primates Hominidae Gorilla Gorilla beringei 149325.2
Primates Hominidae Gorilla Gorilla gorilla 112589
Primates Hominidae Pan Pan troglodytes 45000
Primates Hominidae Pan Pan paniscus 35119.95
```

You can use `grep` on multiple files at a time! Simply, list all the files to use instead of just one file.

## Interfacing R and the UNIX shell

Note: this will work if you're using Mac OSX or UNIX; in Windows options are much more limited.

### Calling R from the command line

In R, you typically work in “interactive” mode — you type a command, it gets executed, you type another command, and so on. Often, we want to be able to re-run a script on different data sets or with different parameters. For that purpose you can store all the commands in a text file (typically, with extension `.R`), and then re-run the analysis by typing in the UNIX command line

```
$ Rscript my_script_file.R
```

To properly automate our analysis and figure generation, however, we can additionally pass command-line arguments to R. This allows us for instance to perform the analysis using a specific input file, or save the figure using a specific file name.

`Rscript` accepts command-line arguments, that need to be parsed within R. The code at the beginning of the following script shows how this is accomplished:

```
# Get all the command-line arguments
args <- commandArgs(TRUE)
# Assign each argument to a variable,
# making sure to convert it to the right
# type of variable (string by default)

# check the number of arguments
num.args <- length(args)
print(paste("Number of command-line arguments:", num.args))
# print all the arguments
if (num.args > 0) {
```

```

    for (i in 1:num.args) {
      print(paste(i, "->", args[i]))
    }
}

# We can initially set to default values
# (but pay attention to the order,
# the optional arguments should be at the end)
input.file <- "test.txt"
number.replicates <- 10
starting.point <- 3.14

if (num.args >= 1) {
  input.file <- args[1]
}
if (num.args >= 2) {
  number.replicates <- as.integer(args[2])
}
if (num.args >= 3) {
  starting.point <- as.double(args[3])
}

print(c(input.file, number.replicates, starting.point))

# Save this script as my_script.R
# Run the script in bash with different arguments
# Rscript my_script.R abc.txt 5 100.0
# Rscript my_script.R abc.txt 5
# Rscript my_script.R abc.txt
# Rscript my_script.R

```

## Calling the command line from R

You can call the operating system from within R (assuming you're in /advanced\_computing\_2/code):

```
system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno")
```

You can also capture the output from the shell commands and save it into R. Everything is treated as text (convert to numeric if necessary):

```
numlines <- system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno",
                   intern = TRUE)
```

```
numlines
```

```
## [1] "43142"
```

You can also use a combination of shell commands and `read.table` to capture more complex output:

```
mydf <- system("grep rs125283 ../../basic_computing_1/data/H938_Euro_chr6.geno",
               intern = TRUE)
mydf <- read.table(file = textConnection(mydf))
mydf
```

```

##   V1          V2 V3 V4 V5 V6  V7
## 1 6 rs12528302 G  A 26 59 39
## 2 6 rs12528322 G  A  0 21 103
## 3 6 rs12528313 G  T  1 25 98
## 4 6 rs12528341 C  T  3 31 90

```

## Regular expressions in R

Sometimes data is hidden in free text. Think of citations in a manuscript, mentions of DNA motifs in tables, etc. You could copy and paste data from these unstructured texts yourself, but if you have much text, the task is very boring and error-prone. What you need is a way to describe a text pattern to a computer, and then have it extract the data automatically. Regular Expressions do exactly that.

Because you want to describe a text pattern using text, a level of abstraction is inevitable. What you want to do is to construct a pattern using **literal** characters and **metacharacters**.

For all our examples, we will use the package **stringr**, which makes the regular expression syntax consistent (there are many *dialects*), and provides a set of easy-to-use functions:

```
library(stringr)
```

All the functions have a common structure. For example, **str\_extract** extracts text matching a pattern: **str\_extract(text, pattern)**. The simplest possible expression is one in which the pattern is described literally (i.e., we want to find exactly the text we're typing):

```
str_extract("a string of text", "t")
```

```

## [1] "t"

str_extract_all("a string of text", "t")

## [[1]]
## [1] "t" "t" "t"

```

Of course, you need to be able to describe much more general patterns. Use the following metacharacters:

- **\d** Match a digit character (0–9)
- **\D** Match any character that is not a digit
- **\n** Match a newline
- **\s** Match a space
- **\t** Match a Tab
- **\b** Match a “word boundary”
- **\w** Match a “word” character (alphanumeric)
- **.** Match any character

Some examples (note that to escape characters, you want to use two backslashes — you need to escape the backslash itself!):

```

# find the first digit
str_extract("123.25 grams", "\d")

## [1] "1"

```

```

# find word separator + word character + word separator
str_extract("Albert Einstein was a genius", "\\b\\w\\b")

## [1] "a"

# find all digits
str_extract_all("my cell is 773 345 6789", "\\d")

## [[1]]
## [1] "7" "7" "3" "3" "4" "5" "6" "7" "8" "9"

# extract all characters
str_extract_all("for example, this and that", ".")
```

```

## [[1]]
## [1] "f" "o" "r" " " "e" "x" "a" "m" "p" "l" "e" ","
## [18] " " "a" "n" "d" " " "t" "h" "a" "t"
```

Of course, you don't want to type \\w fifteen times, in case you are looking for a string that is 15 characters long! Rather, you can use quantifiers:

- \* Match zero or more times. Match as many times as possible.
- \*? Match zero or more times. Match as few times as possible.
- + Match one or more times. Match as many times as possible.
- +? Match one or more times. Match as few times as possible.
- ? Match zero or one times. In case both zero and one time match, prefer one.
- ?? Match zero or one times, prefer zero.
- {n} Match exactly n times.
- {n,} Match at least n times. Match as many times as possible.
- {n,m} Match between n and m times.

### Exercise

What does this do? Try to guess, and then type the command into R

```

str_extract_all("12.06+3.21i", "\\d+\\.\\.?\\d+")
my_str <- "most beautiful and most wonderful have been, and are being, evolved."
str_extract(my_str, "\\b\\w{6,10}\\b")
str_extract(my_str, "\\b\\w+\\b")
str_extract(my_str, "w\\w*")
str_extract(my_str, "b\\w*")
str_extract(my_str, "b\\w+?")
str_extract(my_str, "\\s\\wn\\w+")
```

What if you want to match the characters ?, +, \*, .? You will need to escape them: for example, \\. matches the “dot” character.

You can specify anchors to signal that the match has to be in certain special positions in the text:

- ^ Match at the beginning of a line.
- \$ Match at the end of a line.

