NGS Course Nove Hrady Documentation

Release 1.0

Libor Morkovsky, Vaclav Janousek

January 23, 2017

This course aims to introduce the participants to UNIX - an interface that is one of the most convenient options for working with big (textual) data. Special attention is put on working with Next Generation Sequencing (NGS) data. Most of the NGS data formats were designed to be textual, and textual data is where UNIX excels in its versatility. By combining basic UNIX tools one can achieve results that would require finding specialized software for each of the tasks in other environments.

Not knowing 'the right way' to do things can be intimidating for the beginning users, so an additional section exploring the 'best practice' is available for self learning.

Schedule:

Friday (14:00-18:00)	
Afternoon I	Course Intro, Introduction to Unix
Afternoon Break (~ 0.5h)	
Afternoon II	Basic Unix
Saturday (9:00-18:00)	
Morning	Introduction to Genomics, Advanced Unix I
Lunch (~ 1.5h)	
Afternoon I	Advanced Unix II
Afternoon Break (~ 0.5h)	
Afternoon II	Data Visualization
Sunday (9:00-18:00)	
Morning	Introduction to NGS
Lunch (~ 1.5h)	
Afternoon I	Unix tools for genomics
Afternoon Break (~ 0.5h)	
Afternoon II	Exercise

Initial instructions:

CHAPTER 1

Installation instructions

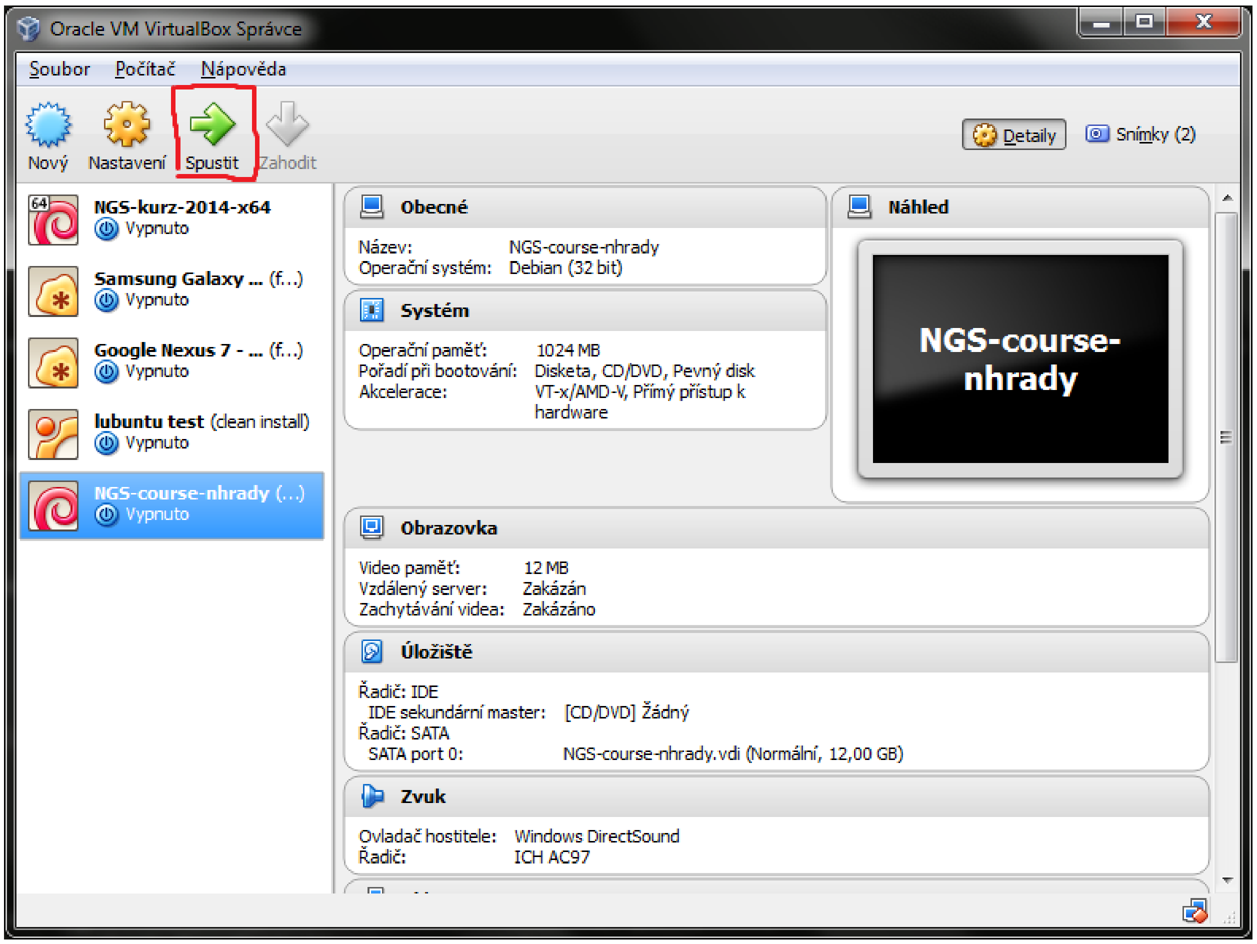
We will be using a virtual computer pre-installed with Debian Linux and sample data necessary for the exercises.

Note: You need to install the image even if your main system is Linux / Mac OS X!

Installation steps (it should take about 10 minutes):

- Install VirtualBox (https://www.virtualbox.org/wiki/Downloads). It works on Linux and Mac too.
- Download the virtual machine image from this link: https://goo.gl/ugMHB0 You'll get a single file with `.ova` extension on your hard drive.
- You can either double click the `.ova` file, or run VirtualBox, and choose `File > Import Appliance`. Follow the instructions after the import is started.

After successful installation you should see something like this (only the machine list will contain just one machine). Check whether you can start the virtual machine: click `Start` in the main VirtualBox window:



Obecné

```
Název:              NGS-course-nhrady
Operační systém:    Debian (32 bit)
```

Systém

```
Operační paměť:       1024 MB
Pořadí při bootování: Disketa, CD/DVD, Pevný disk
Akcelerace:           VT-x/AMD-V, Přímý přístup k
                      hardware
```

Obrazovka

```
Video paměť:           12 MB
Vzdálený server:       Zakázán
Zachytávání videa:     Zakázáno
```

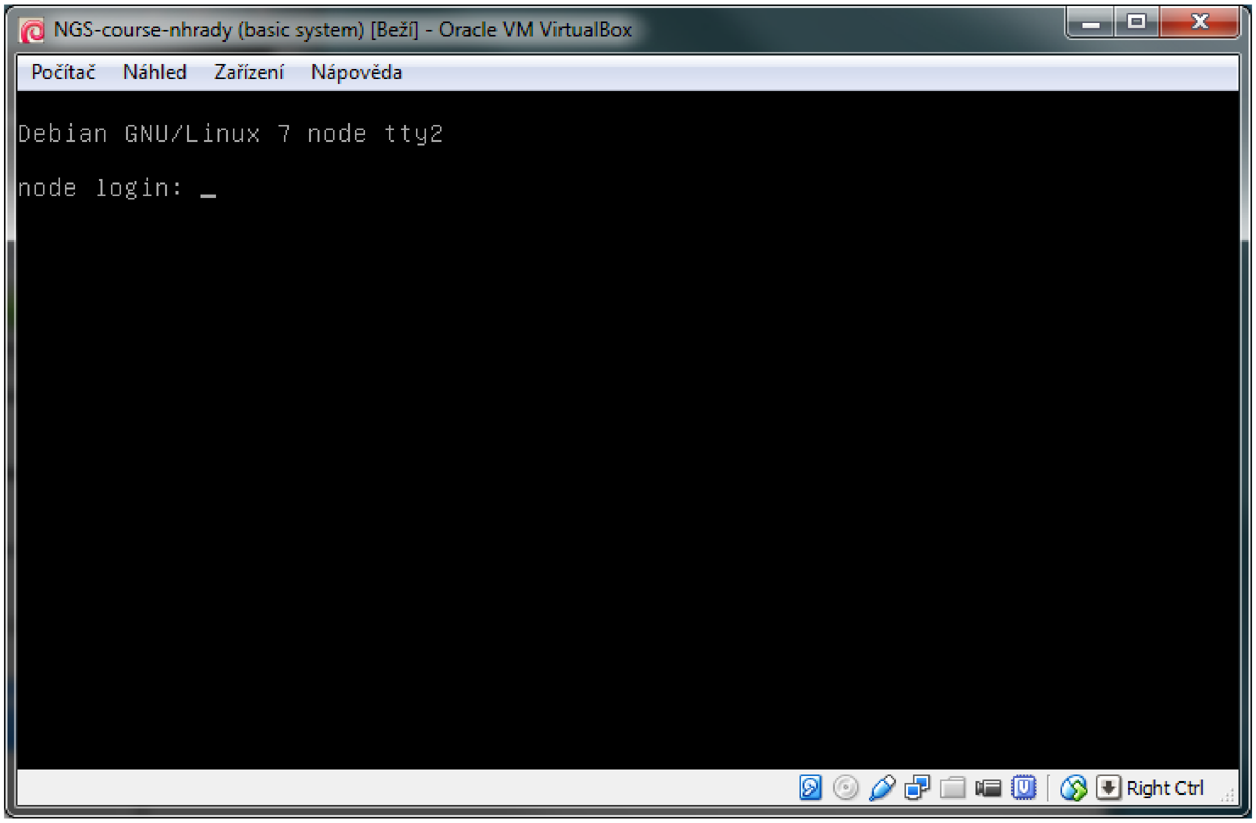
Úložiště

```
Řadič: IDE
  IDE sekundární master:   [CD/DVD] Žádný
Řadič: SATA
  SATA port 0:             NGS-course-nhrady.vdi (Normální, 12,00 GB)
```

Zvuk

```
Ovladač hostitele:   Windows DirectSound
Řadič:               ICH AC97
```

After a while you should see something like this:


NGS-course-nhrady (basic system) [Běží] - Oracle VM VirtualBox
Počítač Náhled Zařízení Nápověda
Right Ctrl

```
Debian GNU/Linux 7 node tty2

node login: _
```

You don't need to type anything into that window, just checking that it looks like the screen shot is enough.

1.1 How to access the machine

Because it is much more comfortable to use a native terminal application than the small VM screen, you will connect to the machine depending on what system you are using.

Machine configuration details:

- Administrative user: *root*, password: *debian*
- Normal user: *user*, password: *user*
- ssh on port 2222
- RStudio on port 8787

In case of any problems try to find contact the tutors, we'll try to resolve all problems before the course.

1.2 Windows

Install PuTTY and WinSCP. PuTTY will be used to control the virtual computer. WinSCP will be used to transfer files between your computer and the virtual computer.

- PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html - look for putty.exe)
- WinSCP (http://winscp.net/eng/download.php - look for Installation package).

1.3 Mac OS X and Linux

Ssh is used to control the virtual computer. It should be installed in your computer.

Files can be transferred with `scp`, `rsync` or `lftp` (recommended) from the command line. *Scp* and *rsync* could be already installed in your system, if you want to use *lftp*, you'll probably have to install it yourself.

Mac users that prefer grapical clients can use something like *CyberDuck*. See http://apple.stackexchange.com/questions/25661/whats-a-good-graphical-sftp-utility-for-os-x .

1.4 Time to log in!

Try to log in following the instructions in *Connect to control the machine*.

CHAPTER 2

Connecting to the virtual machine

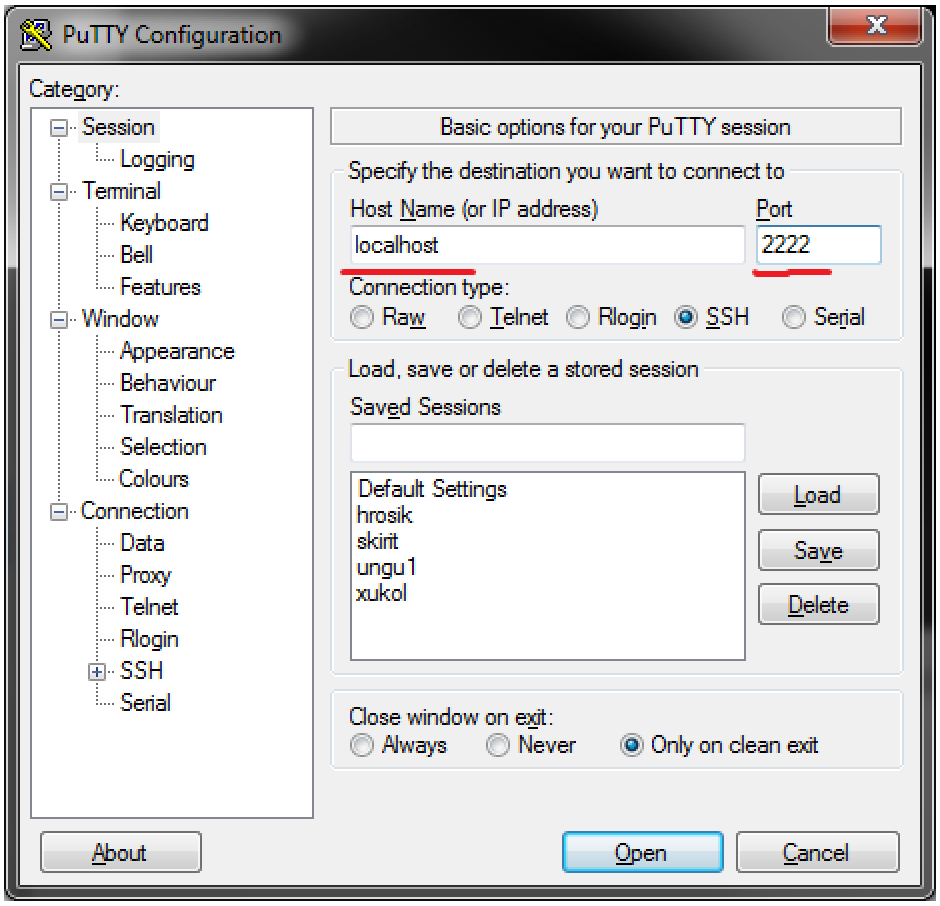
Note: You need to start the virtual machine first!

2.1 Connect to control the machine

To control the machine, you need to connect to the ssh service. This is also referred to as 'logging in'.

In Windows this is done with PuTTY.

- start PuTTY
- fill Host Name: `localhost`
- fill Port: `2222`
- click Open or press <Enter>



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): localhost

Port: 2222

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

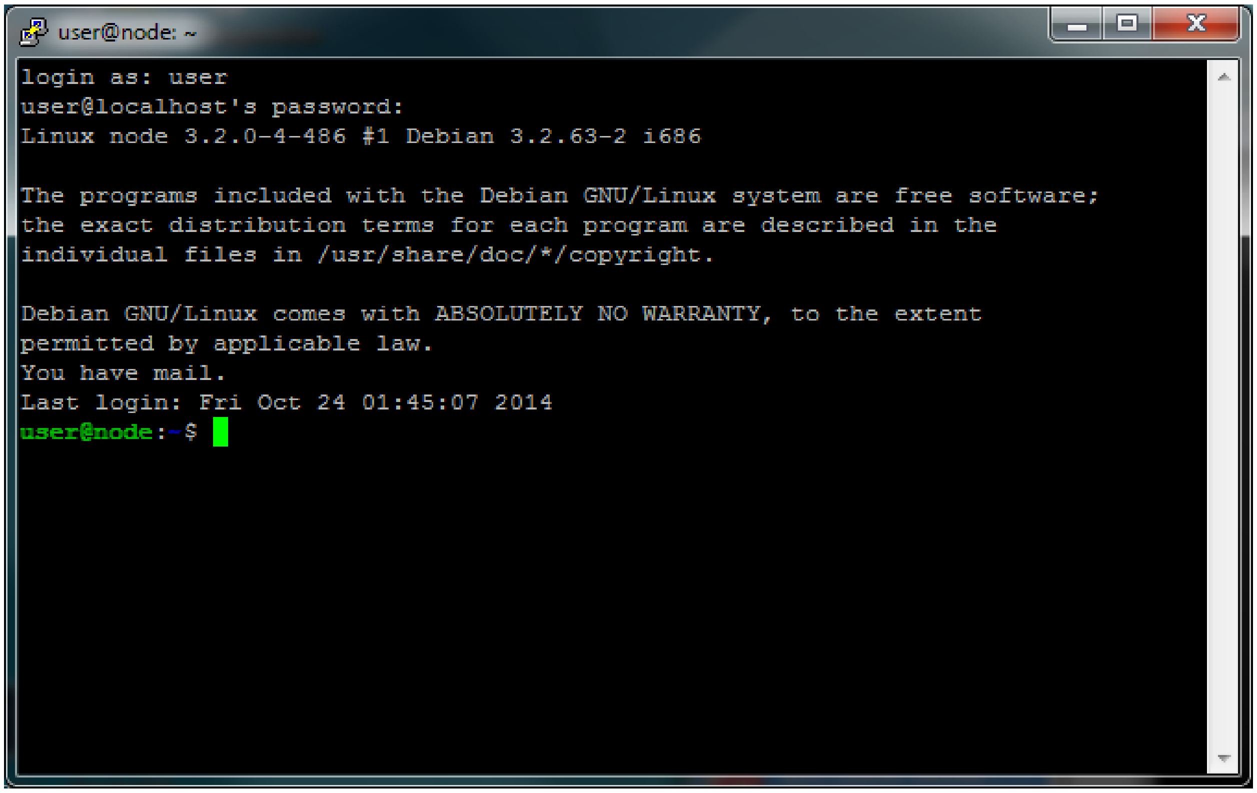
Saved Sessions

Default Settings
hrosik
skirit
ungu1
xukol

Close window on exit: Always Never [✓ Only on clean exit]