```
str_extract("Ah, ba ba ba Barbara Ann", "\\w{2,}$$")
```

```
## [1] "Ann"
```

```
str_extract("Ah, ba ba ba Barbara Ann", "^\\w{2,}")
```

```
## [1] "Ah"
```

To match one of several characters, list them between brackets:

```
str_extract("01234567890", "[3120]+")
```

```
## [1] "0123"
```

```
str_extract("01234567890", "[3-5]+")
```

```
## [1] "345"
```

```
str_extract("supercalifragilisticexpialidocious", "[a-i]{3,}")
```

```
## [1] "agi"
```

To match either of two patterns, use alternations:

```
str_extract_all("The quick brown fox jumps over the lazy dog", "fox|dog")
```

```
## [[1]]
```

```
## [1] "fox" "dog"
```

If you need more complex alternations, use parentheses to separate the patterns.

Parentheses can also be used to define **groups**, which are used when you want to capture unknown text that is however flanked by known patterns. For example, suppose you want to save the user name of a UofC email:

```
str_match_all("sallesina@uchicago.edu mjsmith@uchicago.edu",
  "\b([a-zA-Z0-9]*?)@uchicago.edu")
```

```
## [[1]]
```

```
##   [,1]          [,2]
## [1,] "sallesina@uchicago.edu" "sallesina"
## [2,] "mjsmith@uchicago.edu"     "mjsmith"
```

Note that you have to use `str_match` or `str_match_all` to obtain information on the groups.

## Useful functions

Many functions have a similar `str_***_all` version, returning all matches.

- `str_detect(strings, pattern)` do the `strings` contain the `pattern`? Returns a logical vector.
- `str_locate(strings, pattern)` find the character position of the pattern
- `str_extract(strings, pattern)` extracts the first match
- `str_match(strings, pattern)` like `extract` but capture groups defined by parentheses
- `str_replace(strings, pattern, newstring)` replaces the first matched `pattern` with `newstring`

## Exercises in groups

### Extract primers and polymorphic sites

In the file `data/Ptak_etal_2004.txt` you find a text version of the supplementary materials of Ptak *et al.* (PLoS Biology 2004, doi:10.1371/journal.pbio.0020155).

- Take a look at the file. You see that it first lists the primers used for the study (e.g., TAP2-17-3' -> CTTGGATATAACACCAAACGCA), and then the polymorphic sites for 24 chimpanzees (e.g., .
- Read the text as a single string, intervalled by new lines

```
my_txt <- paste(readLines("../data/Ptak_etal_2004.txt"), collapse="\n")
```

- Use regular expressions to produce a data frame containing the primers used in the study:

```
head(primers)
```

| ID          | Sequence                |
|-------------|-------------------------|
| 1 TAP2-1-5' | GAGAATCACTTGAACCTGGAG   |
| 2 TAP2-2-5' | TTGTCCACAGTGTACCACATGA  |
| 3 TAP2-3-5' | TATTTCCTCCTGGGGTTTCCTT  |
| 4 TAP2-4-5' | CATGATGTGTCATGCTGAATTG  |
| 5 TAP2-5-5' | ATAGAACAAAGAACCAAAGCCCA |
| 6 TAP2-6-5' | GGACAACAGATAAAAGTTGCCCT |

- Write another regular expression to extract the polymorphic region for each chimp. Use `str_replace_all` to remove extra spaces and newlines. The results should look like:

| Chimp | Sequence   |
|-------|--|
| 1 311 | ATACCCTGGAGGCAGAACATCTTCCGATAGACGCCAGTCCCTAGTTGT...  |
| 2 312 | GCACCCCTGGAGGCAGAGCTCTTCCGATAGACGCCAGTCCCTAGTTGC...  |
| 3 313 | GCACCCCTGGAGGCAGAGCTCTTCCGATAGACGCCAGTCCCCAGTTGT...  |
| 4 314 | GCACCCCTGGAGGCAGAGCTCTTCCGATAGACCCCCAGTCCCCAGTTGC... |
| 5 317 | GCACCCCTGGAGGTTGGGGGGCCCCCAGGAGGCCAGCCCCCGGTGCT...   |
| 6 320 | GCACCCCTGGAGATAAGGGCCCCCAGGAGGCCAGCCCCCGGTGCT...     |

### A map of *Science*

Where does science come from? This question has fascinated researchers for decades and even led to the birth of the field of “Science of Science”, where researchers use the same tools they invented to investigate nature to gain insights on the development of science itself. In this exercise, you will build a map of *Science*, showing where articles published in *Science* magazine have originated. You will find two files in the directory `data/MapOfScience`. The first, `pubmed_results.txt`, is the output of a query to PubMed listing all the papers published in *Science* in 2015. You will extract the US ZIP codes from this file, and then use the file `zipcodes_coordinates.txt` to extract the geographic coordinates for each ZIP code.

- Read the file `pubmed_results.txt`, and extract all the US ZIP codes.
- Count the number of occurrences of each ZIP code using `dplyr`.
- Join the table you’ve created with the data in `zipcodes_coordinates.txt`
- Plot the results using `ggplot2` (either use points with different colors/alphas, or render the density in two dimensions)

# **Workshops**

**Workshop Cobey  
Workshop Novembre  
Workshop Osborne  
Workshop Vander Griend  
Workshop Munro & Rust**

# Unraveling dynamics from time series

Instructors: Sarah Cobey and Sylvia Ranjeva

BSD Bootcamp on Quantitative Biology @ MBL

## Goal

The aim of this workshop is to show how mathematical models can be used to investigate dynamical systems. We will begin by constructing and analyzing simple models of the spread of an infectious disease. We'll work with pencil and paper (or the whiteboard versions) before simulating the dynamics in R. We'll see that small nonlinearities can generate complex dynamics, which we can summarize with bifurcation diagrams. Finally, we'll discuss the use of likelihood in model fitting and review the criteria by which to judge a model.

At the end of the workshop, students should have facility analyzing simple models analytically and simulating systems of differential equations in R. They should also understand the steps involved in creating bifurcation diagrams. Conceptually, they should understand the spectrum of “mechanistic” to “statistical” models and the limitations of analyses based on correlations and linear assumptions. They should also be able to define the concept of the likelihood and rank factors to consider in judging the strength of a model.

## Audience

This workshop is intended for biologists investigating nonlinear dynamical systems. The material is thus relevant to molecular and cell biologists, population geneticists, ecologists, immunologists, etc. The lecture assumes some familiarity with ordinary differential equations (ODEs), and the exercises assume familiarity with R. Although both ODEs and R will be covered to some extent in the tutorials, if you are eager to test your knowledge, you can inspect the code and suggested readings below.

## Installation

Please have a recent version of R installed before the workshop. We will also use the deSolve package. It would be helpful if you could try installing this package before the workshop.

## Workshop resources

- Slides
- Code
- Data: We will not be analyzing any real data as part of the workshop. However, if you have a time series in mind that you would like to focus on, please contact the instructors beforehand.

## Readings and additional resources

These readings are not mandatory, but they review some of the techniques and models we will cover in the workshop.

- Lotka-Volterra Model (Wikipedia)
- Compartmental SIR Models (Wikipedia)
- Chaos theory (Wikipedia)
- May 1976: “Simple mathematical models with very complicated dynamics”
- Earn et al. 2000: “A simple model for complex dynamical transitions in epidemics”
- Shrestha et al. 2011: “Statistical inference for multi-pathogen systems”

# Quantitative Bootcamp Dynamical Systems Tutorial

Ed Munro and Mike Rust

August 12, 2016

## Contents

|          |   |           |
|----------|---|-----------|
| 0.1      | Dynamical analysis of cell cycle progression . . . . .                                      | 1         |
| 0.2      | A biochemical circuit for cell cycle control. . . . .                                       | 1         |
| <b>1</b> | <b>Analysis of a very simple circuit</b>  | <b>5</b>  |
| 1.1      | Graphical approach: Flux balance analysis . . . . .   | 6         |
| 1.2      | Numerical approach . . . . .  | 7         |
| 1.3      | Plotting the steady state level of Cdk1 as a function of Cyclin input . . . . .             | 9         |
| 1.4      | Numerical construction of steady state response to cyclin levels: . . . . .                 | 10        |
| <b>2</b> | <b>How does positive feedback lead to switch-like activation of Cdk1?</b>                   | <b>13</b> |
| 2.1      | Flux balance analysis of the positive feedback circuit . . . . .                            | 16        |
| 2.2      | Numerical construction of steady state response to cyclin levels . . . . .                  | 18        |
| <b>3</b> | <b>From switch-like transitions to free-running oscillations: Adding negative feedback.</b> | <b>21</b> |
| 3.1      | Graphical . . . . .   | 24        |
| 3.2      | Numerical . . . . .   | 27        |
| 3.3      | A closer look at the limit cycle dynamics . . . . .   | 27        |
| 3.4      | Challenges . . . . .  | 27        |



## 0.1 Dynamical analysis of cell cycle progression

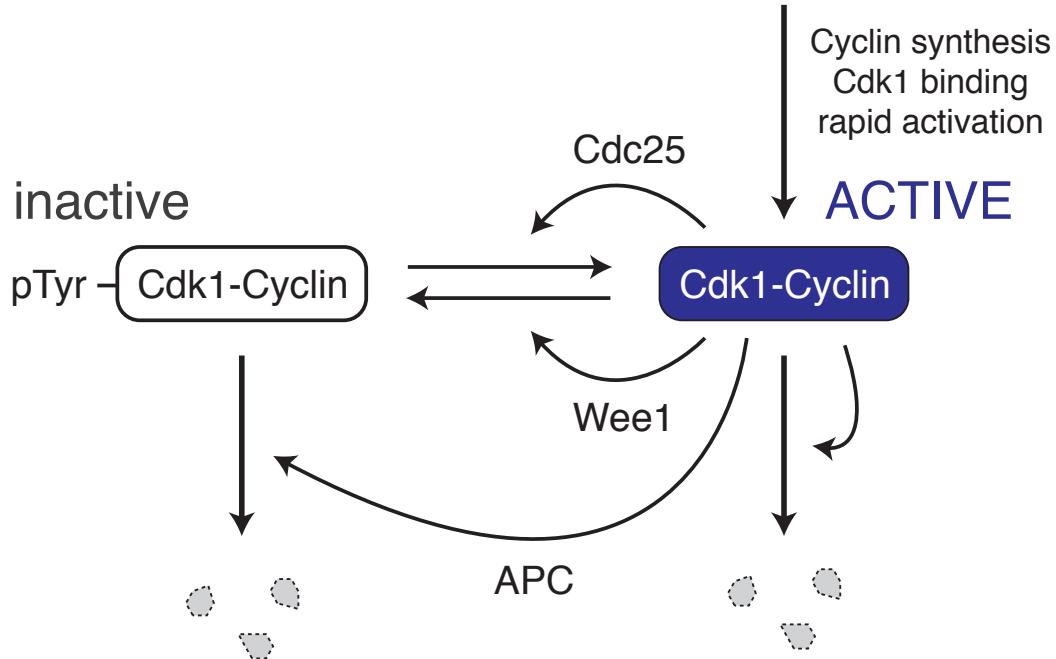
In this tutorial, we will explore how one can turn simple biochemical circuit diagrams into mathematical equations and then analyse those equations, both graphically and numerically, to get insights into circuit's dynamics.

We will focus on one of the most intensively studied biochemical circuits in all of biology, the circuit that controls cell cycle progression in eukaryotic cells. The state of the cell cycle is encoded by the activity of cyclin-dependent kinases (CDKs). CDKs complex with proteins called Cyclins, which undergo cycles of synthesis and destruction over time. When complexed with Cyclin, Cdk can become active and phosphorylate different target proteins to regulate diverse cell cycle functions.

In some cells, such as early embryonic cells, the cell cycle is a freely running oscillator, in which Cyclin levels, and CDK activity, go up and down with clock-like regularity. Other cells progress through the cell cycle in a series of steps, involving switch-like transitions in CDK activity (low  $\rightarrow$  high or high  $\rightarrow$  low), that are controlled by external inputs like nutrient availability or by internal checkpoint controls that monitor the completion of key events like DNA replication or chromosome alignment. As you will see, switch-like transitions and oscillatory dynamics are intimately connected.

## 0.2 A biochemical circuit for cell cycle control.

Figure 1 summarizes a set of basic facts about the core biochemical circuit that is thought to generate cell cycle oscillations. Cyclin protein is synthesized at a constant rate. It forms a complex with Cdk1 (Cdk1-Cyclin) which is rapidly and constitutively activated by the Cdk-activating kinase (CAK, not shown).



**Figure 0.1** Biochemical circuit view of the core cell cycle oscillator

Active Cdk1-Cyclin then participates in three feedback loops: (1) It inhibits a tyrosine kinase called Wee1, which phosphorylates and inactivates Cdk1; (2) It activates a protein phosphatase called Cdc25, which dephosphorylates and reactivates Cdk1 and (3) It promotes the activity of the APC complex which in turn promotes the destruction of Cdk1-Cyclin complexes (both active and inactive).

**Question:** A feedback loop is called positive if it is self-reinforcing (e.g. active Cdk1-Cyclin promotes more active Cdk1-Cyclin) and negative if it is self-antagonizing. Which of these feedback loops is positive and which is negative?

In this tutorial, you will explore whether and how these feedback loops can account for two key properties of cell cycle control observed in experiments:

- The ability to convert slow changes in Cyclin levels into switch-like (all-or-none)

changes in Cdk1 activity.

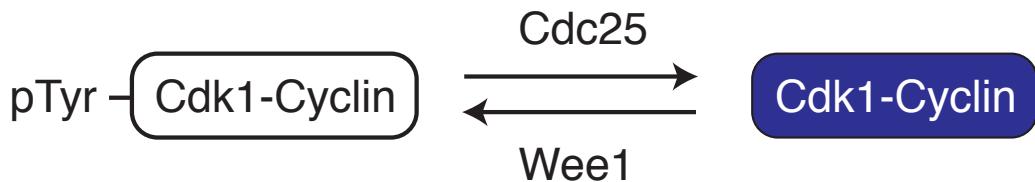
- The ability to produce free-running oscillations in the absence of any additional inputs.



## Session 1

### Analysis of a very simple circuit

Lets start by analyzing a very simple circuit in which the Cdk1-Cyclin complex undergoes rapid conversion between inactive and active states at constant rates.



**Figure 1.1** Simple interconversion of active and inactive Cdk1 at constant rates

From experimental studies, we know that Cyclin binds Cdk1 with very high affinity and that Cdk1 is always much more abundant than Cyclin. We will assume throughout this tutorial that all available Cyclin is complexed with Cdk1, and therefore that the total amount of the complex is the same as the total amount of Cyclin. From now on, we will refer to the Cdk1-Cyclin complex simply as Cdk1.

We want to write an equation that describes how the rate of change in active Cdk1 depends on the current amounts of active and inactive Cdk1. If we let  $C$  represent the total amount of Cyclin protein, and  $A$  represent the amount of active Cdk1, then the amount of inactive Cdk1 is  $(C - A)$ .

The total rate of change in active Cdk1 is equal to the rate at which inactive Cdk1 becomes active minus the rate at which active Cdk1 becomes inactive. If each molecule of inactive Cdk1 becomes active with a fixed probability per unit time described by  $k_{act}$ , then the total activation rate is  $k_{act}(C - A)$ . Likewise, if each molecule of active Cdk1 becomes

## 6 Session 1 Analysis of a very simple circuit

inactive with a probability per unit time described by  $k_{inact}$ , then the inactivation rate is  $k_{inact}A$ .

Putting these together, we can write the following differential equation:

$$\frac{dA}{dt} = k_{act}(C - A) - k_{inact}A \quad (1.1)$$

Equation 1.1 specifies how the rate of change in active Cdk1 depends on its current level and on the three parameters:  $C$ ,  $k_{act}$  and  $k_{inact}$ . You could solve this equation with pencil and paper if you knew the values of all three parameters. However, this is almost never true for more complicated circuits. Here you will learn approaches to solving such equations that do not require pencil and paper. We will focus on two complementary approaches: graphical and computational.

### 1.1 Graphical approach: Flux balance analysis

Lets start with a simple graphical approach called flux balance analysis. The basic idea is to divide the right hand side of Equation 1 into a positive contribution, the positive flux and a negative contribution, the negative flux. As you will see, plotting the positive and negative fluxes as a function of  $A$  can tell us almost everything we want to know about the dynamics of this circuit. Lets start by defining a list of parameters in R and specifying provisional values:

*Here and throughout the tutorial, when you encounter a piece of R code, your response should be to type (or copy and paste) the code into an R terminal window and execute:*