In the black wnidow that appears, type your credentials:

- login as: `user`
- user@localhost's password: `user`



```
login as: user
user@localhost's password:
Linux node 3.2.0-4-486 #1 Debian 3.2.63-2 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have mail.
Last login: Fri Oct 24 01:45:07 2014
user@node:~$
```

In Mac OS X or Linux, you start your terminal program first (‘Terminal’, ‘Konsole’, ‘xterm’). In the terminal window your shell is running (probably ‘bash’). Here you use ssh to connect to the virtual machine:

```
ssh -p 2222 user@localhost
```

2.2 Testing the Internet connection

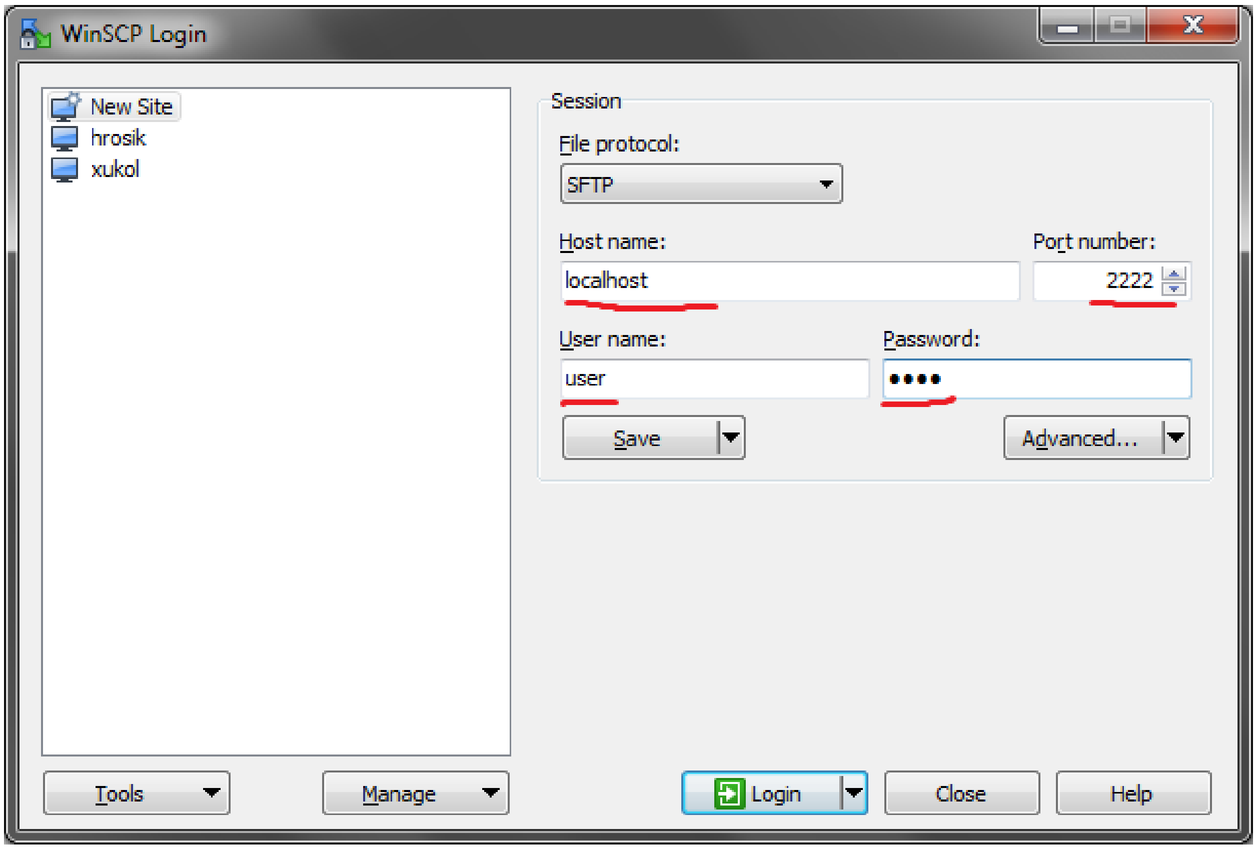
When you’re logged in, check your internet connection from the virtual machine. Your main computer has to be connected to the internet, of course. Copy the following command, and paste it to the command prompt (click right mouse button in PuTTY window).

```
wget -q -O - http://goo.gl/n8XK2Y | head -1
# <!DOCTYPE html>
```

If the `<!DOCTYPE html>` does not appear, something is probably wrong with the connection.

2.3 Connect to copy files

In Windows, WinSCP is used to copy files to Linux machines. You use the same information as for PuTTY to log in.



Session

File protocol: SFTP

Host name: localhost

Port number: 2222

User name: user

Password: ••••

In Mac OS X or Linux, the most simple command to copy a file into a home directory of `user` on a running virtual machine is:

```
scp -P 2222 myfile user@localhost:~
```

2.4 Connect to RStudio

This is the easiest one, just click this link: Open RStudio. Login with the same credentials (user, user).

Course contents:

Etherpad with 'live' code.

Unix - Basics

This session will give you all the basics that you need to smoothly move around when using a UNIX system (in the text mode!).

3.1 Basic orientation

3.1.1 Check your keyboard

Before we do any serious typing, make sure you know where are the important keys. I'd suggest using English keyboard, if you don't want to constantly press right alt and five random letters before you find the one you need. You will definitely need those keys:

```
[] - squared brackets
{} - curly brackets
<> - angle brackets (smaller-than, bigger-than sign)
() - parentheses
~ - tilde
/ - slash
\ - back slash
| - pipe
^ - caret
$ - dollar sign
: - colon
; - semicolon
. - dot
, - comma
# - hash
_ - underscore
- - dash
* - asterisk
! - exclamation mark
? - question mark
& - ampersand
@ - at sign
'' - quotation mark single
"" - quotation mark double
```

3.1.2 Directory structure

Unlike 'drives' in MS Windows, UNIX has a single directory tree that starts in `/` (called root directory). Everything can be reached from the root directory. The next important directory is `~` (called user's home directory). It is a shortcut for `/home/user` here, `/home/..your login name..` in general.

Your bash session has a *working directory* that can be changed with `cd` (change directory) and printed with `pwd` (print working directory). All filenames and paths you type refer to your working directory (relative paths), unless you start them with `/` (absolute paths).

Try the following commands in the order they are provided, and figure out what they do. Then use your knowledge to explore the directory structure of the virtual machine.

Figure out what these commands do:

```
pwd
ls
ls /
ls ..
ls ~
cd
cd /
cd ..
cd ~
```

A neat trick to go back where you've been before the last `cd` command:

```
cd -
```

More in *Moving around & manipulation with files and directories*.

3.1.3 Helpful commands (dir content and its size, disc usage)

```
ls -shaR # list content of a directory
du -sh # disc usage (by directory)
df -h # disc free
ls | wc -l # what does this command do?
locate # find a file/program
```

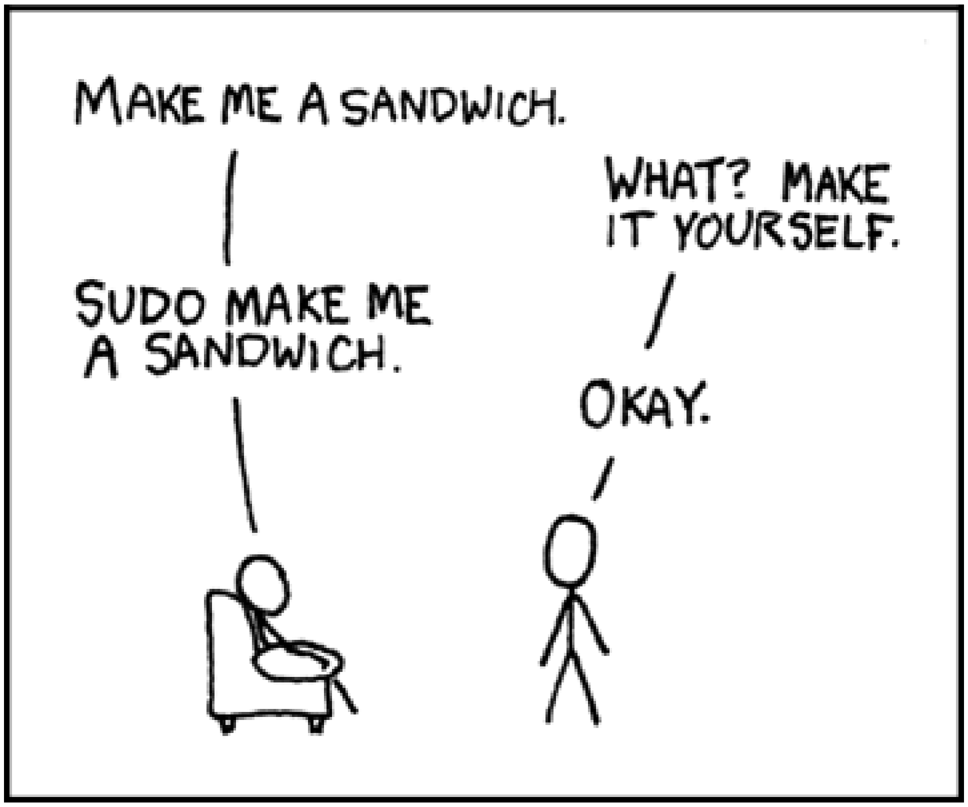
3.1.4 Moving or copying files and directories

```
touch # make a file
mkdir -p # make a directory (``-p`` makes missing directory above)
rm -r # remove a file/directory
mv # move a file/directory
cp -r # copy a file/directory
ln -s # make a symbolic link
```

Prepare data directory in your HOME directory and copy FASTQ data from common repository:

```
cd ~
mkdir -p data/fastq
sudo cp -r /data/fastq/fastq.tar.gz data/fastq/.
cd data/fastq
ls
```

Note: Normal users cannot change (and break) the (UNIX) system. There is one special user in each system called `root`, who has the rights to make system wide changes. You can either directly log in as root, or use `sudo` (super user do) to execute one command as `root`.



3.1.5 Uncompressing files

```
# Compressed tarball archives
tar -xzvf fastq.tar.gz

# gzipped files
gunzip file.txt.gz
```

3.1.6 Viewing plain text file content

```
less -SN
tail -n 5
head -n 5
cat
nano
```

Try these commands:

```
cd ~
head -n 8 HRTMUOC01.RL12.00.fastq
tail -n 8 HRTMUOC01.RL12.00.fastq
```

3.1.7 Pipes

Using the | (pipe) character you instruct the shell to take the output of the first command and use it as an input for the second command.

The complement to `head` is `tail`. It displays last lines of the input. It can be readily combined with `head` to show the second sequence in the file.

```
head -8 HRTMUOC01.RL12.00.fastq | tail -4 | less -S

# or the third sequence data ;)
< HRTMUOC01.RL12.00.fastq head -8 | tail -4 | less -S
```

Exercise (How many reads are there?):

We found out that FASTQ files have a particular structure (four lines per read). To find the total number of reads in our data, we will use another tool, `wc` (stands for *word count*, not for a toilet at the end of the pipeline;). `wc` counts words, lines and characters.

Our data is in three separate files. To merge them on the fly we'll use another tool, `cat` (for conCATenate). `cat` takes a list of file names and outputs a continuous stream of the data that was in the files (there is no way to tell where one file ends from the stream).

# now double click on each file name in the listing, # and click right mouse button to paste (insert space in between)

```
cat HRTMUOC01.RL12.00.fastq | wc -l
```

The number that appeared is four times the number of sequences (each sequence takes four lines). And there is even a built-in calculator in bash:

```
echo $(( 788640 / 4 ))
expr XXXX / 4
```

3.1.8 Globbing

Imagine you've got 40 FASTQ files instead of 3. You don't want to copy and paste all the names! There is a feature that comes to rescue. It's called *globbing*. It allows you to specify more filenames at once by defining some common pattern. All your read files have `.fastq` extension. `*.fastq` means *a file named by any number of characters followed by '.fastq'*.

```
cat HRTMUOC01.RL12.*.fastq | wc -l
expr XXXX / 4

cat HRTMUOC01.RL12.0?.fastq | wc -l
expr XXXX / 4
```

3.1.9 Producing lists

What do these commands do?

```
touch file-0{1..9}.txt file-{10..20}.txt
touch 0{1..9}-{a..f}.txt {10..12}-{a..f}.txt
touch 0{1..9}-{jan,feb,mar}.txt {10..12}-{jan,feb,mar}.txt
```

Exercise:

Program runs 20 runs of simulations for three datasets (hm, ss, mm) using three different sets of values: small (sm), medium sized (md) and large (lg). There are three groups of output files, which should go into subdirectory A, B and C. Make a directory for each dataset-set of parameters-run-subdirectory. Count the number of directories.

Producing lists of subdirectories

```
mkdir -p {2013..2015}/{A..C}
mkdir -p {2013..2015}/0{1..9}/{A..C} {2013..2015}/{10..12}/{A..C}
```

3.1.10 Variables & Loops

```
CPU=4
echo $CPU

FILE=data/fastq/HRTMUOC01.RL12.00.fastq
echo $FILE

FILES=`ls ~/data/fastq/*.fastq`
echo $FILES
```

```
list=`ls ~/data/fastq/HRTMUOC01.RL12.0{1..9}.fastq`

for i in $list
do
  echo $i
done

for i in $list
do
  head -n1 $i | wc -c
done
```

3.2 Use multiple windows (and be safe when the network fails)

First, type `screen` in your terminal:

```
screen
```

Screen creates the first window for you. To create another one press `ctrl+a c`. To switch between the windows press `ctrl+a space`.

Note: Keyboard shortcuts notation: `ctrl+a space` means press `ctrl` key and `a` key simultaneously and `space` key after you release both of the previous keys.

3.3 Installing software

The easiest way to install software is via a package manager (eg. `apt-get` for all Debian variants). When the required software is not in the repositories, or one needs the latest version, it's necessary to take the more difficult path. The canonical UNIX way is:

```
wget -O - ..url.. | tar xvz   # download and unpack the 'tarball' from internet
cd ..unpacked directory..     # set working directory to the project directory
./configure                   # check your system and choose the way to build it
make                          # convert source code to machine code (compile it)
sudo make install             # copy the results to your system
```

3.3.1 htop

Installing software from common repository:

```
sudo apt-get install htop
```

3.3.2 Bedtools

Install software which is not in the common repository. You just need to find a source code and compile it:

```
wget https://github.com/arq5x/bedtools2/releases/download/v2.25.0/bedtools-2.25.0.tar.gz
tar -zxvf bedtools-2.25.0.tar.gz
cd bedtools2
make
```

Another common place where you find a lot of software is *GitHub*. We'll install `bedtools` from a GitHub repository:

```
cd ~/sw

# get the latest bedtools
git clone https://github.com/arq5x/bedtools2
```

This creates a *clone* of the online repository in `bedtools2` directory.

```
cd bedtools2
make
```

3.4 Exercise

Note:

1. What is the output of this command `cd ~/ && ls | wc -l`?

1. The total count of files in subdirectories in home directory
2. The count of lines in files in home directory
3. The count of files/directories in home directory
4. The count of files/directories in current directory

2. How many directories this command `mkdir {1999-2001}-{1st,2nd,3rd,4th}-{1..5}` makes?

1. 56
2. 60
3. 64
4. 72

3. When files created using this command `touch file0{1..9}.txt file{10..30}.txt`, how many files matched by `ls file?.txt` and `ls file*0.txt`

1. 30 and 0
2. 0 and 30
3. 30 and 4
4. 0 and 3

4. Which file would match this pattern `ls *0?0.*`?

1. file36500.tab
2. file456030
3. 5460230.txt
4. 456000.tab

5. Where do we get with this command `cd ~/ && cd ../..`?

1. two levels below home directory
2. one level above home directory
3. to root directory
4. two levels above root directory

6. What number does this command `< file.txt head -n10 | tail -n+9 | wc -l` print? (Assume the file.txt is not empty)

1. 0
2. 1
3. 2
4. 3

CHAPTER 4

Unix - Advanced I

This session focuses on plain text file data extraction/modification using built-in Unix tools.

4.1 Pattern search & regular expressions

`grep` is a useful tool to search for patterns using a mini-language called **regular expressions**.

```
^A          # match A at the beginning of line
A$          # match A at the end of line
[0-9]       # match numerical characters
[A-Z]       # match alphabetical characters
[ATGC]      # match A or T or G or C
.           # match any character
A*          # match A letter 0 or more times
A\{2\}      # match A letter exactly 2 times
A\{1,\}     # match A letter 1 or more times
A+          # match A letter 1 or more times (extended regular expressions)
A\{1,3\}    # match A letter at least 1 times but no more than 3 times
AATT\|TTAA  # match AATT or TTAA
\s          # match whitespace (also TAB)
```

Use mouse annotation file (GTF)

```
cd ~
cp /data/mus_mda/05-fst2genes/Mus_musculus.NCBIM37.67.gtf.gz ~/data
# does not work, let's investigate

ll /data/mus_mda/05-fst2genes

# the permissions are not right..fix it
sudo chmod o+r /data/mus_mda/Mus_musculus.NCBIM37.67.gtf.gz

gunzip data/Mus_musculus.NCBIM37.67.gtf.gz
less -S data/Mus_musculus.NCBIM37.67.gtf
```

Note: You can check file permissions by typing `ll` instead of `ls`. `rwx` stand for *Read*, *Write*, *eXecute*, and are repeated three times, for *User*, *Group*, and *Others*. The two names you see next to the permissions are file's owner user and group.

You can change the permissions - if you have the permission to do so - by e.g. `chmod go+w` - "add write permission to group and others".

1. Count the number of records on the chromosome X

```
< data/Mus_musculus.NCBIM37.67.gtf grep '^X' | wc -l
```

2. Count the number of records on chromosome X and Y

```
< data/Mus_musculus.NCBIM37.67.gtf grep '^[XY]' | wc -l
< data/Mus_musculus.NCBIM37.67.gtf grep '^X\|^Y' | wc -l
```

3. Count the number of 'CDS' on the chromosome X

```
< data/Mus_musculus.NCBIM37.67.gtf grep 'CDS' | grep '^X' | wc -l
```

Use nightingale variant call file (VCF)

```
cd ~
cp /data/vcf_examples/luscinia_vars_flags.vcf.gz ~/data
gunzip data/luscinia_vars_flags.vcf.gz
less -S data/luscinia_vars_flags.vcf
```

1. Count the number variants in the file

```
< data/luscinia_vars_flags.vcf grep -v '^#' | wc -l
```

2. Count the number of variants passing/failing the quality threshold

```
< data/luscinia_vars_flags.vcf grep -v '^#' | grep 'PASS' | wc -l
< data/luscinia_vars_flags.vcf grep -v '^#' | grep 'FAIL' | wc -l
```

3. Count the number of variants on the chromosome Z passing the quality threshold

```
< data/luscinia_vars_flags.vcf grep -v '^#' | grep 'PASS' | grep '^chrZ\s' | wc -l
```

4.2 Cutting out, sorting and replacing text

We are going to use these commands: `cut`, `sort`, `uniq`, `tr`, `sed`.

Note: `sed` (text Stream EDitor) can do a lot of things, however, pattern replacement is the best thing to use it for. The 'sed language' consists of single character commands, and is no fun to code and even less fun to read (what does `sed 'h;G;s/\n//'` do?;). Use `awk` for more complex processing.

General syntax:

```
sed 's/pattern/replacement/'

# Replace one or more A or C or G or T by N
sed 's/^[AGCT]\{1,\}/N/'

# The same thing using extended regular expressions:
sed -r 's/^[AGCT]+/N/'
```

Use nightingale variant call file (VCF)

1. Which chromosome has the highest and the least number of variants?

```
< data/luscinia_vars_flags.vcf grep -v '^#' | cut -f 1 |
sort | uniq -c | sed 's/^ \{1,\}//' | tr " " "\t" | sort -k1,1nr

# sed -r (extended regular expressions)
< data/luscinia_vars_flags.vcf grep -v '^#' | cut -f 1 |
sort | uniq -c | sed -r 's/^ +//' | tr " " "\t" | sort -k1,1nr
```

2. What is the number of samples in the VCF file?

```
< data/luscinia_vars_flags.vcf grep -v '^##' | head -n1 |
cut --complement -f 1-9 | tr "\t" "\n" | wc -l
```

Figure out alternative solution for exercise 2.

Note: Difference between `sed` and `tr`:

`tr` (from TRansliterate) replaces (or deletes) individual characters: Ideal for removing line ends (`tr -d "\n"`) or replacing some separator to TAB (`tr ";" "\t"`).

`sed` replaces (or deletes) complex patterns.

4.3 Joining multiple files + subshell

Use `paste`, `join` commands.

Note: Shell substitution is a nice way to pass a pipeline in a place where a file is expected, be it input or output file (Just use the appropriate sign). Multiple pipelines can be used in a single command:

```
cat <( cut -f 1 file.txt | sort -n ) <( cut -f 1 file2.txt | sort -n ) | less
```

Use nightingale FASTQ file

1. Join all nightingale FASTQ files and create a TAB separated file with one line per read

```
# repeating input in paste causes it to take more lines from the same source
cat *.fastq | paste - - - - | cut -f 1-3 | less
```

2. Make a TAB-separated file having four columns:

   (a) chromosome name

   (b) number of variants in total for given chromosome

   (c) number of variants which pass

   (d) number of variants which fails

```
# Command 1
< data/luscinia_vars_flags.vcf grep -v '^#' | cut -f 1 |
sort | uniq -c | sed 's/^ \{1,\}//' | tr " " "\t" > data/count_vars_chrom.txt

# Command 2
< data/luscinia_vars_flags.vcf grep -v '^#' | cut -f 1,7 | sort -r |
uniq -c | sed 's/^ \{1,\}//' | tr " " "\t" | paste - - |
cut --complement -f 2,3,6 > data/count_vars_pass_fail.txt
```

```
# Command 3
join -1 2 -2 3 data/count_vars_chrom.txt data/count_vars_pass_fail.txt | wc -l

# How many lines did you retrieved?

# You have to sort the data before sending to ``join`` - subshell
join -1 2 -2 3 <( sort -k2,2 data/count_vars_chrom.txt ) \
<( sort -k3,3 data/count_vars_pass_fail.txt ) | tr " " "\t" > data/count_all.txt
```

All three commands together using subshell:

```
# and indented a bit more nicely
IN=data/luscinia_vars_flags.vcf
join -1 2 -2 3 \
    <( <$IN  grep -v '^#' |
      cut -f 1 |
      sort |
      uniq -c |
      sed 's/^ \{1,\}//' |
      tr " " "\t" |
      sort -k2,2 ) \
    <( <$IN grep -v '^#' |
      cut -f 1,7 |
      sort -r |
      uniq -c |
      sed 's/^ \{1,\}//' |
      tr " " "\t" |
      paste - - |
      cut --complement -f 2,3,6 |
      sort -k3,3  ) |
  tr " " "\t" \
> data/count_all.txt
```

4.4 Exercise

4.4.1 How many bases were sequenced?

`wc` can count characters (think bases) as well. But to get a reasonable number, we have to get rid of the other lines that are not bases.

One way to do it is to pick only lines comprising of letters A, C, G, T and N. There is a ubiquitous mini-language called *regular expressions* that can be used to define text patterns. *A line comprising only of few possible letters* is a text pattern. `grep` is the basic tool for using regular expressions:

```
cat *.fastq | grep '^[ACGTN]*$' | less -S
```

Check if the output looks as expected. This is a very common way to work - build a part of the pipeline, check the output with `less` or `head` and fix it or add more commands.

Now a short explanation of the `^[ACGTN]*$` pattern (`grep` works one line a time):

- `^` marks beginning of the line - otherwise `grep` would search anywhere in the line
- the square brackets (`[]`) are a *character class*, meaning one character of the list, `[Gg]rep` matches `Grep` and `grep`
- the `*` is a count suffix for the square brackets, saying there should be zero or more of such characters

- `$` marks end of the line - that means the whole line has to match the pattern

To count the bases read, we extend our pipeline:

```
cat *.fastq | grep '^[ACGTN]*$' | wc -c
```

The thing is that this count is not correct. `wc -c` counts every character, and the end of each line is marked by a special character written as `\n` (n for newline). To get rid of this character, we can use another tool, `tr` (transliterate). `tr` can substitute one letter with another (imagine you need to lowercase all your data, or mask lowercase bases in your Fasta file). Additionally `tr -d` (delete) can remove characters:

```
cat *.fastq | grep '^[ACGTN]*$' | tr -d "\n" | wc -c
```

Note: If you like regular expressions, you can hone your skills at http://regex.alf.nu/.

CHAPTER 5

Unix - Advanced II

Scripting session:

- scripting in one line (or more;) using `awk`
- writing functions and scripts in shell
- speeding up your processing by running in parallel

5.1 Scripting in one line (awk)

`awk` is most often used instead of `cut`, when the fields are separated by spaces and padded - `awk` can ignore the whitespace. `awk` can also be used when one wants to reorder the columns: `echo " 1 3 5" | awk '{print $2, $3, $1}'`.

And now for something completely different - *Use the nightingale FASTQ files:*

1. Extract IDs of a FASTQ file and count the number of reads

```
< data/fastq/HRTMUOC01.RL12.00.fastq awk '{ if( (NR + 3) % 4 == 0 ){ print $0 } }' | wc -l
```

2. Make a file with read ID and read lengths in one line

```
< data/fastq/HRTMUOC01.RL12.00.fastq \
awk 'BEGIN{
    OFS="\t"
}{
    if(( NR + 3 ) % 4 == 0 ){
        id = $0
    }else{
        if( (NR + 3) % 4 == 1 ){
            print id,length($0)
        }
    }
}' | less
```

3. Get average read length

```
< data/fastq/HRTMUOC01.RL12.00.fastq \
awk 'BEGIN{
    OFS="\t"; l=0; n=0
}{
    if( ( NR + 3 ) % 4 == 1 ){
        l = l + length($0);
        n = n + 1;
    }
}END{
    print "Average read length:", l/n
}'
```

4. Filter out short sequences (set the minimum size allowed)

```
< data/fastq/HRTMUOC01.RL12.00.fastq \
awk -v l=80 '{
  if( (NR + 3) % 4 == 0 ){
    id=$0;
  }else if( (NR + 3) % 4 == 1 ){
    seq=$0;
  }else if( (NR + 3) % 4 == 2 ){
    q=$0;
  }else{
    if( length(seq) >= l ){
      print id"\n"seq"\n"q"\n+";
    }
  }
}' | less
```

5.2 Functions in Shell

Create a command `uniqt` that will behave as `uniq`, but there will be no padding (spaces) in front of the numbers, and numbers will be separated by <tab>, so eg. `cut` will work.

Do not use the same name as the original command, otherwise you'll create an endless loop.

```
uniqt() { uniq -c | sed -r 's/^ *([0-9]+) /\1\t/'  ;}
```

5.3 Shell Scripts

```
nano script.sh
```

Make a script `filter_fastq.sh` which reads a FASTQ file, filters out short sequences and saves to a file named `$INPUT-filtered`:

```
#!/bin/bash

FILE=$1
LENGTH=$2
OUT=$1-filtered

< $FILE awk -v l=$LENGTH '{
    if( (NR + 3) % 4 == 0 ){
        id=$0;
    }else if( (NR + 3) % 4 == 1 ){
        seq=$0;
    }else if( (NR + 3) % 4 == 2 ){
        q=$0;
    }else{
        if( length(seq) >= l ){
            print id"\n"seq"\n"q"\n+";
        }
    }
}' > $OUT

echo File `basename $FILE` done
```

To run the script:

```
chmod +x filter_fastq.sh
# check with ls, filter_fastq.sh should be green now
# and using ll you should see the 'x' (eXecutable) permission
./filter_fastq.sh data/fastq/HRTMUOC01.RL12.00.fastq 80

# or, without a need for the shebang line (#!) in the file
# and without +x permission
bash filter_fastq.sh data/fastq/HRTMUOC01.RL12.00.fastq 80
```

5.4 Parallel

Runs one instance of the command per each CPU in your machine. Regretably your **virtual** machine has only one CPU, so this won't help much. But modern machines do have four and more CPUs, and then it really helps.

Do control the number of jobs (`-j`) only when sharing the machine with someone, or when you're sure that yout task is IO bound. Otherwise `parallel` does a good job choosing the number of tasks to run for you.

```
parallel 'bash script.sh {} > {}.out' ::: {1..10}
```

Run the `filter_fastq.sh` in parallel:

```
parallel 'bash filter_fastq.sh {} 80' ::: data/fastq/*.fastq
```

There is a lot of magic to be done with `{.}`, `{/}`, `{#}` placeholders, check `man parallel`. If your data is a single file, but the processing of one line is not dependent on the other lines, `split` will help.

CHAPTER 6

Graphics session

A picture is worth a thousand words.

Especially when your data is big. We'll try to show you one of the easiest ways to get nice pictures from your UNIX. We'll be using R, but we're not trying to teach you R. R Project is huge, and mostly a huge mess. We're cherry picking just the best bits;)

6.1 Summarization

R is best for working with 'tables'. That means data, where each line contains the same amount of 'fields', delimited by some special character like `;` or `<tab>`. The first row can contain column names. VCF is almost a nice tabular file. The delimiter is `<tab>`, but there is some mess in the beginning of the file:

```
</data/mus_mda/00-popdata/popdata_mda.vcf.gz zcat | less -S
```

6.1.1 Prepare the input file

Our input dataset is huge, and we want to use only some subset of the animals. Let's choose few european individuals. They have `RDS, KCT, MWN` and `BAG` in their names. Each programmer is lazy and prefers to avoid mistakes by letting the machine do the work - let's find the right columns with unix tools.

```
# we'll be reusing the same long file name, store it into a variable
IN=/data/mus_mda/00-popdata/popdata_mda.vcf.gz

# create a new 'project' directory in data
mkdir -p ~/projects/plotvcf/data
cd ~/projects/plotvcf

# check the file once again, and find the obligatory VCF columns
<$IN zcat | less -S

# let the computer number the fields..
<$IN zcat |
  grep -v '^##' |
  head -1 |
  tr "\t" "\n" |
  nl |
  less

# it's 1-9
# now find the sample columns according to the pattern
<$IN zcat |
  grep -v '^##' |
  head -1 |
  tr "\t" "\n" |
  nl |
  grep -e RDS -e KCT -e MWN -e BAG |
  awk '{print $1}' |
  paste -sd,

# it's 67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85

# now decompress and subset the data with cut
<$IN zcat |
  cut -f1-9,67-85 \
> data/popdata_mda_euro.vcf
```

We want to get rid of the comment lines starting with `##`, and keep the line starting with `#` as column names (getting rid of the `#` itself):

```
IN=data/popdata_mda_euro.vcf

# get rid of the '##' lines (quotes have to be there, otherwise
# '#' means a comment in bash)
<$IN grep -v '##' | less -S

# better formatted output
<$IN grep -v '##' | column -t | less -S

# it takes too much time, probably column -t is reading too much of the file
# we're good with first 1000 lines
<$IN grep -v '##' | head -1000 | column -t | less -S

# good, now trim the first #
<$IN grep -v '##' | tail -c +2 | less -S

# all looks ok, store it (tsv for tab separated values)
# skip the very first character with tail
<$IN grep -v '##' | tail -c +2 > data/popdata_mda_euro.tsv
```

Now we will switch to R Studio. You can just click here: Open RStudio.

In R Studio choose `File > New file > R Script`. R has a working directory as well. You can change it with `setwd`. Type this code into the newly created file:

```
setwd('~/projects/plotvcf')
```

With the cursor still in the `setwd` line, press `ctrl+enter`. This copies the command to the console and executes it. Now press `ctrl+s`, and save your script as `plots.R`. It is a better practice to write all your commands in the script window, and execute with `ctrl+enter`. You can comment them easily, you'll find them faster than in `.Rhistory`...

6.1.2 Load and check the input

Tabular data is loaded by `read.table` and it's shorthands. On a new line, type `read.table` and press `F1`. Help should pop up. We'll be using the `read.delim` shorthand, that is preset for loading `<tab>` separated data with US decimal separator:

```
d <- read.delim('data/popdata_mda_euro.tsv')
```

A new entry should show up in the 'Environment' tab. Click the arrow and explore. Click the 'd' letter itself.

You can see that `CHROM` was encoded as a number only and it was loaded as `integer`. But in fact it is a factor, not a number (remember e.g. chromosome X). Fix this in the `read.delim` command, loading the data again and overwriting *d*. The plotting would not work otherwise:

```
d <- read.delim('data/popdata_mda_euro.tsv', colClasses=c("CHROM"="factor"))
```

6.1.3 First plot

We will use the `ggplot2` library. The 'grammatical' structure of the command says what to plot, and how to represent the values. Usually the `ggplot` command contains the reference to the data, and graphic elements are added with `+ geom_..()`. There are even some sensible defaults - e.g. `geom_bar` of a factor sums the observations for each level of the factor:

```
library(ggplot2)
ggplot(d, aes(CHROM)) + geom_bar()
```

This shows the number of variants in each chromosome. You can see here, that we've included only a subset of the data, comprising chromosomes 2 and 11.

6.1.4 Summarize the data

We're interested in variant density along the chromosomes. We can simply break the chromosome into equal sized chunks, and count variants in each of them as a measure of density.

There is a function `round_any` in the package `plyr`, which given precision rounds the numbers. We will use it to round the variant position to 1x10^6 (million base pairs), and then use this rounded position as the block identifier. Because the same positions repeat on each chromosome, we need to calculate it once per each chromosome. This is guaranteed by `group_by`. `mutate` just adds a column to the data.

You're already used to pipes from the previous exercises. While it's not common in R, it is possible to build your commands in a similar way thanks to the `magrittr` package. The name of the package is an homage to the Belgian surrealist René Magritte and his most popular painting.

Although the magrittr `%>%` operator is not a pipe, it behaves like one. You can chain your commands like when building a bash pipeline:

```
library(plyr)
library(dplyr)

# 'bash-like' ordering (source data -> operations -> output)
d %>%
  group_by(CHROM) %>%
  mutate(POS_block=round_any(POS, 1e6)) ->
  dc

# the above command is equivalent to
dc <- mutate(group_by(d, CHROM), POS_block=round_any(POS, 1e6))
```

Now you can check how the `round_any` processed the `POS` value. Click the `dc` in the **Environment** tab and look for `POS_block`. Looks good, we can go on. The next transformation is to count variants (table rows) in each block (per chromosome): You can use `View` in **R Studio** instead of `less` in bash.