```
1 |   params <- list (k_act=0.1 , k_inact=0.1 , C=100 )
```

Now we define R functions for the positive and negative flux contributions:

```
2 | posflux <- function(A, params) {  
3 |   return (params$k_act * (params$C-A) )  
4 | }  
  
5 | negflux <- function(A, params) {  
6 |   return (params$k_inact * A )  
7 | }
```

Now plot the positive and negative fluxes as a function of  $A$ :

```
1 | A = seq (0 , 100 , 1 )  
2 | plot(A,posflux(A,params), type="l", col="red", main="Flux balance", xlim  
|   =c(0, 100), ylim=c(0, 10))  
  
| lines(A,negflux(A,params), col="blue")
```

The interpretation of this plot is very simple: Every possible state of the system is represented by a point on the horizontal axis, corresponding to a particular amount of A. The sum of the positive and negative flux equals  $dA/dt$ , the total rate of change in A. So  $dA/dt$  is positive when the positive flux is greater than the negative flux; it is negative when the positive flux is less than the negative flux and it is 0 (i.e. the system is at steady state) when the two fluxes are equal. This means that from the flux balance plot, for any current value of A, we can read off immediately how A will change in the next instant of time!

Note that there is a unique steady state point at which the positive and negative curves cross one another, and therefore where  $dA/dt = 0$ . The molecular meaning of this steady state point is that Cdk1 molecules are becoming activated exactly as fast as they are becoming inactivated, so the overall state of the system is constant.

### Exercise 1.1

**Question:** Do you think this steady state is stable or unstable? That is, if you pushed the value of active Cdk1 up or down a little, would it return to the steady state value or move further away?

Here is a simple way to answer this question: Reproduce the above plot on a sheet of paper. Beneath the plot, at different points along the active Cdk1 (A) axis, draw arrows whose direction represents the sign of the difference between the positive and negative flux, and whose magnitude represents the magnitude of that difference. These arrows tell you, for a given value of active Cdk1, in what direction and how fast the amount of active Cdk1 is changing. Reading these arrows, is the steady state point stable or unstable?

## 1.2 Numerical approach

An complementary approach is to solve Equation 1 numerically, using a computer to solve for  $A(t)$  which tells us how the state of the system changes as a function of time. To do so, we need to define a function that represents the right hand side of Equation 1; we need to supply particular values for the parameters  $k_{act}$ ,  $k_{inact}$  and C; an initial value for A, and a sequence of time points at which to evaluate the solution.

Lets start by importing a package called desolve, which defines the relevant functions:

```
2 | ## Import the 'deSolve' package
2 | library (deSolve)
```

## 8 Session 1 Analysis of a very simple circuit

We have already defined the list of parameters above. Now we define the right hand side function:

```
1 | rhs <- function(t, A, params) {  
2 |   return (with (params, list (k_act * (C-A) - k_inact*A)))  
3 | }
```

We define a starting value for A:

```
| A0 <- 0.0
```

and we define a sequence of time points at which to evaluate the solutions:

```
| t <- seq (0 , 100 , 1 )
```

Now we call a numerical ODE solver function called ode to solve the equations, passing it the initial value A0, the time sequence t, the right hand side function rhs, and the list of parameters params:

```
| out <- ode (A0, t, rhs, params)
```

The ode solver returns the solution in the form of an array of values of A at the specified time points contained in t.

We can plot this solution as follows:

```
| plot(out, lwd=2, main="Simple Circuit", xlim=c(0, 100), ylim=c(0, 100))
```

Of course, it is very easy to evaluate the solution for a different initial value of A and add that solution to our plot:

```
1 | A0=10  
2 | out <- ode (A0, t, rhs, params)  
| lines (out, lwd=2 )
```

### Exercise 1.2

Try repeating this for a number of different initial conditions. How does the long term (steady state) behavior of this circuit depend on the initial conditions? Does this agree with what you learned from the graphical analysis?

### 1.3 Plotting the steady state level of Cdk1 as a function of Cyclin input

A key question that we would like to answer is: How does the steady state level of active Cdk1 vary as the amount of Cyclin changes? Again, there are two complementary ways to address this question: graphical and numerical.

#### Graphical construction of steady state response to cyclin levels:

The graphical approach is a simple extension of the flux balance analysis: For each value of C, the flux balance plot tells us what the steady state value of A will be. Note that in Equation 1, only the positive flux (the activation rate) depends on C. This suggests the following simple approach: Plot the negative flux once, and then plot the positive flux for a sequence of different values of C and observe how the position of the crossing (the steady state value of A) varies with C.

Here is a start:

```

A = seq(0, 100, 1)
plot(A, negflux(A, params), type="l", col="red", main="Flux balance
      plot", xlim=c(0, 100), ylim=c(0, 10))
3
params$c=0
lines(A, posflux(A, params), col=blue)
6
params$c=10
lines(A, posflux(A, params), col=blue)

```

### Exercise 1.3

Now continue for a sequence of increasing values of C. On a separate sheet of paper, sketch a graph that shows (qualitatively) the relationship between the steady state level of Cdk1 and the input level of Cyclin.

**Question:** What is the shape of this graph?

**Question:** How does the fraction of active Cyclin/Cdk1 relative to the amount of Cyclin change as the total amount of Cyclin increases?

## 10 Session 1 Analysis of a very simple circuit

Actually, a slicker way to do this would be to use a simple for loop:

```
1 A = seq (0 , 100 , 1 )
  plot (A,negflux (A,params) , type="l" , col="red" , main="Flux balance
    plot" ,      xlim=c (0 , 100 ) , ylim=c (0 , 10 ))
4 for (C in seq (0 ,200 ,10 )){
  params\$C=C
  lines (A,posflux (A,params) , col="blue" )
7 }
```

### 1.4 Numerical construction of steady state response to cyclin levels:

Here is another way to construct the same graph numerically: Start with  $C = 0$ . Set the initial value of  $A$  to 0 and solve Equation 1 numerically to find the steady state level of  $A$ .

```
1 plot(x=NULL,y=NULL,xlim=c(0,100),ylim=c(0,100),xlab="Cyclin level",
      ylab="Steady State Cdk1")
2 params\$C=0
3 A0 = 0
4 out <- ode (A0, t, rhs, params)
5 points(params\$C,out[101,2])
```

Now increase  $C$  by a small amount; use the previous steady state value for  $A$  as a new initial value and solve again for steady state.