Fig. 6.1: Ceci n’est pas une pipe. This is not a pipe.

```
dc %>%
 group_by(CHROM, POS_block) %>%
 summarise(nvars=n()) %>%
 View
```

Note: To run the whole block at once with `ctrl+enter`, select it before you press the shortcut.

If the data look like you expected, you can go on to plotting:

```
dc %>%
  group_by(CHROM, POS_block) %>%
  summarise(nvars=n()) %>%
  ggplot(aes(POS_block, nvars)) +
    geom_line() +
    facet_wrap(~CHROM, ncol = 1)
```

Now you can improve your plot by making the labels more comprehensible:

```
dc %>%
  group_by(CHROM, POS_block) %>%
  summarise(nvars=n()) %>%
  ggplot(aes(POS_block, nvars)) +
    geom_line() +
    facet_wrap(~CHROM, ncol = 1) +
    ggtitle("SNP denisty per chromosome") +
    ylab("number of variants") +
    xlab("chromosome position")
```

If you prefer bars instead of a connected line, it's an easy swap with ggplot.

```
dc %>%
  group_by(CHROM, POS_block) %>%
  summarise(nvars=n()) %>%
  ggplot(aes(POS_block, nvars)) +
    geom_bar(stat="identity") +
    facet_wrap(~CHROM, ncol = 1) +
    ggtitle("SNP denisty per chromosome") +
    ylab("number of variants") +
    xlab("chromosome position")
```

The `stat="identity"` is there, because `geom_bar` counts the rows otherwise. This could have saved us some more typing - but takes longer to compute:

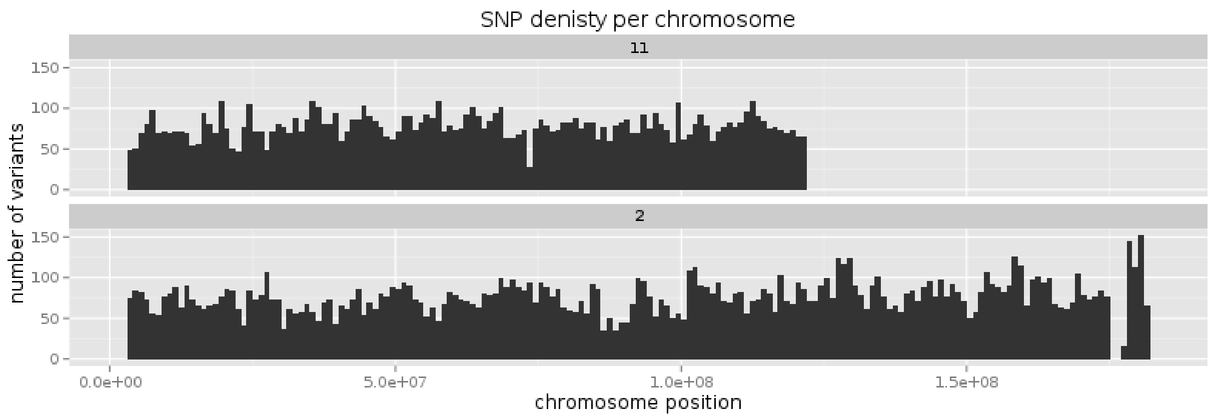
```
ggplot(d, aes(POS)) +
  geom_histogram() +
  facet_wrap(~CHROM, ncol = 1) +
  ggtitle("SNP denisty per chromosome") +
  ylab("number of variants") +
  xlab("chromosome position")
```

`ggplot` warned you in the **Console**:

```
stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```

You can use `binwidth` to adjust the width of the bars, setting it to 1x10^6 again:

```
ggplot(d, aes(POS)) +
  geom_histogram(binwidth=1e6) +
  facet_wrap(~CHROM, ncol = 1) +
  ggtitle("SNP denisty per chromosome") +
  ylab("number of variants") +
  xlab("chromosome position")
```



6.2 Tidy data

To create plots in such a smooth way like in the previous example the data has to loosely conform to some simple rules. In short - each column is a variable, each row is an observation. You can find more details in the Tidy data paper. There is an R package `tidyr` that helps you to get the data into the required shape.

The vcf is *tidy* when using the `CHROM` and `POS` variables. Each variant (SNP) is a row. The data is not tidy regarding variants in particular individuals. Individual identifier is a variable for this case, but it is stored as column name. This is not 'wrong', this format was chosen so the data is smaller. But it does not work well with `ggplot`.

Now if we want to look at genotypes per individual, we need the genotype as a single variable, not 18. `gather` takes the values from multiple columns and gathers them into one column. It creates another column where it stores the originating column name for each value.

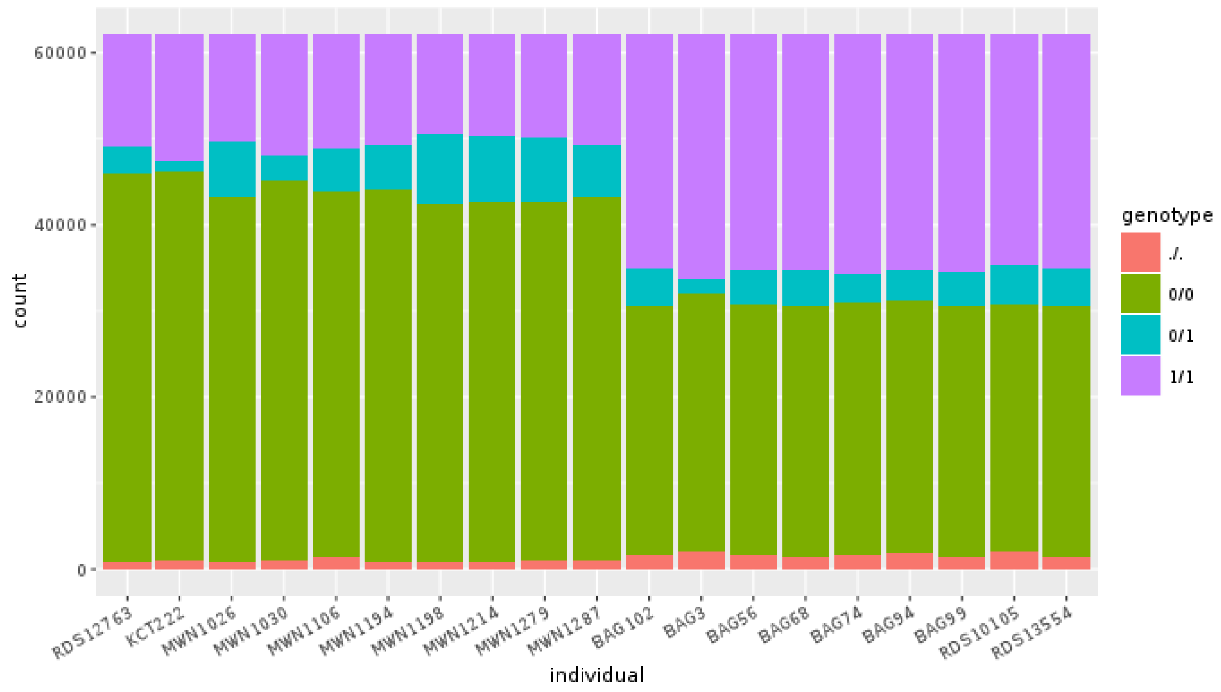
```
library(tidyr)
dm <- d %>% gather(individual, genotype, 10:28 )
```

Look at the data. Now we can plot the counts of reference/heterozygous/alternative alleles.

```
ggplot(dm, aes(individual, fill=genotype)) + geom_bar()
```

Again, most of the code is spent on making the plot nicer:

```
ggplot(dm, aes(individual, fill=genotype)) +
  geom_bar() +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```



Now try to change parts of the command to see the effect of various parts. Delete `, fill=genotype` (including the comma), execute. A bit boring. We can get much more colours by colouring each base change differently:

```
# base change pairs, but plotting sometnihg else than we need (probably)
ggplot(dm, aes(individual, fill=paste(REF, ALT))) + geom_bar()
```

What could be interesting is the transitions to transversions ratio, for each individual:

```
# transitions are AG, GA, CT, TC
# transversions is the rest
transitions <- c("A G", "G A", "C T", "T C")
dm %>%
  mutate(vartype = paste(REF, ALT) %in% transitions %>% ifelse("Transition", "Transversion")) %>%
  ggplot(aes(individual, fill=vartype)) +
  geom_bar()

# works a bit, but not what we expected
# now count each homozygous ref as 0,
# heterozygous as 1 and homozygous alt as 2
# filter out uncalled
dm %>%
  filter(genotype != './.') %>%
  mutate(vartype = paste(REF, ALT) %in% transitions %>% ifelse("Transition", "Transversion"),
         score = ifelse(genotype == '0/0', 0, ifelse(genotype == '0/1', 1, 2))) %>%
  group_by(individual, vartype) %>%
  summarise(score=sum(score)) %>%
  spread(vartype, score) %>%
  mutate(TiTv=Transition / Transversion) %>%
  ggplot(aes(individual, TiTv)) +
  geom_point() +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```

CHAPTER 7

Read quality

Many steps of genomic data processing have some associated quality value for their results. Here we will briefly check the first and last of those. But there is no simple way to set your quality thresholds. You have to recognize completely bad data. But after that there is a continuum. Sometimes you just need an idea of the underlying biology. Find some variants for further screening. Sometimes you're trying to pinpoint particular variant causing a disease.

Each read that comes out of the (now common) sequencing machines like Illumina or Ion Torrent has a quality score assigned with each of the bases. This is not true for the upcoming NanoPore or almost forgotten SOLiD machines, that are reading more bases a time.

7.1 Phred encoding

The quality in Fastq files is encoded in Phred quality score, a number on a logarithmic scale, similar to decibels.

Phred quality	Probability of error
20	1 in 100
40	1 in 10,000
60	1 in 1,000,000

Each Phred number is in turn encoded as a single character, so there is straightforward mapping between the bases and the quality scores. The most common mapping is `Phred+33`:

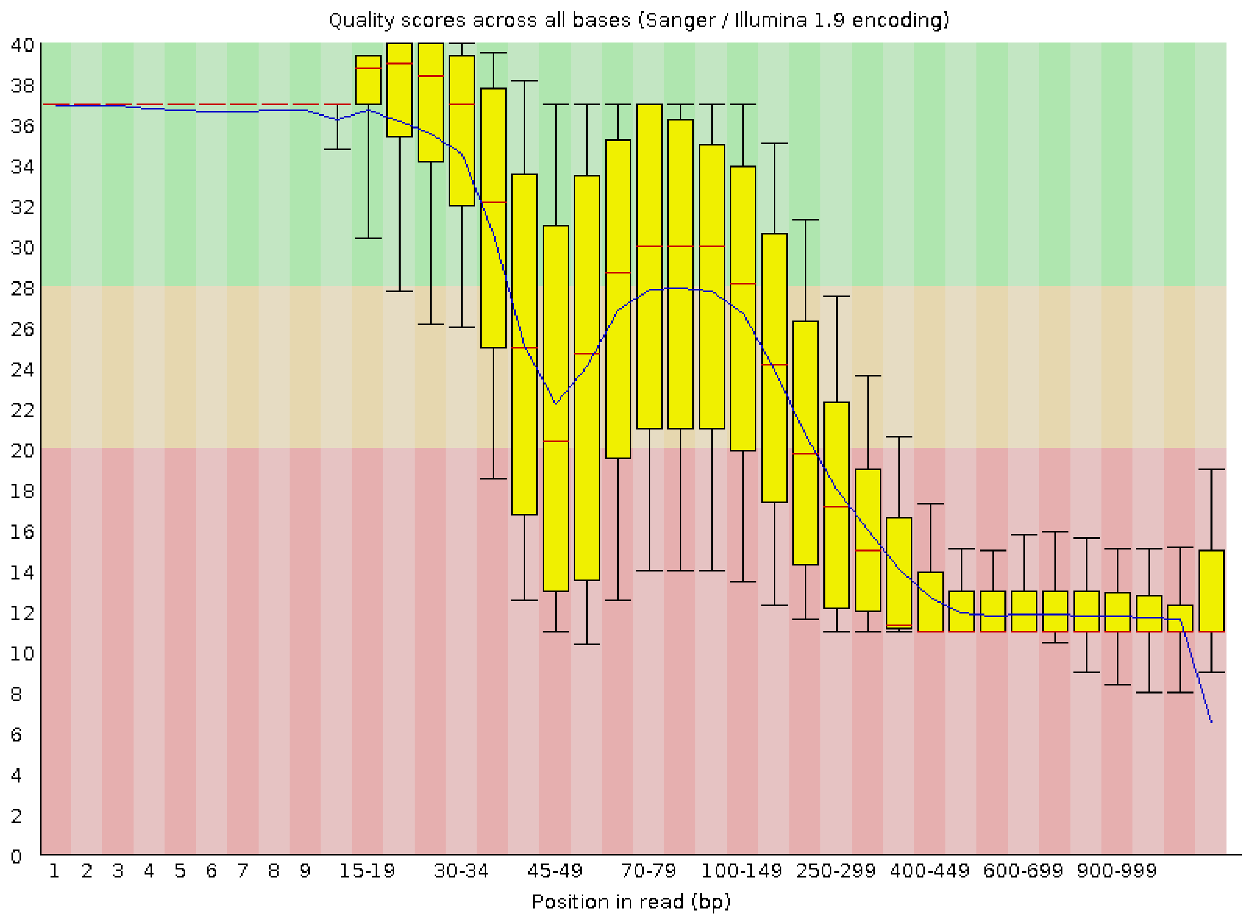
```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ
| |                         |    |         |
0.2......................26...31........41
```

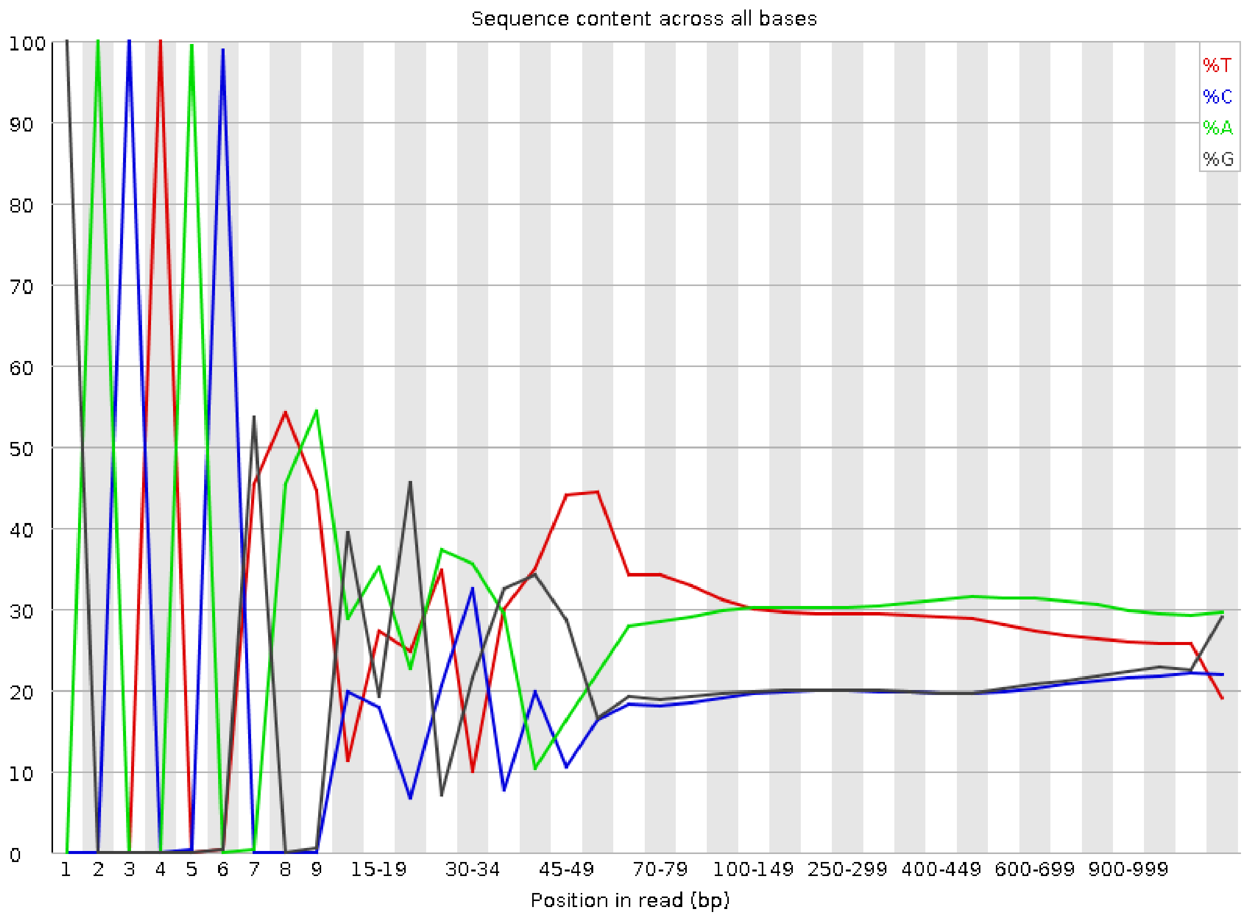
7.2 FastQC

FastQC is a nice tool that you run with your data and get nice graphical reports on the quality. It is not completely installed in your images, so you can try to run a tool that was just unpacked (this is how this tool is distributed, there is no install script - a daily bread of a bioinformatician;). You have to use the full path to the tool to run it:

```
# make a project directory for the qualities
cd
mkdir -p projects/quality
cd projects/quality
ln -s ~/data/fastq 00-reads

mkdir 01-quality
~/sw/FastQC/fastqc -o 01-quality --noextract 00-reads/HRTMUOC01.RL12.01.fastq
```

Now transfer the `.html` files from the virtual machine to yours. Open the files on your machine. You should see a report with plots like this:





7.3 Parsing Fastq and decoding Phred

To understand better what is in the FastQC plots, we will try to reproduce the plots using UNIX and ggplot. You should be able to understand the following pipeline, at least by taking it apart with the help of head or less. A brief description:

- `sed` replaces the leading '@' with an empty string in the first of every four lines and deletes the third of every four lines (the useless '+' line)
- `paste` merges every three lines
- `awk` selects only reads longer than 50 bases
- `head` takes first 1,000 sequences
- `awk` creates a Phred decoding table, then uses it to decode the values, outputs one row for each base (see 'tidy data')

```
mkdir 02-quality-handmade
IN=00-reads/HRTMUOC01.RL12.01.fastq

<$IN sed '1~4s/^@//;3~4d' |
  paste - - - |
  awk 'length($2) > 50' |
  head -1000 |
  awk 'BEGIN{OFS="\t";
         for(i=33;i<127;i++) quals[sprintf("%c", i)] = i - 33;
       }
       {
         l = length($2)
         for(i=1;i<=l;i++) {
           print $1, i, l - i, substr($2, i, 1), quals[substr($3, i, 1)];}
       }'\
> 02-quality-handmade/quals.tsv
```

7.4 Quality by position

The first of the FastQC plots shows a summary of base qualities according to position in the read. But it does not show quality scores for all possible positions, they are grouped into classes of similar importance. The further the base in the read, the bigger the group.

Fire up R Studio by clicking the link. Set your working directory to the directory with the quality data (in **Console**, don't forget to use tab completion):

```
setwd('~/projects/quality')
```

Now create a file where your plotting code will live, `File > New file > R Script`, then save it as `plots.R`. First we will read in the data.

```
d <- read.delim("02-quality-handmade/quals.tsv",
                col.names=c("seq", "pos", "end_pos", "base", "qual"),
                header=F)
```

We did not include column names in the data file, but it is easy to provide them during the load via `col.names` argument. Let's look at base quality values for first 10 sequences:

```
library(ggplot)
library(dplyr)

sel <- levels(d$seq)[1:10]
ggplot(d %>% filter(seq %in% sel), aes(pos, qual, colour=seq, group=seq)) + geom_line()
```

The qualities on sequence level don't seem to be very informative. They're rather noisy. A good way to fight noise is aggregation. We will aggregate the quality values using boxplots and for different position regions. First set up the intervals:

```
# fastqc uses bins with varying size:
# 1-9 by one, up to 75 by 5, up to 300 by 50, rest by 100

breaks <- c(0:9, seq(14, 50, by=5), seq(59, 100, by=10), seq(149, 300, by=50), seq(400, 1000, by=100

# create nice labels for the intervals
labs <- data.frame(l=breaks[1:length(breaks)-1], r=breaks[2:length(breaks)]) %>%
  mutate(diff=r-l, lab=ifelse(diff > 1, paste0(l+1, "-", r), as.character(r)))
```

Check the `breaks` and `labs` variables. In the FastQC plot there are vertical quality zones, green, yellow and red. To replicate this, we need the values of the limits:

```
# data for quality zones
quals <- data.frame(ymin=c(0, 20, 28), ymax=c(20, 28, 40), colour=c("red", "orange", "green"))

# check if the quality zones look reasonably
```

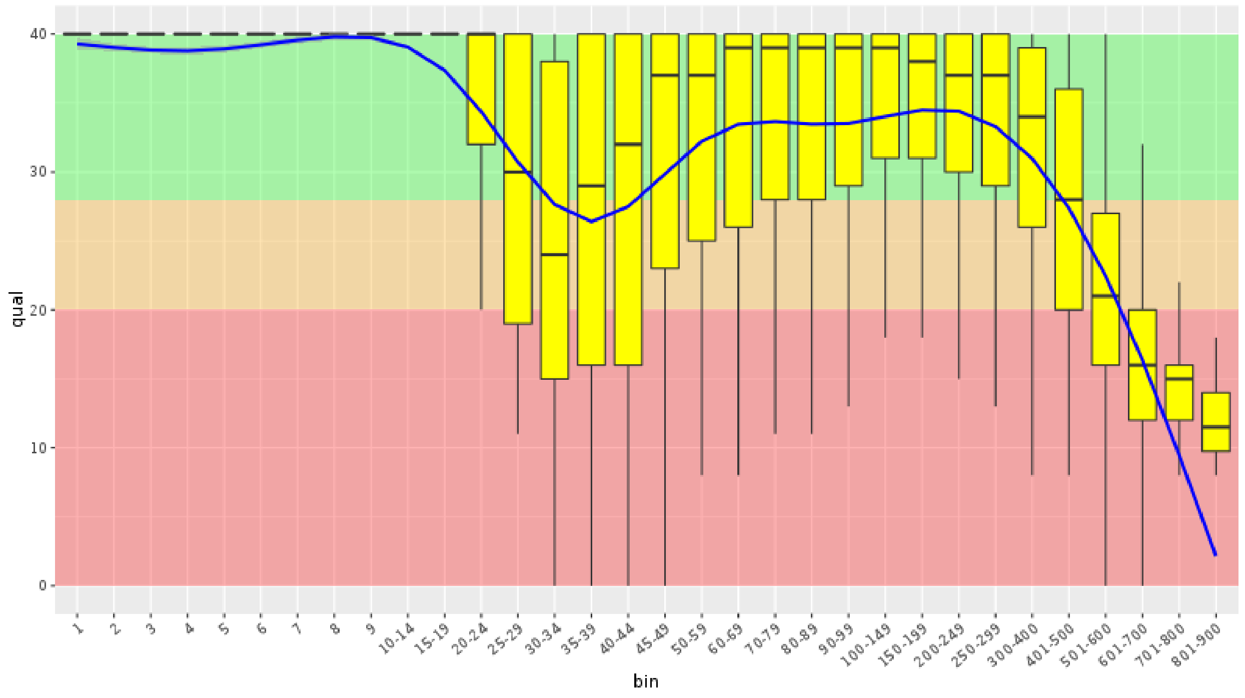
```
ggplot(quals, aes(ymin=ymin, ymax=ymax, fill=colour)) +
  geom_rect(alpha=0.3, xmin=-Inf, xmax=Inf) +
  scale_fill_identity() +
  scale_x_discrete()
```

Now we can use the breaks to create position bins:

```
dm <- d %>% mutate(bin=cut(pos, breaks, labels=labs$lab))

# plot the qualities in the bins
ggplot(dm, aes(bin, qual)) +
  geom_boxplot(outlier.colour=NA) +
  ylim(c(0, 45))
```

Zones and boxplots look ok, we can easily combine those two into one plot. That's pretty easy with ggplot. We use `theme` to rotate the x labels, so they're all legible. In real world application the qualities are binned first, and then the statistics are calculated on the fly, so it is not necessary to load all the data at once.

```
ggplot(dm) +
  geom_rect(xmin=-Inf, xmax=Inf, data=quals, aes(ymin=ymin, ymax=ymax, fill=colour), alpha=0.3) +
  scale_fill_identity() +
  geom_boxplot(aes(bin, qual), outlier.colour=NA, fill="yellow") +
  geom_smooth(aes(bin, qual, group=1), colour="blue") +
  theme(axis.text.x=element_text(angle = 40, hjust = 1))
```



Now we can do the base frequency plot. We already have the position bins, so just throw ggplot at it:

```
ggplot(dm, aes(bin, fill=base)) + geom_bar()
```

We're almost there, just need to normalize the values in each column so they sum up to 1. Ggplot can do it for us:

```
ggplot(dm, aes(bin, fill=base)) + geom_bar(position="fill")
```

It's possible to rearrange the legend by reordering levels of the factor. As you can see, the visual fine-tuning added the most of the code:

```
levs <- rev(c("A", "C", "G", "T", "N"))
dm %>%
  mutate(baseo=factor(base, levels=rev(levs))) %>%
  ggplot(aes(bin, fill=baseo, order=factor(baseo, levs))) + geom_bar(position="fill")
```

If you still want to get the line chart, you need to calculate the relative frequencies yourself:

```
dm %>%
  select(base, bin) %>%
  table %>%
  data.frame %>%
  group_by(bin) %>%
  mutate(Freqn=Freq / sum(Freq)) ->
  t

t %>%
  mutate(baseo=factor(base, levels=levs)) %>%
  ggplot(aes(bin, Freqn, colour=baseo, group=baseo)) + geom_line(size=1.3)
```

Now you can think for a while about what is better about the bar chart, and what is better about the line chart.

Now check the quality of reads in (create the commands on your own, modyfying the provided ones):

```
/data/ecoli
```

CHAPTER 8

Genome assembly

We'll be assembling a genome of *E. coli* from paired Illumina reads. We're using `Velvet`, which is a bit obsolete, but nice for teaching purposes. Runs quite fast and does not consume that much memory. In reality we'd use something like `Megahit` for prokaryotic organisms or `Spades` for the rest today.

You've already got a project directory for the assembly: `~/projects/assembly`. You can find linked `00-reads` there with the fastq files, and two assembly results should the process would fail for anyone.

```
cd ~/projects/assembly

# look what's around
ll
```

Velvet is used in two phases, the first phase prepares the reads, the second phase does the assembly itself. Open the Velvet manual. When using anything more complex than a simple notepad you will actually save your time by reading the manual. Surprised?;)

Also - run `screen` at this point, because you want to continue working, while Velvet will be blocking one of the consoles.

```
# load and prepare the reads
velveth 03-velvet-k21 21 -fastq -short -separate 00-reads/MiSeq_Ecoli_MG1655_50x_R1.fastq 00-reads/Mi

# do the assembly - you can flip through the manual in the meantime..
velvetg 03-velvet-k21
```

Running the whole assembly process is actually that simple. What is not simple is deciding, whether your assembly is correct and whether it is the best one you can get with your data. There is actually a lot of trial and error involved if you're decided to get the best one. People usually combine several assemblers, test several settings for each assembler and then combine the best runs from each of the assemblers with another assembler..;)

The best criteria for evaluating your assembly are usually external - N50 is nice, but does not tell you much about chimeric contigs for example. So overlaps with another assembly of some close species, the number of genes that can be found using protein sequences from a close species are good metrics.

```
# check the expected (assembly) coverage
~/sw/velvet_1.2.10/contrib/estimate-exp_cov/velvet-estimate-exp_cov.pl 03-velvet-k21/stats.txt | less
```

On the other hand when it's bad, any metrics will do - the reported N50 of 94 basepairs means there is something [terribly] wrong. Let's try to use the information on read pairs.

```
velveth 04-velvet-k31-paired 31 -fastq -shortPaired -separate 00-reads/MiSeq_Ecoli_MG1655_50x_R1.fast

velvetg 04-velvet-k31-paired -exp_cov auto -ins_length 150 -read_trkg yes
```

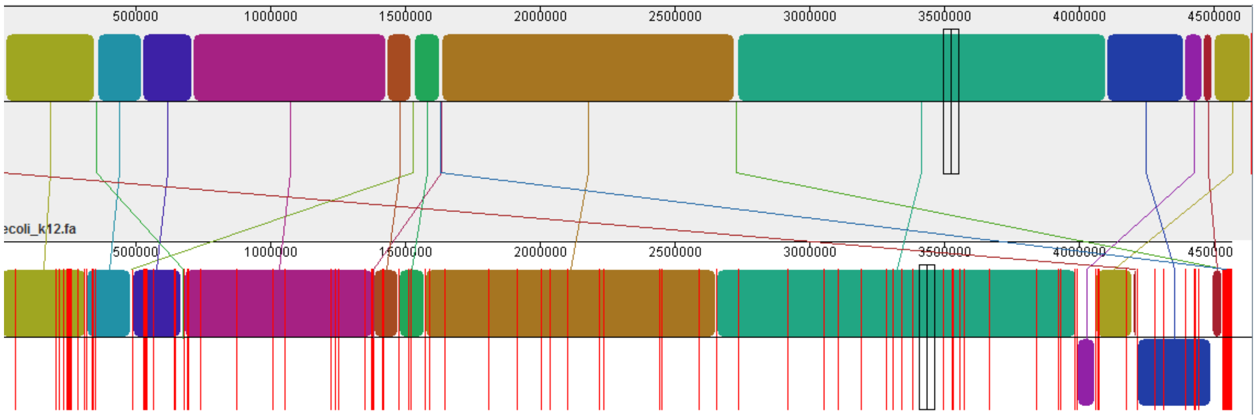
```
# check the expected (assembly) coverage
~/sw/velvet_1.2.10/contrib/estimate-exp_cov/velvet-estimate-exp_cov.pl 04-velvet-k31-paired/stats.txt

# check observerd insert length
~/sw/velvet_1.2.10/contrib/observed-insert-length.pl/observed-insert-length.pl 04-velvet-k31-paired
```

The `observed-insert-length.pl` calculates suggestions for `-ins_length` and `-ins_length_sd` parameters to `velvetg`, so let's try if the suggestions improve the assembly:

```
velvetg 04-velvet-k31-paired -exp_cov auto -ins_length 297 -ins_length_sd 19.4291558645302 -read_trkg
```

Now the run is really fast, because most of the work is already done. The N50 improved significantly, and we also got ~50 contigs less, which could mean a better assembly. When there is already some reference sequence available, we can compare the size of the reference sequence with our assembly - the more similar, the better;) 4.6 Mbp is quite close to 4,561,418 ... nice.

Using `Mauve` we can align the result with the reference *E. coli* genome:



Here you can get the genome yourself. If there is time, we can try to map our reads to it:

```
wget -O ecoli.fa.gz https://owncloud.cesnet.cz/public.php?service=files&t=0c7d291a4ad1ff39911ef1fa879
```

CHAPTER 9

Genomic tools session

9.1 Genome feature arithmetics & summary

Explore bedtools & bedops functionality

- http://bedtools.readthedocs.org/en/
- http://bedops.readthedocs.org/en/

Prepare files - we work with mouse genome data:

```
cd
mkdir projects/bed_examples
cp /data/bed_examples/* projects/bed_examples/.
cd projects/bed_examples
```

1. Merge the overlapping open chromatin regions in `encode.bed` file

In this first exercise we will work with open chromatin regions based on DNaseI hypersensitive sites in file `encode.bed` obtained from ENCODE database. As this database contains open chromatin regions from multiple experiments, the open chromatin regions may overlap. In our analysis we want to merge these regions so that the same/similar regions is present only once. You can use `bedtools merge` tool:

```
# Explore the encode.bed file
less encode.bed

# Count the number of regions before merging
wc -l encode.bed

# The data has to be sorted: use subshell to sort data before merging
bedtools merge -i <( sortBed -i encode.bed ) > encode-merged.bed

# Count the number of regions after merging
wc -l encode-merged.bed
```

2. Count the number of open chromatin regions in merged file overlapping with genes

In the second exercise we would like to parse and count those open chromatin regions which overlap with known genes retrieved from Ensembl database or are within 1000 bp on each side of a gene.

```
# Explore the Ensembl.NCBIM37.67.bed file
less Ensembl.NCBIM37.67.bed

# Count the number of open chromatin regions overlapping with genes
# or are within 1000 bp window on each side of a gene
```

```
bedtools window -w 1000 \
-a <( sortBed -i encode-merged.bed ) \
-b <( sortBed -i Ensembl.NCBIM37.67.bed ) |
wc -l

# Count the number of open chromatin regions overlapping with genes
bedtools intersect \
-a <( sortBed -i encode-merged.bed ) \
-b <( sortBed -i Ensembl.NCBIM37.67.bed ) |
wc -l
```

3. Count the number of merged open chromatin regions file overlapping with genes

Here, we are supposed to do right the opposite, i.e. count the number of genes containing open chromatin region from the ENCODE dataset.

```
bedtools intersect \
-a <( sortBed -i encode-merged.bed ) \
-b <( sortBed -i Ensembl.NCBIM37.67.bed ) -wb |
cut -f 7 |
sort -u |
wc -l
```

4. Make three sets of sliding windows across mouse genome (1 Mb, 2.5 Mb, 5 Mb) with the step size 0.2 by the size of the window and obtain gene density within these sliding windows. To speed up the process we focus only on chromosome X.