```
1 params\$C=10
2 A0 = out[101,2]
3 out <- ode (A0, t, rhs, params)
4 points(params\$C,out[101,2])
```

### Exercise 1.4

Repeat this process many times as you step  $\text{params\$C}$  from 0 to some very high level (say  $\text{params\$C} = 200$ ), adding each steady state value to your plot.

Hint: try using a simple for loop to accomplish the this task.

**Question:** How does this plot compare to the qualitative plot you constructed on paper using flux balance analysis?

**Important note:** For this very simple scenario, we could have gotten this answer much more directly by setting  $dA/dt = 0$  in Equation 1 and then solving algebraically for  $A$  in terms of  $C$ ,  $k_{act}$  and  $k_{inact}$  to obtain:

$$0 = k_{act}(C - A) - k_{inact}A \implies A = \frac{k_{act}C}{k_{act} + k_{inact}} \quad (1.2)$$

However, this is only possible because the right hand side of Equation 1 is very simple. As soon as we add more complicated feedback, the equations we write down become impossible to solve this way!

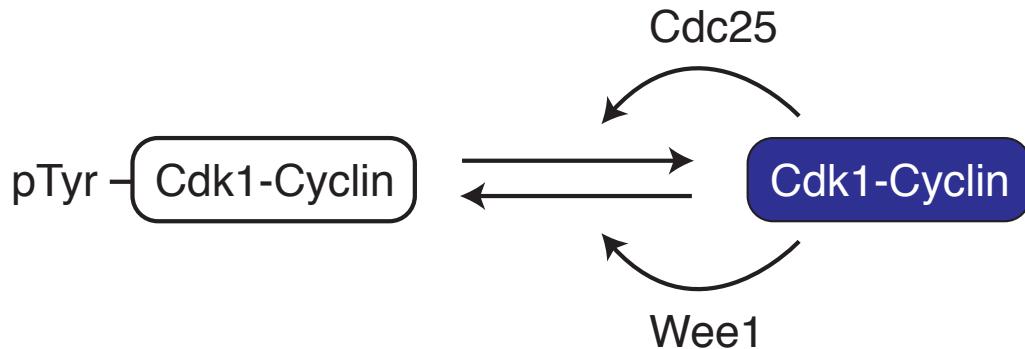
**Challenge:** Above you constructed a response curve in which the input is the level of Cyclin and the response is the steady state level of Cdk1. Use the same approaches to construct a response curve in which the input is the rate of Cdk1 activation  $k_{act}$ . Now do the same for the rate of Cdk1 inactivation  $k_{inact}$ .



## Session 2

### How does positive feedback lead to switch-like activation of Cdk1?

For the simple circuit that you analyzed above, the steady state level of active Cdk1 depends linearly on the level of Cyclin input. Now let's ask how this steady state response changes when we add the two feedback loops in which (a) active Cdk-1 activates Cdc-25 and Cdc-25 promotes activation of Cdk-1, (b) active Cdk-1 inhibits Wee-1 and Wee-1 promotes inactivation of Cdk-1.



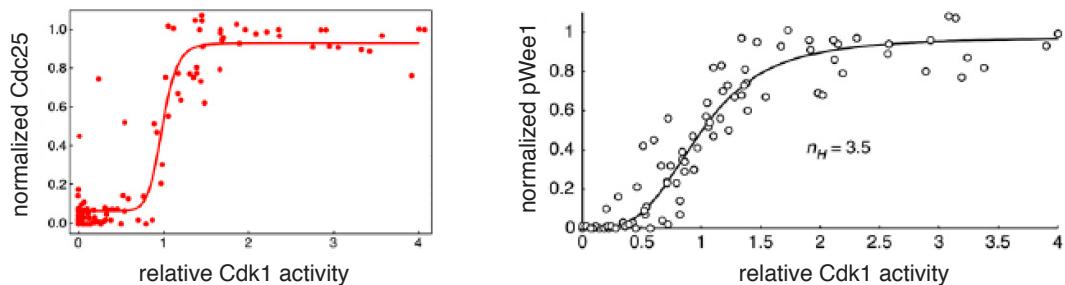
**Figure 2.1** Interconversion of active and inactive Cdk1 with feedback control of activation and inactivation rates

Now the rate of Cdk1 activation is proportional to the product of Cdc25 enzyme activity and inactive Cdk1, and the rate of inactivation is proportional to the product of Wee1 enzyme activity and active Cdk1:

14 Session 2 How does positive feedback lead to switch-like activation of Cdk1?

$$\frac{dA}{dt} = Cdc25(A) \cdot (C - A) - Wee1(A) \cdot A \quad (2.1)$$

We write  $Cdc25(A)$  and  $Wee1(A)$  in Equation 2 to reflect the fact that Cdc25 and Wee1 activities themselves depend on how phosphorylated these enzymes are (through the feedback loops involving Cdk1): Phosphorylated Cdc25 is active; phosphorylated Wee1 is inactive. To build our model, we will use empirical data showing what these functions look like. The dependencies of Cdc25 and Wee1 phosphorylation on Cdk1 activity have been measured experimentally in cytoplasmic extracts and they look like this:



**Figure 2.2** Normalized levels of phosphorylated (active) Cdc25 and phosphorylated (inactive) Wee1 as a function of Cdk1 activity

This kind of S-shaped, or sigmoidal dependence is very common in biology. The sharp rise in output level above some threshold value of the input is called ultrasensitivity. Ultrasensitive responses can arise in many ways in biochemical circuits, for example through cooperative binding, or multisite phosphorylation, or substrate competition (see Ferrell and Ha, 2013 for a nice discussion of this). The saturation of the response at high levels of input reflects the fact that there exists a maximum number of Cdc25 or Wee1 molecules that can become active.

Hill functions are a very useful way to represent sigmoidal responses in mathematical models. Originally invented by Archibald Hill to describe the sigmoidal binding curve for  $O_2$  to hemoglobin, The equation for a Hill function looks like this:

$$Hill(x, K, n) = \frac{x^n}{K^n + x^n} \quad (2.2)$$

where  $x$  is the input variable, and  $K$  and  $n$  are parameters.

Lets define the Hill function in R and graph it for a particular choice of K and n values.

```

2 hill <- function(X,K,n) {
3   return(X^n/(K^n + X^n))
4 }
5 x = seq(0,4,0.1)
6 plot(x,hill(x,1,4),xlim=c(0,4),ylim=c(0,1),xlab=X,ylab=Hill(X))

```

How do the parameters K and n control the shape of the Hill function? Lets plot  $\text{Hill}(x, K, n)$  for a fixed value of  $K$  ( $K = 1$ ) and a range of different  $n$  values:

```

1 plot(x=NULL,y=NULL,xlim=c(0,4),ylim=c(0,1),xlab="X",ylab="Hill(X)")
2 for (n in seq(1,10,1)) {
3   lines(x,hill(x,2,n))
4 }

```

### Exercise 2.1

Now repeat the above, but this time plot the Hill function for a fixed value of  $n$  (say  $n = 4$ ), and a range of  $K$  values.

**Question:** What key features of the Hill functions shape do K and n control?

The solid lines in Figure 4 above show the results of fitting Hill functions to the experimental measurements of phosphorylated (active) Cdc25 and Wee1 vs Cdk, yielding Hill coefficients  $n = 3.5$  for Wee1 and  $n = 11$  for Cdc25. These fits justify the use of Hill functions to represent  $Cdc25(A)$  and  $Wee1(A)$  in Equation 2 above. That is, we can write:

$$Cdc25(A) = a_{Cdc25} + f_{Cdc25} \frac{A^{11}}{K_{Cdc25}^{11} + A^{11}} \quad (2.3)$$

where  $a_{Cdc25}$  represents the basal level of Cdc25 activity,  $f_{Cdc25}$  is the strength of Cdk1 feedback, and  $K_{Cdc25}$  represents the threshold for this feedback.

Similarly, we can write:

$$Wee1(A) = a_{inact} + f_{inact} \frac{K_{inact}^{3.5}}{K_{inact}^{3.5} + A^{3.5}} \quad (2.4)$$

where  $a_{inact}$  represents the minimum level of Wee1 activity when fully inhibited,  $f_{inact}$  is the additional activity when uninhibited, and  $K_{inact}$  is the threshold for inhibitory feedback.

## 16 Session 2 How does positive feedback lead to switch-like activation of Cdk1?

Importantly, the values of  $a_{Cdc25}$ ,  $a_{inact}$ ,  $f_{Cdc25}$ ,  $f_{inact}$ ,  $K_{Cdc25}$ , and  $K_{inact}$  have all been estimated by fitting equations 2.3 and 2.4 to experimental data (see Xiong and Ferrell, 2013).

Using these values, and substituting our expressions for  $Cdc25(A)$  and  $Wee1(A)$  into Equation 2 above gives:

$$\frac{dA}{dt} = \left(0.05 + 1.2 \frac{A^{11}}{34^{11} + A^{11}}\right) \cdot (C - A) - \left(0.08 + 0.4 \frac{A^{3.5}}{33^{3.5} + A^{3.5}}\right) \cdot A \quad (2.5)$$

This equation may look complicated, but it is built in straightforward way out of the pieces discussed above. If it looks overwhelming go back and see where each piece came from. Note here that we've suppressed the units for protein concentrations for simplicity. In Equation 4, the rate of change in  $A$  depends only on  $A$  itself, and on the level of Cyclin ( $C$ ). However, unlike the simple circuit, we can no longer solve Equation 4 with pencil and paper. Here is where flux balance analysis and numerical solutions become really useful! Lets apply the same tools we used for the simple circuit above to analyze the more complicated dynamics of this feedback circuit. As before, we will first characterize active Cdk1 dynamics for a particular level of Cyclin; then well explore how the steady state Cdk1 levels vary with Cyclin levels.

### 2.1 Flux balance analysis of the positive feedback circuit

Lets start by defining a list of parameters and assigning their values from the published data:

```
feedback_params <- list(a_act = 0.16, f_act = 0.8, K_act = 34, n_act =  
11, a_inact = 0.08, f_inact = 0.4, K_inact = 33, n_inact = 3.5, C =  
60)
```

Note that we have assigned a nominal value for Cyclin ( $C = 60$ ).

#### Exercise 2.2

Modify the flux balance analysis approach that we used in Section 1.1.1 to identify the steady states for this system and determine their stability: i.e. first define new positive and negative flux functions based on terms from the right hand side of Equation 4; then plot these functions vs  $A$ .

**Question:** How many steady state crossings are there? which of these do you think are stable and which are unstable?

As you did for the simpler case above, answer this question by reproducing the flux balance plot and drawing arrows below the Cdk1 axis that indicate the magnitude and direction of the rate of change in Cdk1 for different values of Cdk1.

### Exercise 2.3

Numerical analysis of the positive feedback circuit

Now verify your conclusions from the flux balance analysis by solving Equation 4 numerically for different starting values of Cdk1. Following the approach we used in Section 1.1.2, define a new function for the right hand side of Equation 4, and define a sequence of time values at which to evaluate the solutions. Now choose a bunch of different starting values for A, and for each one, solve the equations numerically and plot the solutions.

**Question:** How many different stable steady state points are there?

**Question:** For what range of initial  $A$  values does  $A$  approach each of these stable steady states?

**Question:** What happens if you try to pick starting values closer and closer to an unstable steady state?

**Question:** Do you think the circuit could remain at this point for very long if it were operating in a real cell? Why or why not?

**Summary:** A key take-home message of this analysis is that, at least for some Cyclin

## 18 Session 2 How does positive feedback lead to switch-like activation of Cdk1?

levels, this circuit can exhibit what we call bistable dynamics - it can home in on and remain stably at one of two possible states, depending on initial conditions. How do bistable dynamics shape the response of the circuit to changing levels of Cyclin?

### Exercise 2.4

Map the steady steady response to changing levels of Cyclin.

We are particularly interested in asking how steady state levels of Cdk1 activity change as the levels of Cyclin slowly increase. Can positive feedback generate a switch-like response in Cdk1 activity to slow increases in Cyclin, as observed during cell cycle progression in living cells? Again we can answer these questions using a combination of graphical and numerical approaches:

**Graphical construction of steady state response to cyclin levels:** Lets start by adapting the graphical approach that we used for the simple circuit above. As for the simpler circuit, only the positive flux depends on the value of C. So again, you can plot the negative flux vs A once, and then plot the positive flux vs A for a range of different values of C. Modify the code we used in Section 1.1.3 to do this.

Of course now, for some values of C, there are multiple stable and unstable steady states. As you vary C, these steady states move continuously along the A axis. This is more complicated than the simple circuit, but you can still make sense of whats happening:

As before, on a separate sheet of paper, construct a graph in which the horizontal axis represents the Cyclin level (C) and the vertical axis represents steady state levels of Cdk1(A). Draw a curve with a solid line to indicate the changing positions of stable steady states and curves with a dashed line to indicate the positions of unstable steady states.

We call this graph a bifurcation diagram.

**Question:** What does this bifurcation diagram tell you about the steady state response to changing Cyclin levels? How do you make sense of this when there are multiple possible steady state values for some levels of Cyclin?

Take a few moments to ponder these questions and discuss with your partners before going to the next step.

## 2.2 Numerical construction of steady state response to cyclin levels

Your analysis shows that for an intermediate range of C values, the circuit exhibits bistable dynamics: I.e. for some values of C, the circuit can adopt one of two possible stable steady

states, depending on initial conditions. This suggests that the steady state response of this circuit to changing Cyclin levels will depend on the history of prior Cyclin levels.

### Exercise 2.5

During cell cycle progression, Cyclin and active Cdk1 levels both start out low, and steady state Cdk1 activity levels respond continuously to slow increases in Cyclin levels. Revise the numerical approach that we used for the simple circuit in Section 1.1.4 above to determine the steady state Cdk1 activity responds to slowly increasing Cyclin levels.

Start with low values for C and A; solve to find a steady state value for C. Now increase C in a series of steps. At each step solve the equations numerically, using the previous steady state level of A as a new initial value, and then plot the steady state value of Cdk1 (A) vs the level of input Cyclin.

**Question:** How does this plot differ from the one you constructed earlier for the simple circuit lacking positive feedback? How does it relate to the bifurcation diagram you constructed in the last step?

### Exercise 2.6

Now repeat the same exercise, but this time start with a very high level of Cyclin, and map out the steady state Cdk1 response as you decrease the Cyclin level stepwise towards zero.

Now, on the same graph, plot the steady state Cdk1 values that you obtained by increasing Cyclin slowly from a low value and by decreasing Cyclin slowly from a high value.