```
# Explore fasta index file
less genome.fa.fai

# Make 1Mb sliding windows (step 200kb)
bedtools makewindows \
-g <( grep '^X' genome.fa.fai ) \
-w 1000000 \
-s 200000 \
-i winnum \
> windows_1mb.bed

# Make 2.5Mb sliding windows (step 500kb)
bedtools makewindows \
-g <( grep '^X' genome.fa.fai ) \
-w 2500000 \
-s 500000 \
-i winnum \
> windows_2-5mb.bed

# Make 5Mb sliding windows (step 1Mb)
bedtools makewindows \
-g <( grep '^X' genome.fa.fai ) \
-w 5000000 \
-s 1000000 \
-i winnum \
> windows_5mb.bed

# Obtain densities of genes within individual windows
bedtools coverage \
-a <( sortBed -i Ensembl.NCBIM37.67.bed ) \
-b windows_1mb.bed \
> gdens_windows_1mb.tab
```

```
bedtools coverage \
-a <( sortBed -i Ensembl.NCBIM37.67.bed ) \
-b windows_2-5mb.bed \
> gdens_windows_2-5mb.tab

bedtools coverage \
-a <( sortBed -i Ensembl.NCBIM37.67.bed ) \
-b windows_5mb.bed \
> gdens_windows_5mb.tab
```

The gene density can be visualized in R-Studio.

9.2 VCFtools

Explore vcftools functionality

- http://vcftools.sourceforge.net

Prepare data files into `~projects/diff` directory:

```
cd
mkdir projects/diff

cp /data/mus_mda/00-popdata/* projects/diff/.

cd projects/diff

# View and explore the files within the 'vcf' directory
ls
```

Obtaining the basic file statistics (number of variants & number of samples):

```
vcftools --gzvcf popdata_mda.vcf.gz
```

Viewing and printing out the content of the VCF file:

```
# To print out the content of the VCF file

vcftools --gzvcf popdata_mda.vcf.gz --recode --out new_vcf

# To view the content directly

vcftools --gzvcf popdata_mda.vcf.gz --recode --stdout | less -S
```

Basic data filtering - use of appropriate flags:

```
--keep ind.txt # Keep these individuals
--remove ind.txt # Remove these individuals
--snps snps.txt # Keep these SNPs
--snps snps.txt --exclude # Remove these SNPs
```

To select a subset of samples:

```
vcftools --gzvcf popdata_mda.vcf.gz \
--keep euro_samps.txt \
--recode \
--stdout |
less -S
```

Select subset of samples and SNPs based on physical position in genome:

```
# Flags you can use:
--chr 11 # Keep just this chromosome
--not-chr 11 # Remove this chromosome
--not-chr 11 -not-chr 2 # Remove these two chromosomes
--from-bp 20000000 # Keep SNPs from this position
--to-bp 22000000 # Keep SNPs to this position
--bed keep.bed # Keep only SNPs overlapping with locations listed in a file
--exclude-bed remove.bed # The opposite of the previous
```

```
vcftools --gzvcf popdata_mda.vcf.gz \
--chr 11 \
--from-bp 22000000 \
--to-bp 23000000 \
--keep euro_samps.txt \
--recode \
--stdout |
less -S
```

Select subset of samples and then select SNPs with no missing data and with minor allele frequency (MAF) no less than 0.2:

```
# Flags you can use:
--maf 0.2 # Keep just variants with Minor Allele Freq higher than 0.2
--hwe 0.05 # Keep just variants which do not deviate from HW equilibrium (p-value = 0.05)
--max-missing (0-1) # Remove SNPs with given proportion of missing data (0 = allowed completely miss
--minQ 20 # Minimal quality allowed (Phred score)
```

```
vcftools --gzvcf popdata_mda.vcf.gz \
--keep euro_samps.txt \
--recode \
--stdout |
vcftools \
--vcf - \
--max-missing 1 \
--maf 0.2 \
--recode \
--stdout |
less -S

vcftools --gzvcf popdata_mda.vcf.gz \
--keep euro_samps.txt \
--recode \
--stdout |
vcftools --vcf - \
--max-missing 1 \
--maf 0.2 \
--recode \
--stdout \
> popdata_mda_euro.vcf
```

Use the newly created `popdata_mda_euro.vcf` representing variants only for a subset of individuals and variants to calculate Fst index. In order for vcftools to calculate Fst index the populations have to be specified in the output - each one with a separate file (`--weir-fst-pop pop1.txt` and `--weir-fst-pop pop2.txt`).

```
# Flags you can use:
--site-pi # Calculates per-site nucleotide diversity (π)
--window-pi 1000000 --window-pi-step 250000 # Calculates per-site nucleotide diversity for windows o
```

```
--weir-fst-pop pop1.txt --weir-fst-pop pop2.txt # Calculates Weir & Cockerham's Fst
--fst-window-size 1000000 --fst-window-step 250000 # Calculates Fst for windows of 1Mb with 250Kb ste
```

```
vcftools --vcf popdata_mda_euro.vcf \
--weir-fst-pop musculus_samps.txt \
--weir-fst-pop domesticus_samps.txt \
--stdout |
less -S
```

9.3 Exercise

Get a population differentiation calculated as Fst between *M. m. musculus* and *M. m. domesticus* within a given sliding window and find candidate genes within highly differentiated regions:

1. use `vcftools` to filter data and calculate Fst for individual SNPs
2. use `bedtools makewindows` to create sliding windows of three sizes:
   (a) 100 kb + 10 kb step
   (b) 500 kb + 50 kb step
   (c) 1 Mb + 100 kb step
3. calculate average Fst for each window
4. use R-Studio and ggplot2 to plot Fst values across the genome
5. use R or `tabtk` to obtain the 99th percentile and use it to obtain a set of candidate genomic regions
6. use `bedtools intersect` to get a list of candidate genes

Extract genotype data for European mouse individuals and filter out variants having more than one missing genotype and minor allele frequency 0.2 (we have already started - you should have prepared VCF file with European samples and filtered out variants with missing genomes and low minor allele frequency).

```
cd ~/projects/diff

vcftools --gzvcf popdata_mda.vcf.gz \
--keep euro_samps.txt \
--recode --stdout |
vcftools --vcf - \
--max-missing 1 \
--maf 0.2 \
--recode \
--stdout \
> popdata_mda_euro.vcf
```

Calculate Fst values for variants between *M. m. musculus* and *M. m. domesticus* populations (populations specified in `musculus_samps.txt` and `domesticus_samps.txt`):

```
vcftools --vcf popdata_mda_euro.vcf \
--weir-fst-pop musculus_samps.txt   \
--weir-fst-pop domesticus_samps.txt \
--stdout |
tail -n +2 |
awk -F $'\t' 'BEGIN{OFS=FS}{print $1,$2-1,$2,$1":"$2,$3}' \
> popdata_mda_euro_fst.bed
```

Make the three sets of sliding windows (100 kb, 500 kb, 1 Mb) and concatenate them into a single file:

```
cp /data/mus_mda/02-windows/genome.fa.fai .

## Create windows of 1 Mb with 100 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) \
-w 1000000 \
-s 100000  \
-i winnum |
awk '{print $0":1000kb"}' \
> windows_1000kb.bed

## Create windows of 500 kb with 500 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) \
-w 500000 \
-s 50000  \
-i winnum |
awk '{print $0":500kb"}' \
> windows_500kb.bed

## Create windows of 100 kb with 10 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) \
-w 100000 \
-s 10000  \
-i winnum | \
awk '{print $0":100kb"}' \
> windows_100kb.bed

## Concatenate windows of all sizes
cat windows_*.bed > windows.bed
```

Calculate average Fst within the sliding windows:

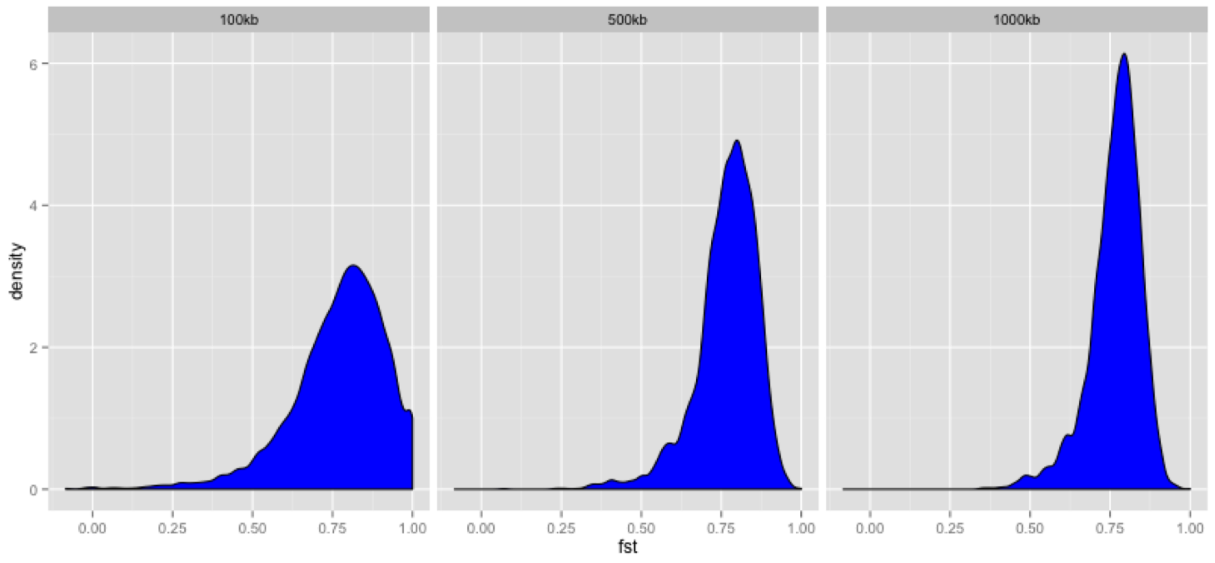
```
## Input files for bedtools groupby need to be sorted

# Join Fst values and the 'windows.bed' file
bedtools intersect \
  -a <( sortBed -i windows.bed ) \
  -b <( sortBed -i popdata_mda_euro_fst.bed ) -wa -wb \
> windows_fst.tab

# Run bedtools groupby command to obtain average values of Fst
bedtools groupby -i <( sort -k4,4 windows_fst.tab ) \
-g 1,2,3,4 \
-c 9 \
-o mean |
tr ":" "\t" > windows_mean_fst.tab
```

Visualize the average Fst values within the sliding windows of the three sizes between the two house mouse subspecies in R-Studio. Plot the distribution of the Fst values for the three window sizes and also plot the average Fst values along the chromosomes.

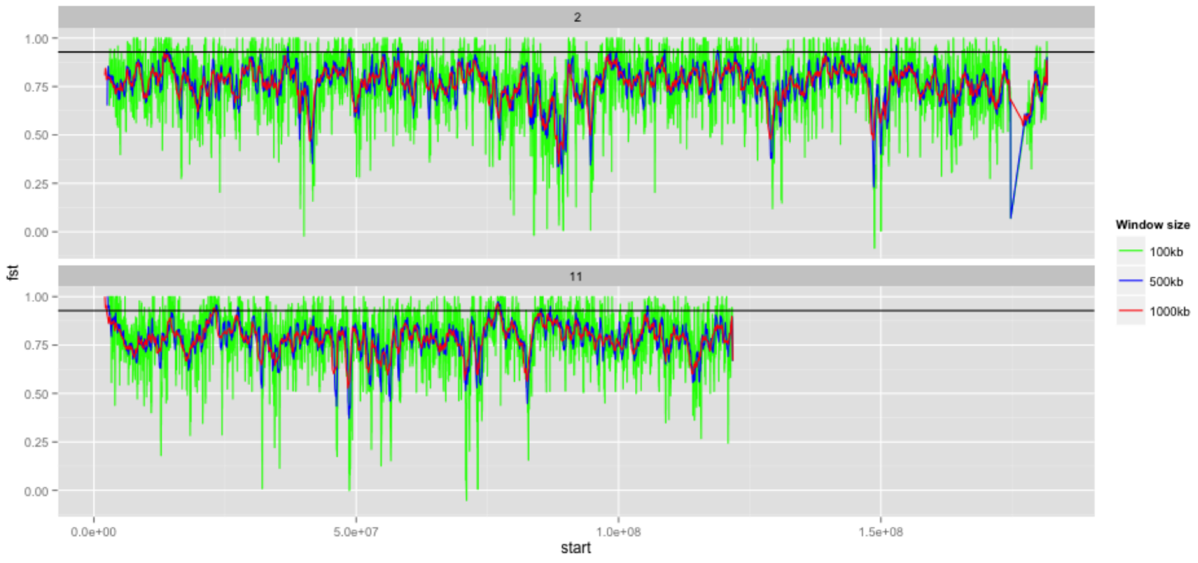
Note: R ggplot2 commands to plot population differentiation

```
library(ggplot2)

setwd("~/projects/diff")

fst <- read.table("windows_mean_fst.tab", header=F, sep="\t")
```

```
# shorthand for TAB separated files
fst <- read.delim("windows_mean_fst.tab", header=F)

names(fst) <- c("chrom", "start", "end", "win_id","win_size", "avg_fst" )

# the 'old' way
fst$win_size <- factor(fst$win_size, levels=c("100kb", "500kb", "1000kb"))

# dplyr version of the command above
library(dplyr)
fst %>%
  mutate(win_size = factor(win_size, levels=c("100kb", "500kb", "1000kb")) ->
  fst

ggplot(fst, aes(avg_fst)) +
        geom_density(fill=I("blue")) +
        facet_wrap(~win_size)
```



```
ggplot(fst, aes(y=avg_fst, x=start, colour=win_size)) +
        geom_line() +
        facet_wrap(~chrom, nrow=2) +
        scale_colour_manual(name="Window size", values=c("green", "blue","red"))

q <- quantile(subset(fst,win_size=="500kb",select="avg_fst")[,1],prob=0.99)[[1]]

ggplot(fst, aes(y=avg_fst, x=start, colour=win_size)) +
        geom_line() +
        facet_wrap(~chrom, nrow=2) +
        geom_hline(yintercept=q, colour="black") +
        scale_colour_manual(name="Window size", values=c("green", "blue","red"))
```



Find the 99th percentile of genome-wide distribution of Fst values in order to guess possible outlier genome regions. 99th percentile can be obtained running R as command line or by using `tabtk`. The output would be a list of windows having Fst higher than or equal to 99% of the data.

```
## Use of variables: var=value
## Use $() to pass the output of command/pipeline to a variable

# Calculate 99th percentile by R
q500=$( grep 500kb windows_mean_fst.tab |
  cut -f 6 |
  Rscript -e 'quantile(as.numeric(readLines("stdin")),probs=0.99)[[1]]' |
  cut -d " " -f 2 )

# Calculate 99th percentile by tabtk
q500=$( grep 500kb windows_mean_fst.tab |
  tabtk num -c 6 -Q |
  cut -f 13 )

## Inspect the variable
echo $q500

grep 500kb windows_mean_fst.tab |
  awk -v a=$q500 -F $'\t' 'BEGIN{OFS=FS}{if($6 >= a){print $1,$2,$3}}' |
  sortBed |
  bedtools merge -i stdin \
> signif_500kb.bed
```

Use the mouse gene annotation file to retrieve genes within the windows of high Fst (i.e. putative reproductive isolation loci).