**Question:** How do these two plots differ and why? How do these two plots relate to the bifurcation diagram?

**Optional Challenge:** What is the point of having two feedback loops instead of just one?

One possibility is that two feedback loops make the switchlike behavior of the circuit more robust with respect to variation in parameter values. Here is a way to test this idea:

**20** Session 2 How does positive feedback lead to switch-like activation of Cdk1?

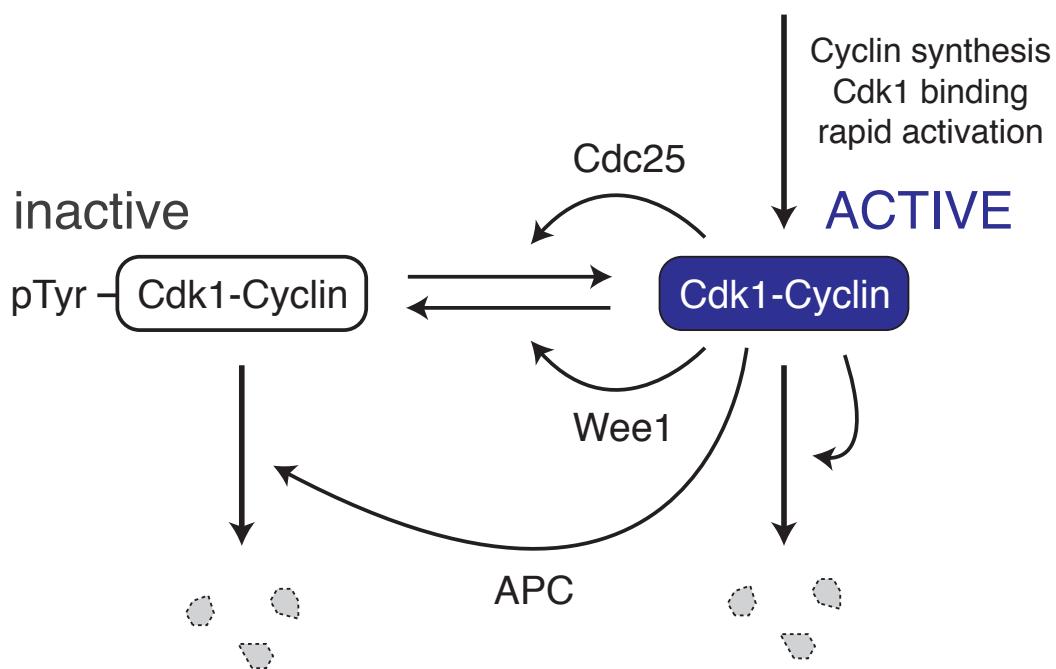
Looking at the bifurcation diagram you constructed above, choose an intermediate value of Cyclin for which the circuit exhibits bistable dynamics.

First, consider the parameter  $f_{inact}$ , which controls the strength of feedback through Wee1 and effectively controls the maximum rate of Cdk1 inactivation. Use flux balance analysis, ask: For what range of values of  $f_{inact}$  does the circuit continue to exhibit bistable dynamics?

Now consider a reduced circuit in which there is no feedback through Wee1, i.e the rate of Cdk1 inactivation is controlled by a constant parameter  $k_{inact}$ , as in Equation 1. Again use flux balance analysis to ask: For what range of values of  $k_{inact}$  does the circuit continue to exhibit bistable dynamics? What version of the circuit tolerates a larger range of variation in Cdk1 inactivation rates?

## Session 3

### From switch-like transitions to free-running oscillations: Adding negative feedback.



**Figure 3.1** The full circuit with negative feedback

## 22 Session 3 From switch-like transitions to free-running oscillations: Adding negative feedback.

So far we have considered Cyclin levels as an external input or control variable. In real cells, Cyclin synthesis and degradation are coupled to Cdk1 activity: As Cyclin builds up above a threshold, Cdk1 becomes active; active Cdk1 in turn promotes APC-dependent degradation of Cyclin, causing Cyclin levels to fall rapidly again. Figure 3.1 again is a schematic view of the Cyclin/Cdk1 circuit with this additional negative feedback.

Could switch-like activation of Cdk1 and negative feedback in which active Cdk1 promotes APC-dependent Cyclin degradation lead to robust free-running oscillations in Cyclin and Cdk1 activity? Again, we can use simple mathematical models to explore how this works. Lets start with a simple thought experiment.

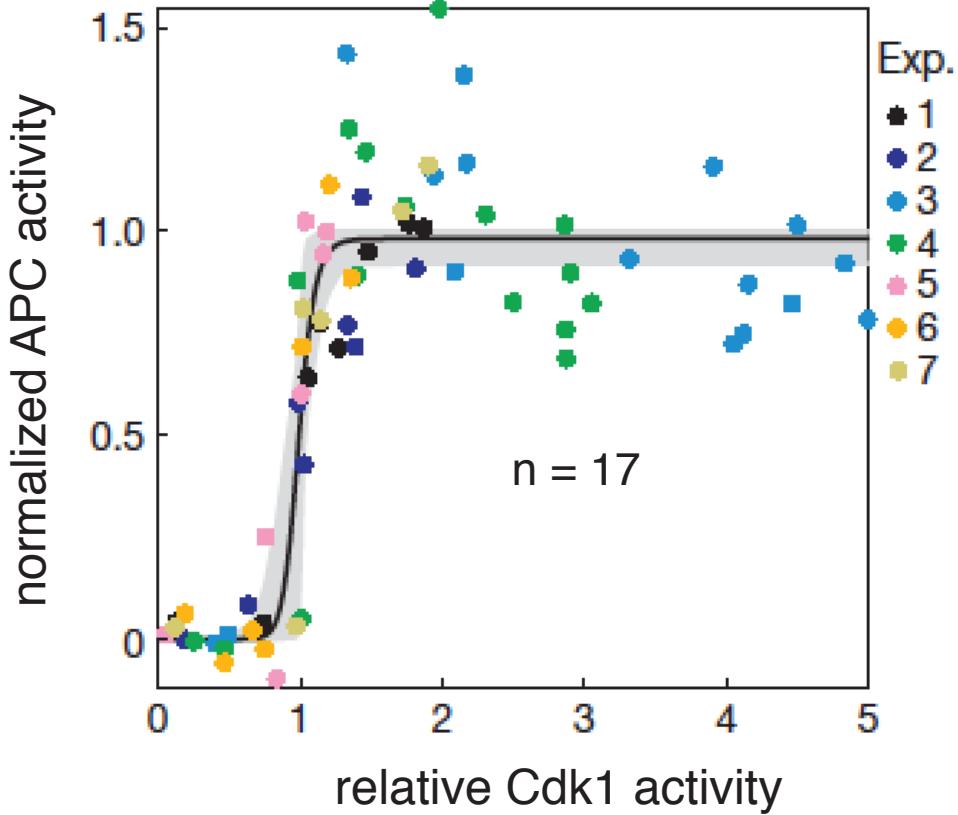
### Exercise 3.1

**A simple thought experiment:** Consider the bifurcation diagram and the response curves for steady state Cdk1 vs Cyclin that you constructed in Part 2. Discuss with your partners how negative feedback in which Cdk1 activates APC and Cyclin degradation could drive the rise and fall of Cyclin levels and Cdk1 activity.

Now lets test your intuition by adding negative feedback to our circuit model, based on the arrows in Figure 5 above. We need to write an additional equation for the rate of change in Cyclin due to synthesis and degradation:

$$\frac{dC}{dt} = k_{syn} - k_{deg}(A) \cdot C \quad (3.1)$$

Here, we have written  $k_{deg}(A)$  to reflect the fact that the degradation rate is a function of APC activity which in turn depends on the level of Cdk1 activity. Recent experimental measurements from the James Ferrells lab show that, like Cdc25 and Wee1, APC activity is a switch-like function of Cdk1 activity, which is well-fit by a Hill function with Hill coefficient  $n = 17$ :



**Figure 3.2** Dependence of APC activity on Cdk1

Again, we can represent the dependence of Cyclin degradation rates on Cdk1 using a function of the form:

$$k_{deg}(A) = a_{deg} + f_{deg} \frac{A^{n_{deg}}}{K_{deg}^{n_{deg}} + A^{n_{deg}}} \quad (3.2)$$

where  $a_{deg} = 0.01$ ,  $f_{deg} = 0.04$ ,  $K_{deg} = 32$ , and  $n_{deg} = 17$  are estimated from experiments.

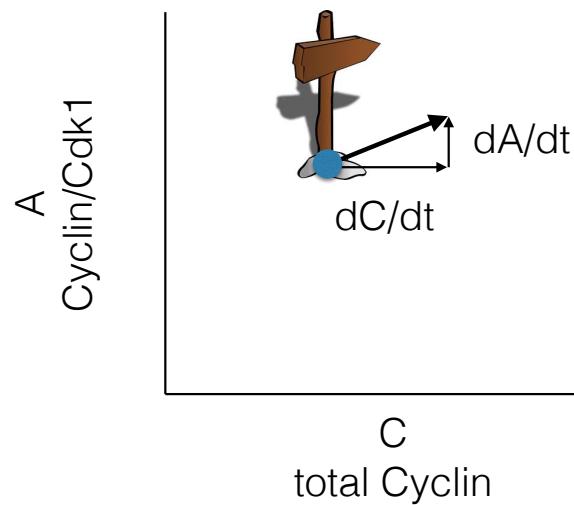
Plugging into Equation 3.1, and combining with our previous equation for active Cdk1 ( $A$ ), we now have a pair of differential equations:

**24** Session 3 From switch-like transitions to free-running oscillations: Adding negative feedback.

$$\begin{aligned}\frac{dA}{dt} &= \left(0.16 + 0.8 \frac{A^{11}}{35^{11} + A^{11}}\right) \cdot (C - A) - \left(0.08 + 0.4 \frac{A^{3.5}}{30^{3.5} + A^{3.5}}\right) \cdot A \\ \frac{dC}{dt} &= k_{synth} + \left(0.01 + 0.04 \frac{A^{17}}{32^{17} + A^{17}}\right) A\end{aligned}\quad (3.3)$$

Again, we can analyze the dynamics this system using a combination of graphical and numerical approaches:

### 3.1 Graphical



**Figure 3.3** A phase plane

The state of this system is described by two values: the concentration of Cyclin and the concentration of active Cdk1. We can think of the state of the system, at any moment in time, as being defined by a point in a 2D "phase plane". The position of this point on the A-axis indicates how much active Cyclin/Cdk1 there is in the cell at that moment, and the position along the C-axis indicates the total amount of Cyclin. The differential equations for  $dC/dt$  and  $dA/dt$  above are rules that tell you, given the current values of A and C, how much the system will move in the C direction and A direction in the next instant in time. We can think of these rules as arrows attached to each point in the 2D plane, together constituting what we call a flow field. The flow along these arrows defines what will happen to the system.

Analyzing the behavior of a system with two dynamical variables in the phase plane is more difficult than the 1D examples from before. More kinds of behavior are possible, but we can still use graphical tools to think intuitively about what will happen.

First, we need to load a library that defines some useful tools for graphical "phase plane analysis"

```
1 ## Loading the 'phaseR' library
2 library(phaseR)
```

Now lets define a function that computes the right hand sides of our pair of differential equations:

```
1  RHS_2D <- function(t, y, parameters) {
  with(as.list(parameters), {
    C = y[1]
    A = y[2]
    dc=k_syn-(a_deg + f_deg*A^n_deg/(K_deg^n_deg + A^n_deg))*C
    pda = (a_act+f_act*A^n_act/(K_act^n_act+A^n_act))*(C-A)
    nda = (a_inact+f_inact*A^n_inact/(K_inact^n_inact+A^n_inact))
      *A
    return(list(c(dc,pda-nda)))
  })
10 }
```

The arguments to this function are the current time t, a list y containing the current values of C (y[1]) and A (y[2]) and a list of parameter values. The function returns a list of two values corresponding to  $dC/dt$  and  $dA/dt$

Finally, let's define the list of parameters and assign their values:

```
2 full_pars = list(k_syn=1.0,a_deg=0.01,f_deg=0.04,K_deg=32,n_deg=17,a_act
=0.16,f_act=0.8,K_act=35,n_act=11,a_inact=0.08,f_inact=0.4,K_inact
=30,n_inact=3.5)
```

**26** Session 3 From switch-like transitions to free-running oscillations: Adding negative feedback.

OK, now we can use a predefined function (part of the phaseR library) that computes and displays the flow field:

```
2   flowField(RHS_2D, x.lim = c(0, 100),  
3             y.lim = c(0, 100),  
4             xlab="C", ylab="A",  
5             parameters = full_pars,  
6             points = 15, add = FALSE)
```

Question: What can you infer about the dynamics of this circuit by looking at the flow field?

Recall that for the 1D system, it was very useful to know where the steady states are. How do we accomplish this in 2D? Instead of plotting positive and negative fluxes and locating steady states where these two curves cross, we will instead plot two nullclines” and locate steady states where these cross. The A nullcline is the set of (A,C) value pairs for which  $dA/dt = 0$ , and the C nullcline is the set of (A,C) value pairs for which  $dC/dt = 0$ . Where these cross, both  $dA/dt$  and  $dC/dt$  must be 0. To think intuitively about the meaning of the nullclines, notice that when the system is on the A-nullcline, the next instantaneous step must be in the C direction (and vice versa). Thus, the arrows in the flow field must point along the C direction on the A-nullcline.

For this particular system of equations, you could find the A and C nullclines algebraically with pencil and paper. Feel free to do that now if you really want to show off your algebra chops. However, if you just want to get the nullclines quickly with minimal effort, you can use this built in R function to plot the nullclines over the flowfield you plotted a moment ago:

```
2   nullclines(FHN, x.lim = c(0, 100),  
3             y.lim = c(0, 100),  
4             parameters = full_pars,  
5             points = 500)
```

### Exercise 3.2

**Question:** How many steady states are there? Do you think they are stable or unstable? How could you tell?

### 3.2 Numerical

The short answer to the above question is that you cannot determine if a steady state is stable or unstable just by looking at the shapes of the nullclines (except in very special cases). A very straightforward (and quite powerful) way to investigate the dynamics of this circuit is to solve the equations numerically for different starting values of A and C and then plot the solutions in the A/C plane.

```
| trajectory(FHN, y0 = c(10,10), t.end = 500,
| parameters = full_pars)
```

### Exercise 3.3

Repeat this for a series of different starting values. Does the long term behavior of this circuit depend on the initial conditions? If so, how?

### 3.3 A closer look at the limit cycle dynamics

Plot A(t) and C(t) vs time. Does the system go to a steady state? If a system does not approach a steady state point, but instead stably cycles through the same states over and over, we say that the system is attracted to a "limit cycle". Relate the time series to the limit cycle view in the phase plane. Infer that there is slow movement along the upper and lower branches of the Cyc nullcline and fast jumps between branches.

A mathematical model of the cell cycle like this of course simplifies many molecular details and leaves out some of the biology. The power of a description like this is that it now allows to ask which features of the molecular circuit are crucial for the observed behavior. Now that you have assembled these tools, you can computationally manipulate the cell cycle circuit to see how the behavior changes.

### 3.4 Challenges

**Challenge:** What is the importance of switch-like dependence of Cyclin degradation on Cdk1?

Set  $n = 1$  in the differential equation for Cyc and in the equation for the Cyc nullcline.

Again plot nullclines and plot solutions for a variety of different initial conditions. What is the behavior of this system?

Is it possible to get oscillatory dynamics to return by tuning e.g. the threshold or strength of negative feedback?

**Challenge:** How tolerant is the cell cycle oscillator to changes in the threshold for activation of Cyclin degradation by Cdk1? How does this tolerance change as you vary the steepness of the feedback (i.e. the exponent  $n$ )?

**28** Session 3 From switch-like transitions to free-running oscillations: Adding negative feedback.

**Challenge:** Explore what happens as you vary Cyclin synthesis and degradation rates. How does the period of the oscillation change? Why?

**Challenge:** The original form of the circuit had two feedback loops on active Cyclin/Cdk1: one acted on the phosphatase Cdc25, and one acted on the kinase Wee1. If you computationally remove the feedback loop involving Wee1, what happens to oscillations? Can you get a different answer if you change the parameters describing the feedback loop through Cdc25? It may help to go back to the single differential equation describing  $dA/dt$  and reanalyze it without the Wee1 feedback loop to look for multiple steady states.