```
</data/mus_mda/05-fst2genes/Mus_musculus.NCBIM37.67.gtf.gz zcat > Mus_musculus.NCBIM37.67.gtf

bedtools intersect \
    -a signif_500kb.bed \
    -b Mus_musculus.NCBIM37.67.gtf -wa -wb |
  grep protein_coding |
  cut -f 1,2,3,4,12 |
  cut -d ' ' -f 1,3,9 |
  tr -d '";' |
  sort -u \
> candidate_genes.tab
```

CHAPTER 10

Variant quality

In this part you will be working on your own. You're already familiar with the VCF format and some reformatting and plotting tools. There is a file with variants from several nightingale individuals:

```
/data/vcf_examples/luscinia_vars.vcf.gz
```

Your task now is:

- pick only data for chromosomes `chr1` and `chrZ`
- extract the sequencing depth `DP` from the `INFO` column
- extract variant type by checking if the `INFO` column contains `INDEL` string
- load these two columns together with the first six columns of the VCF into R
- explore graphically (barchart of variant types, histogram of qualities for INDELs and SNPs, ...)

And a bit of guidance here:

- create a new project directory in your `projects`
- get rid of the comments (they start with `#`, that is `^#` regular expression)
- filter lines based on chromosomes (`grep -e chr1 -e chrZ`)
- extact the first 6 columns (`cut -f1-6`)
- extract `DP` column (`egrep -o 'DP=[^;]*' | sed 's/DP=//'`)
- check each line for `INDEL` (`awk '{if($0 ~ /INDEL/) print "INDEL"; else print "SNP"}'`)
- merge the data (columns) before loading to R (`paste`)
- add column names while loading the data with `read.delim(..., col.names=c(...))`

  Good luck! (We will help you;)

Additional reference materials:

CHAPTER 11

Links

Here is few sources that you can go through, when you get bored during the course...

11.1 Bash

- http://www.tldp.org/LDP/abs/html/
- http://www.catonmat.net/blog/bash-one-liners-explained-part-three/
- http://wiki.bash-hackers.org/howto/redirection_tutorial
- http://wiki.bash-hackers.org/scripting/bashchanges

11.2 R Studio

- http://www.rstudio.com/resources/cheatsheets/

11.3 Visual design

- one secret link you got by email
- http://www.smashingmagazine.com/2014/03/28/design-principles-visual-perception-and-the-principles-of-gestalt/
- http://www.vanseodesign.com/web-design/gestalt-principles-of-perception/
- http://colorbrewer2.org

11.4 Genomic tools

- http://bedtools.readthedocs.org/en/
- http://bedops.readthedocs.org/en/
- http://vcftools.sourceforge.net

11.5 Genomic data formats

- http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-40
- http://www.ensembl.org/info/website/upload/gff.html
- https://genome.ucsc.edu/FAQ/FAQformat.html#format1

CHAPTER 12

Best practice

This is a collection of tips, that may help to overcome the initial barrier of working with a 'foreign' system. There is a lot of ways to achieve the solution, those presented here are not the only correct ones, but some that proved beneficial to the authors.

12.1 Easiest ways to get UNIX

To get the most basic UNIX tools, you can download an install Git for Windows. It comes with a nice terminal emulator, and installs to your right-click menu as 'Git Bash here' - which runs terminal in the folder that you clicked. Git itself is meant for managing versions of directories, but it cannot live without the UNIX environment, so someone did the hard work and packaged it all nicely together.

If you need more complete UNIX environment with many packages 'inside' your Windows, you can use Cygwin. But it is quite complete, which leads you to believe that it can replace native UNIX, which you'll find painfully later is not true;)

An easy way of getting UNIX environment in Windows is to install a basic Linux into a virtual machine as you have seen in the course. It's much more convenient that the dual boot configurations, and the risk of completely breaking your computer is lower. You can be using UNIX while having all your familiar stuff at hand. The only downside is that you have to transfer all the data as if the image was a remote machine. Unless you're able to set up windows file sharing on the Linux machine. This is the way the author prefers (you can ask;).

It's much more convenient to use a normal terminal like PuTTY to connect to the machine rather than typing the commands into the virtual screen of VirtualBox - It's usually lacking clipboard support, you cannot change the size of the window, etc.

Mac OS X and Linux are UNIX based, you just have to know how to start your terminal program (`konsole`, `xterm`, `Terminal` ...).

12.2 Essentials

Always use `screen` for any serious work. Failing to use screen will cause your jobs being interrupted when the network link fails (given you're working remotely), and it will make you keep your home computer running even if your calculation is running on a remote server.

Track system resources usage with `htop`. System that is running low on memory won't perform fast. System with many cores where only one core ('CPU') is used should be used for more tasks - or can finish your task much faster, if used correctly.

12.3 Data organization

Make a new directory for each project. Put all your data into subdirectories. Use symbolic links to reference huge data that are reused by more projects in your current project directory. Prefix your directory names with 2 digit numbers, if your projects have more than few subdirectories. Increase the number as the data inside is more and more 'processed'. Keep the code in the top directory. It is easy to distinguish data references just by having `[0-9]{2}-` prefix.

Example of genomic pipeline data directory follows:

```
00-raw --> /data/slavici/all-reads
01-fastqc
02-mm-cleaning
03-sff
10-mid-split
11-fastqc
12-cutadapt
13-fastqc
22-newbler
30-tg-gmap
31-tg-sim4db
32-liftover
33-scaffold
40-map-smalt
50-variants
51-variants
60-gff-primers
```

Take care to note all the code used to produce all the intermediate data files. This has two benefits: 1) your results will be really **reproducible** 2) it will **save you much work** when doing the same again, or trying different settings

If you feel geeky, use `git` to track your code files. It will save you from having 20 versions of one script - and you being completely lost a year later, when trying to figure out which one was the one that was actually working.

12.4 Building command lines

Build the pipelines command by command, keeping `| less -S` (or `| head` if you don't expect lines of the output to be longer than your terminal width) at the end. Every time you check if the output is what you expect, and only after that add the next command. If there is a `sort` in your pipeline, you have to put `head` in front of the `sort`, because otherwise sort has to process all the data before it gives out any output.

I (Libor) do prefer the 'input first' syntax (`<file command | comm2 | comm3 >out`) which improves legibility, resembles the real world pipeline (garden hose, input tap -> garden hose -> garden sprinkler) more, and when changing the input file names when reusing the pipeline, the names are easier to find.

Wrap your long pipelines on `|` - copy and paste to bash still works, because bash knows there has to be something after `|` at the end of the line. Only the last line has to be escaped with `\`, otherwise all your output would go to the screen instead of a file.

```
<infile sort -k3,3 |
  uniq -c -s64 |
  sort -k1rn,1 \
>out
```

You can get a nice progress bar if you use `pv` (pipe viewer) instead of `cat` at the beginning of the pipeline. But again, if there is a `sort` in your pipeline, it has to consume all the data before it starts to work.

Use variables instead of hard-coded file names / arguments, especially when the name is used more times in the process, or the argument is supposed to be tuned:

```
FILE=/data/00-reads/GS60IET02.RL1.fastq
THRESHOLD=300

# count sequences in file
<$FILE awk '(NR % 4 == 2)' | wc -l
# 42308

# count sequences longer that
<$FILE awk '(NR % 4 == 2 && length($0) > $THRESHOLD)' | wc -l
# 14190
```

12.5 Parallelization

Many tasks, especially in Big Data and NGS, are 'data parallel' - that means you can split the data in pieces, compute the results on each piece separately and then combine the results to get the complete result. This makes very easy to exploit the full power of modern multi core machines, speeding up your processing e.g. 10 times. `GNU parallel` is a nice tool that helps to parallelize bash pipelines, check the manual.

CHAPTER 13

Reference manual for UNIX introduction

13.1 Basic orientation in UNIX

Multiple windows (screen)

You're all used to work with multiple windows (in MS Windows;). You can have them in UNIX as well. The main benefit, however, is that you can log off and your programs keep running.

To go into a screen mode type:

```
screen
```

Once in screen you can control screen itself after you press the master key (and then a command): `ctrl+a key`. To create a new window within the screen mode, press `ctrl+a c` (create). To flip among your windows press `ctrl+a space` (you flip windows often, it's the biggest key available). To detach screen (i.e. keep your programs running and go home), press `ctrl+a d` (detach).

To open a detached screen type:

```
screen -r  # -r means restore
```

To list running screens, type:

```
screen -ls
```

Controlling processes (htop/top)

`htop` or `top` serve to see actual resource utilization for each running process. Htop is much nicer variant of standard `top`. You can sort the processes by memory usage, CPU usage and few other things.

Getting help (man)

Just any time you're not sure about program option while building a command line, just flip to next screen window (you're always using screen for serious work), and type `man` and name of the command you want to know more about:

```
man screen
```

13.1.1 Moving around & manipulation with files and directories

Basic commands to move around and manipulate files/directories.

```
pwd    # prints current directory path
cd     # changes current directory path
ls     # lists current directory contents
ll     # lists detailed contents of current directory
```

```
mkdir  # creates a directory
rm     # removes a file
rm -r  # removes a directory
cp     # copies a file/directory
mv     # moves a file/directory
locate # tries to find a file by name
```

Usage:

cd

To change into a specific subdirectory, and make it our current working directory:

```
cd go/into/specific/subdirectory
```

To change to parent directory:

```
cd ..
```

To change to home directory:

```
cd
```

To go up one level to the parent directory then down into the directory2:

```
cd ../directory2
```

To go up two levels:

```
cd ../../
```

ls

To list also the hidden files and directories (`-a`) in current in given folder along with human readable (`-h`) size of files (`-s`), type:

```
ls -ash
```

mv

To move a file data.fastq from current working directory to directory `/home/directory/fastq_files`, type:

```
mv data.fastq /home/directory/fastq_files/data.fastq
```

cp

To copy a file data.fastq from current working directory to directory `/home/directory/fastq_files`, type:

```
cp data.fastq /home/directory/fastq_files/data.fastq
```

locate

This quickly finds a file by a part of its name or path. To locate a file named data.fastq type:

```
locate data.fastq
```

The `locate` command uses a database of paths which is automatically updated only once a day. When you look for some recent files you may not find them. You can manually request the update:

```
sudo updatedb
```

Symbolic links

Symbolic links refer to some other files or directories in a different location. It is useful when one wants to work with some files accessible to more users but wants to have them in a convenient location at the same time. Also, it is useful when one works with the same big data in multiple projects. Instead of copying them into each project directory one can simply use symbolic links.

A symbolic link can are created by:

```
ln -s /data/genomes/luscinia/genome.fa genome/genome.fasta
```

13.2 Exploring and basic manipulation with data

less

Program to view the contents of text files. As it loads only the part of a the file that fits the screen (i.e. does not have to read entire file before starting), it has fast load times even for large files.

To view text file while disabling line wrap and add line numbers add options `-S` and `-N`, respectively:

```
less -SN data.fasta
```

To navigate within the text file while viewing use:

Key	Command
Space bar	Next page
b	Previous page
Enter key	Next line
/<string>	Look for string
<n>G	Go to line <n>
G	Go to end of file
h	Help
q	Quit

cat

Utility which outputs the contents of a specific file and can be used to concatenate and list files. Sometimes used in Czech as translated to 'kočka' and then made into a verb - 'vykočkovat';)

```
cat seq1_a.fasta seq1_b.fasta > seq1.fasta
```

head

By default, this utility prints first 10 lines. The number of first n lines can be specified by `-n` option (or by `-..number..`).

To print first 50 lines type:

```
.. code-block:: bash
```

head -n 50 data.txt

# is the same as head -50 data.txt

# special syntax prints all but last 50 lines head -n -50 data.txt

tail

By default, this utility prints last 10 lines. The number of last n lines can be specified by `-n` option as in case of head.

To print last 20 lines type:

```
tail -n 20 data.txt
```

To skip the first line in the file (e.g. to remove header line of the file):

```
tail -n +2 data.txt
```

grep

This utility searches a text file(s) for lines matching a text pattern and prints the matching lines. To match given pattern it uses either specific string or regular expressions. Regular expressions enable for a more generic pattern rather than a fixed string (e. g. search for `a` followed by 4 numbers followed by any capital letter - `a[0-9]{4}[A-Z]`).

To obtain one file with list of sequence IDs in multiple fasta files type:

```
grep '>' *.fasta > seq_ids.txt
```

To print all but #-starting lines from the vcf file use option `-v` (print non-matching lines):

```
    grep -v ^# snps.vcf > snps.tab
```

The `^#` mark means beginning of line followed directly by `#`.

wc

This utility generates set of statistics on either standard input or list of text files. It provides these statistics:

- line count (`-l`)
- word count (`-w`)
- character count (`-m`)
- byte count (`-c`)
- length of the longest line (`-L`)

If specific word provided it returns count of this word in a given file.

To obtain number of files in a given directory type:

```
    ls | wc -l
```

The `|` symbol is explained in further section.

cut

Cut out specific columns (fields/bytes) out of a file. By default, fields are separated by TAB. Otherwise, change delimiter using `-d` option. To select specific fields out of a file use `-f` option (position of selected fields/columns separated by commas). If needed to complement selected fields (i.e. keep all but selected fields) use `--complement` option.

Out of large matrix select all but first column and row representing IDs of rows and columns, respectively:

```
    < matrix1.txt tail -n +2 | cut --complement -f 1 > matrix2.txt
```

sort

This utility sorts a file based on whole lines or selected columns. To sort numerically use `-n` option. Range of columns used as sorting criterion is specified by `-k` option.

Extract list of SNPs with their IDs and coordinates in genome from vcf file and sort them based on chromosome and physical position:

```
    < snps.vcf grep ^# | cut -f 1-4 | sort -n -k2,2 -k3,3 > snps.tab
```

uniq

This utility takes sorted lists and provides unique records and also counts of non-unique records (`-c`). To have more numerous records on top of output use `-r` option for `sort` command.

Find out count of SNPs on each chromosome:

```
< snps.vcf grep ^# | cut -f 2 | sort | uniq -c > chromosomes.tab
```

tr

Replaces or removes specific sets of characters within files.

To replace characters a and b in the entire file for characters c and d, respectively, type:

```
tr 'ab' 'cd' < file1.txt > file2.txt
```

Multiple consecutive occurrences of specific character can be replaced by single character using `-s` option. To remove empty lines type:

```
tr -s '\n' < file1.txt > file2.txt
```

To replace lower case to upper case in fasta sequence type:

```
tr "[:lower:]" "[:upper:]" < file1.txt > file2.txt
```

13.3 Building commands

Globbing

Refers to manipulating (searching/listing/etc.) files based on pattern matching using specific characters.

Example:

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls *.fasta
# seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
```

Character `*` in previous example replaces any number of any characters and it indicates to `ls` command to list any file ending with ".fasta".

However, if we look for fastq instead, we get no result:

```
ls *.fastq
#
```

Character `?` in following example replaces just right the one character (a/b) and it indicates to ls functions to list files containing *seq2_* at the beginning, any single character in the middle (a/b) and ending with ".fasta"

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq2_?.fasta
# seq2_a.fasta seq2_b.fasta
```

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq2_[ab].fasta
# seq2_a.fasta seq2_b.fasta
```

One can specifically list altering characters (a,b) using brackets `[]`. One may also be more general and list all files having any alphabetical character `[a-z]` or any numerical character `[0-9]`:

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq[0-9]_[a-z].fasta
# seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
```

TAB completition

Using key TAB one can finish unique file names or paths without having to fully type them. (try and see)

From this perspective it is important to think about names for directories in advance as it can spare you a lot time in future. For instance, when processing data with multiple steps one can use numbers at beginnings of names:

- 00-beginning
- 01-first-processing
- 02-second-processsing
- ...

Variables

UNIX environment enables to use shell variables. To set primer sequence `'GATACGCTACGTGC'` to variable `PRIMER1` in a command line and print it on screen using `echo`, type:

```
PRIMER1=GATACGCTACGTGC
echo $PRIMER1
# GATACGCTACGTGC
```

Note: It is good habit in UNIX to use capitalized names for variables: `PRIMER1` not `primer1`.

Pipes

UNIX environment enables to chain commands using pipe symbol `|`. Standard output of the first command serves as standard input of the second one, and so on.

```
ls | head -n 5
```

Subshell

Subshell enables to run two commands and capture the output into single file. It can be helpful in dealing with data files headers. Use of subshell enables to remove header, run the set of operations on the data, and later insert the header back to file. The basic syntax is:

```
(command1 file1.txt && command2 file1.txt) > file2.txt
```

To sort data file based on two columns without including header type:

```
(head -n 1 file1.txt && tail -n +2 file1.txt | sort -n -k1,1 -k2,2) > file2.txt
```

Subshell can be used also to preprocess multiple inputs on the fly (saving useless intermediate files):

```
paste <(< file1.txt tr ' ' '\t') <(<file2.txt tr '' '\t') > file3.txt
```

13.4 Advanced text manipulation (sed)

`sed` "stream editor" allows you to change file line by line. You can substitute text, you can drop lines, you can transform text... but the syntax can be quite opaque if you're doing anything more than substituting *foo* with *bar* in every line (`sed 's/foo/bar/g'`).

13.5 More complex data manipulation (awk)

`awk` enables to manipulate text data in a very complex way. In fact, it is a simple programming language with functionality similar to regular programming languages. As such it enables enormous variability in ways of how to process text data.

It can be used to write a short script and which can be chained along with UNIX commands in one pipeline. The biggest power of *awk* is that it's line oriented and saves you lot of boilerplate code that you would have to write in other languages, if you need moderately complex processing of text files. The basic structure of the script is divided into three parts and any of these three parts may or may not be included in the script (according to the intention of user). The first part `'BEGIN{}'` conducts operation before going through the input file, the middle part `'{}'` goes throughout the input file and conducts operations on each line separately. The last part `'END{}'` conducts operation after going through the input file.

The basic syntax:

```
< data.txt awk 'BEGIN{<before data processing>} {<process each line>} END{<after all lines are p
```

Built-in variables

awk has several built-in variables which can be used to track and process data without having to program specific feature.

The basic four built-in variables:

- `FS` - input field separator
- `OFS` - output field separator
- `NR` - record (line) number
- `NF` - number of fields in record (in line)

There is even more built-in variables that we won't discuss here: `RS`, `ORS`, `FILENAME`, `FNR`

Use of built-in variables:

awk splits each line into columns based on white space. When a different delimiter (e.g. TAB) is to be used, it can be specified using `-F` option. If you want to keep this custom Field Separator in the output, you have to set the Output Field Separator as well (there's no command line option for OFS):

```
< data.txt awk -F $'\t' 'BEGIN{OFS=FS}{print $1,$2}' > output.txt
```

This command takes file data.txt, extract first two TAB delimited columns of the input file and print them TAB delimited into the output file output.txt. When we look more closely on the syntax we see that the TAB delimiter was set using `-F` option. This option corresponds to the `FS` built-in variable. As we want TAB delimited columns in the output file we pass `FS` to `OFS` (i.e. ouput field separator) in the `BEGIN` section. Further, in the middle section we print out first two columns which can be extracted by numbers with `$` symbol (`$1`, `$2`). The numbers correspond to position of the column in the input file. We could, of course, use for this operation the `tr` command which is even simpler. However, the awk enables to conduct any other operation on given data.

Note: The complete input line is stored in `$0`.

The `NR` built-in variable can be used to capture each second line in a file type:

```
< data.txt awk '{ if(NR % 2 == 0){ print $0 }}' > output.txt
```

The `%` symbol represents modulo operator which returns the remainder of division. The `if()` condition is used to decide on whether the modulo is 0 or not.

Here is a bit more complex example of how to use `awk`. We write a command which retrieves coordinates of introns from coordinates of exons.

Example of input file:

```
GeneID            Chromosome   Exon_Start   Exon_End
ENSG00000139618   chr13        32315474     32315667
ENSG00000139618   chr13        32316422     32316527
ENSG00000139618   chr13        32319077     32319325
ENSG00000139618   chr13        32325076     32325184
...               ...          ...          ...
```

The command is going to be as follows:

When we look at the command step by step we first remove header and sort data based on GeneID and Exon_Start columns:

```
< exons.txt tail -n +2 | sort -k1,1 -k3,3n | ...
```

Further, we write a short script using awk to obtain coordinates of introns:

```
... | awk -F $'\t' 'BEGIN{OFS=FS}{
         if(NR==1){
           x=$1; end1=$4+1;
         }else{
           if(x==$1) {
               print $1,$2,end1,$3-1; end1=$4+1;
           }else{
               x=$1; end1=$4+1;
           }
         }
       }' > introns.txt
```

In the `BEGIN{}` part we set TAB as output field separator. Further, using `NR==1` test we set GeneID for first line into `x` variable and intron start into end1 variable. Otherwise we do nothing. For others records `NR > 1` condition `x==$1` test if we are still within the same gene. If so we print exon end from previous line (`end1`) as intron start and exon start of current line we use as intron end. Next, we set new intron start (i.e. exon end from current line) into end1. If we have already moved into new one `x<>$1`) we repeat procedure for the first line and print nothing waiting for next line.

CHAPTER 14

Important NGS formats

A selection of the most commonly used formats in NSG data processing pipelines.

14.1 FASTQ - Short reads

Sequencing instruments produce not only base calls, but usually can assign some quality score to each called base. Fastq contains multiple sequences, and each sequence is paired with quality scores for each base. The quality scores are encoded in text form.

- http://maq.sourceforge.net/fastq.shtml

14.2 SAM - Reads mapped to reference

SAM stands for Sequence Alignment/Mapping format. It includes parts of the original reads, that were mapped to a reference genome, together with the position where they belong to. There is an effective binary encoded counterpart called **BAM**.

- http://samtools.github.io/hts-specs/SAMv1.pdf

14.3 BED and GFF - Annotations

Annotations are regions in given reference genome with some optional additional information. BED is very simple and thus easy to work with for small tasks, GFF (General Feature Format) is a comprehensive format allowing feature nesting, arbitrary data fields for each feature and so on.

- http://genome.ucsc.edu/FAQ/FAQformat.html#format1
- http://www.ensembl.org/info/website/upload/gff.html

14.4 VCF - Variants in individuals

VCF stands for Variant Call Format. Given a reference and a set of sequenced individuals, VCF is a format to store the differences in these individuals, compared to the reference, efficiently. There is also a binary counterpart **BCF**.

- http://samtools.github.io/hts-specs/VCFv4.2.pdf

Additional exercises

These are tasks that do not fit any particular session, but we still consider them interesting enough to share them with you.

15.1 Counting heads

This is a nice example where bash can be used to solve a combinatorial problem by enumerating all the possibilities. And it is a long pipeline, so you have a lot of code to examine;)

> Eight people are sitting around a circular table. Each has a coin. They all flip their coins. What is the probability that no two adjacent people will get heads?

The basic idea is that there is not much possibilities (only 2 to the power of 8, that is 256). We can just enumerate all the combinations and check if there is two adjacent heads.

This is the final solution, take your time to take it apart to see what each piece does.

```
(echo "obase=2;"; printf "%d\n" {0..255}) | bc |  # generate numbers 0-255 in binary
  sed 's/^/0000000/' | egrep -o '.{8}$' |         # pad the output to 8 characters
  sed 'h;G;s/\n//' |                              # double the pattern on each line to simulate ring
  grep -v 11 |                                    # ignore all patterns where two heads (1) are next
  wc -l                                           # count the rest
```

To get a more understandable code, we can split it to functional parts. Then we can just play and try different implementations of the parts:

```
generate () { (echo "obase=2;"; printf "%d\n" {0..255}) | bc ;}
pad () { sed 's/^/0000000/' | egrep -o '.{8}$' ;}
ring () { sed p | paste - - | tr -d "\t" ;}

generate | pad | ring | grep -v 11 | wc -l
```

These are alternative solutions - you can paste them one by one, and check if the pipe is still working.

```
generate () { (echo "obase=2;"; echo {0..255} | tr " " "\n") | bc ;}
generate () { (echo "obase=2;"; echo {0..255} | xargs -n1) | bc ;}
generate () { (echo "obase=2;"; printf "%d\n" {0..255}) | bc ;}

pad () { awk '{print substr(sprintf("0000000%s", $0), length);}' ;}
pad () { sed 's/^/0000000/' | rev | cut -c-8 | rev ;}
pad () { sed 's/^/0000000/' | egrep -o '.{8}$' ;}

ring () { sed p | paste - - | tr -d "\t" ;}
ring () { sed 'h;G;s/\n//' ;}

generate | pad | ring | grep -v 11 | wc -l
```

The question was asking for the probability, thats one more division:

```
echo "scale=3;$( generate | pad | ring | grep -v 11 | wc -l ) / 256" | bc
```

15.1.1 Solutions by participants

One way to get a shorter (but much slower) solution is to ignore the binary conversion altogether, just use a huge list of decimal numbers and filter out anything that does not look like binary. Few variants follow:

```
seq -w 0 11111111 | grep ^[01]*$ | awk '!/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | grep ^[01]*$ | grep -v -e 11 -e ^1.*1$ | wc -l
seq -w 0 11111111 | awk '/^[01]*$/ && !/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | awk '!/[2-9]/ && !/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | grep -v -e [2-9] -e 11 -e ^1.*1$ | wc -l
```

I believe there are still a few ways to make it shorter;)

15.2 Dimensionality reduction

Methods like PCA and MDS (sometimes called PCoA to increase the confusion) are usually regarded as black box by many. Here we try to present a simple example that should help with getting a better idea on what are these magic boxes actually doing.

15.2.1 Load and visualize your data set

Now you can go to R and load the data:

```
setwd('~/projects/banana')
d <- read.csv("webapp/data/rotated.csv")
```

Plot the data to look what we've got:

```
library(ggplot2)
ggplot(d, aes(x, y)) + geom_point() + coord_equal()
```

15.2.2 Correct the distortion

Maybe you can already recognize what's in your data. But it appears to be a bit .. rotated. Here is a code for 3d rotation of points, copy, paste and run it in your R session:

```
# create a 3d rotation matrix
# https://www.math.duke.edu/education/ccp/materials/linalg/rotation/rotm3.html
rotX <- function(t) matrix(c(cos(t), sin(t), 0, -sin(t), cos(t), 0, 0, 0, 1), nrow=3)
rotY <- function(t) matrix(c(1, 0, 0, 0, cos(t), sin(t), 0, -sin(t), cos(t)), nrow=3)
rotZ <- function(t) matrix(c(cos(t), 0, -sin(t), 0, 1, 0, sin(t), 0, cos(t)), nrow=3)
rot3d <- function(tx, ty, tz) rotX(tx) %*% rotY(ty) %*% rotZ(tz)

# rotate a data frame with points in rows
```

```
rot3d_df <- function(df, tx, ty, tz) {
  rmx <- rot3d(tx, ty, tz)
  res <- data.frame(t(apply(df, 1, function(x) rmx %*% as.numeric(x))))
  colnames(res) <- colnames(df)
  res
}
```

Now try to rotate the object a bit, so we can see it better. Try to find good values for the rotation yourself (numbers are in radians, 0..2*PI makes sense):

```
dr <- rot3d_df(d, .9, .1, 2)
ggplot(dr, aes(x, y)) + geom_point() + coord_equal()
```

Enter PCA. It actually finds the best rotation for you. Even in a way that the first axis has the most variability (longest side of the object), the second axis has the maximum of the remaining variability etc.

```
pc <- prcomp(as.matrix(dr))
ggplot(data.frame(pc$x), aes(PC1, PC2)) + geom_point() + coord_equal()
ggplot(data.frame(pc$x), aes(PC1, PC3)) + geom_point() + coord_equal()
ggplot(data.frame(pc$x), aes(PC2, PC3)) + geom_point() + coord_equal()
```

15.2.3 MDS

Metric MDS (multidimensional scaling) with *euclidean* distance equals to PCA. We will use the non-metric variant here, which tries to keep only the order of pairwise distances, not the distances themselves. You prefer MDS when you want to use a different distance than *euclidean* - we're using *manhattan* (*taxicab*) distance here:

```
library(MASS)
dmx <- dist(dr, "manhattan")
mds <- isoMDS(dmx)
ggplot(data.frame(mds$points), aes(X1, X2)) + geom_point() + coord_equal()
```

15.2.4 Shiny

And now there is something you definitely wanted, while you were trying to find the good values for rotation of your object:

```
setwd('webapp')
```

Now `File > Open`, and open `server.R`. There should be a green `Run App` button at the top right of the editor window. Click that button!

Supplemental information:

CHAPTER 16

Course materials preparation

This section contains the steps that we did to produce the materials that course participants got ready-made. That is the **linux machine image**, **online documentation** and the **slide deck**.

16.1 Online documentation

Login to https://github.com. Create a new project called *ngs-course*, with a default readme file.

Clone the project to local machine and initialize *sphinx* docs. Choose `SSH` clone link in GitHub.

```
git clone git@github.com:libor-m/ngs-course.git

cd ngs-course

# use default answers to all the questions
# enter project name and version 1.0
sphinx-quickstart
```

Now track all files created by *sphinx-quickstart* in current directory with *git* and publish to GitHub.

```
git add .
git commit -m 'empty sphinx project'

# ignore _build directory in git
echo _build >> .gitignore
git add .gitignore
git commit -m 'ignore _build directory'

# publish the first docs
# setting up argument less git pull with '-u'
git push -u origin master
```

To get live view of the documents, login to https://readthedocs.org. Your *GitHub* account can be paired with *Read the Docs* account in *Edit Profile/Social Accounts*, then you can simply 'import' new projects from your GitHub with one click. Import the new project and wait for it to build. After the build the docs can be found at http://ngs-course.readthedocs.org (or click the `View` button).

Now write the docs, commit and push. Rinse and repeat. Try to keep the commits small, just one change a time.

```
git add _whatever_new_files_
git commit -m '_your meaningful description of what you did here_'
git push
```

References that may come handy:

- Thomas Cokelaer's cheat sheet

Use http://goo.gl to shorten a link to www.seznam.cz, to get a tracking counter url for the network connectivity test.

16.2 VirtualBox image

16.2.1 Create new VirtualBox machine

- Linux/Debian (32 bit)
- 1 GB RAM - this can be changed at the users machine, if enough RAM is available
- 12 GB HDD as system drive (need space for basic system, gcc, rstudio and some data)
- setup port forwarding
  - 2222 to 22 (ssh, avoiding possible collisions on linux machines with sshd running)
  - 8787 to 8787 (rstudio server)
  - 5690 to 5690 (rstudio + shiny)

16.2.2 Install Debian

Download Debian net install image - use i386 so there is as few problems with virtualization as possible. Not all machines can virtualize x64.

https://www.debian.org/CD/netinst/

Connect the iso to IDE in the virtual machine. Start the machine. Choose `Install`.

Mostly the default settings will do.

- English language (it will cause less problems)
- Pacific time zone (it is connected with language, no easy free choice;)
- hostname `node`, domain `vbox`
- users: root:debian, user:user
- simple partitioning (all in one partition, no LVM)
- Czech mirror to get fast installer file downloads
- pick only SSH server and Standard system utilities

Log in as root:

```
apt-get install sudo
usermod -a -G sudo user
```

Login as user (can be done by `su user` in root shell):

```
# colrize prompt - uncomment force_color_prompt=yes
# add ll alias - uncomment alias ll='ls -l'
# fast sort and uniq
# export LC_ALL=C
# maximal width of man
# export MANWIDTH=120
# # wget impersonating normal browser
# # good for being tracked with goo.gl for example
# alias wgets='H="--header"; wget $H="Accept-Language: en-us,en;q=0.5" $H="Accept: text/html,applica
nano ~/.bashrc
. ~/.bashrc

# set timezone so the time is displayed correctly
echo "TZ='Europe/Prague'; export TZ" >> ~/.profile

# some screen settings
cat >~/.screenrc << 'EOF'
hardstatus alwayslastline
hardstatus string '%{= kG}[%{G}%H%? %1`%?%{g}][%= %{= kw}%-w%{+b yk} %n*%t%?(%u)%? %{-}%+w %=%{g}][%

defscrollback 20000

startup_message off
EOF

# everyone likes git and screen
sudo apt-get install git screen pv curl wget

# add important stuff to python - this also installs GCC
sudo apt-get install python-dev python-pip python-virtualenv

# java because of fastqc
sudo apt-get install openjdk-7-jre-headless
```

This is what it takes to create a basic usable system in VirtualBox. We can shut it down now with `sudo shutdown -h now` and take a snapshot of the machine. If any installation goes haywire from now on, it's easy to revert to this basic system.

16.2.3 Install additional software

R is best used in RStudio - server version can be used in web browser.

```
mkdir sw
cd sw

# install latest R
# http://cran.r-project.org/bin/linux/debian/README.html
sudo bash -c "echo 'deb http://mirrors.nic.cz/R/bin/linux/debian jessie-cran3/' >> /etc/apt/sources.l
sudo apt-key adv --keyserver keys.gnupg.net --recv-key 381BA480
sudo apt-get update
sudo apt-get install r-base

sudo R
> update.packages(.libPaths(), checkBuilt=TRUE, ask=F)
> install.packages(c("ggplot2", "dplyr", "reshape2", "GGally", "stringr", "vegan", "svd", "tsne", "ti

# RStudio with prerequisities
sudo apt-get install gdebi-core
wget https://download2.rstudio.org/rstudio-server-0.99.491-i386.deb
sudo gdebi rstudio-server-0.99.491-i386.deb
```

There are packages that are not in the standard repos, or the versions in the repos is very obsolete. It's worth it to install such packages by hand, when there is not much dependencies.

```
# pipe viewer
cd ~/sw
wget -O - http://www.ivarch.com/programs/sources/pv-1.5.7.tar.bz2 | tar xvj
cd pv-1.5.7/
./configure
make
sudo make install

# parallel
cd ~/sw
wget -O - http://ftp.gnu.org/gnu/parallel/parallel-latest.tar.bz2|tar xvj
cd parallel-*/
./configure
make
sudo make install

# tabtk
cd ~/sw
git clone https://github.com/lh3/tabtk.git
cd tabtk/
# no configure in the directory
make
# no installation procedure defined in makefile
# just copy the executable to a suitable location
sudo cp tabtk /usr/local/bin

# fastqc
cd ~/sw
wget http://www.bioinformatics.babraham.ac.uk/projects/fastqc/fastqc_v0.11.4.zip
unzip fastqc_v0.11.4.zip
rm fastqc_v0.11.4.zip
chmod +x FastQC/fastqc

# vcftools
cd ~/sw
sudo apt-get install pkg-config
wget -O - https://github.com/vcftools/vcftools/releases/download/v0.1.14/vcftools-0.1.14.tar.gz | ta
cd vcftools*
./configure
make
sudo make install

# and some more:
# bcftools, samtools, vcftools, htslib
```

Check what are the largest packages:

```
dpkg-query -Wf '${Installed-Size}\t${Package}\n' | sort -n
```

16.2.4 Sample datasets

Pull some data from SRA. This is prety difficult;)

```
sraget () { wget -O $1.sra http://sra-download.ncbi.nlm.nih.gov/srapub/$1 ;}
```

Use data from my nightingale project, subset the data for two selected chromosomes.

```
# see read counts for chromosomes
samtools view 41-map-smalt/alldup.bam | mawk '{cnt[$3]++;} END{for(c in cnt) print c, cnt[c];}' | sor
# extract readnames that mapped to chromosome 1 or chromosome Z
mkdir -p kurz/00-reads
samtools view 41-map-smalt/alldup.bam | mawk '($3 == "chr1" || $3 == "chrZ"){print $1;}' | sort > kur
parallel "fgrep -A 3 -f kurz/readnames {} | grep -v '^--$' > kurz/00-reads/{/}" ::: 10-mid-split/*.fa

# reduce the genome as well
# http://edwards.sdsu.edu/labsite/index.php/robert/381-perl-one-liner-to-extract-sequences-by-their-i
perl -ne 'if(/^>(\S+)/){$c=grep{/^$1$/}qw(chr1 chrZ)}print if $c' 51-liftover-all/lp2.fasta > kurz/20

# subset the vcf file with grep
# [the command got lost;]
```

Prepare the `/data` folder.

```
sudo mkdir /data
```

Transfer the files to the VirtualBox image, /data directory using WinSCP.

Do the quality checks:

```
cd /data/slavici
~/sw/FastQC/fastqc -o 04-read-qc --noextract 00-reads/*

# update the file database
sudo updatedb
```

16.2.5 Packing the image

Now shut down the VM and click in VirtualBox main window `File > Export appliance`. Upload the file to a file sharing service, and use the *goo.gl* url shortener to track the downloads.

16.3 Slide deck

Libor's slide deck was created using Adobe InDesign (you can get the CS2 version almost legally for free). Vasek's slide deck was created with Microsoft Powerpoint. Images are shamelessly taken from the internet, with the 'fair use for teaching' policy ;)

CHAPTER 17

Slide decks

`Unix - Introduction` (Libor)

`Unix - Basics` (Vasek)

`Graphics session` (Libor)

Old:

`Genomics session` (Vasek)

`Advanced UNIX session` (Vasek)

`Genomic tools` (Vasek)