Unix Tools for Bioinformatics (2015-06-02)

Alexander Jueterbock, Martin Jakt[*]

PhD course: High throughput sequencing of non-model organisms

Contents

Next Generation Sequencing (NGS) data sets are big and although the data is often text based you simply can't open and work with them in a word processor or Excel table - even the attempt to open a big fasta file in a text editor can freeze your computer. Rather more importantly, even if you could, you shouldn't as word processors are not the same as text editors. Word processors are concerned more with the formatting of text and will store a great deal of additional information in addition to the text that you see. This additional information is irrelevant to the data and will in the best case scenario make further analysis impossible. Similarly try to avoid spreadsheets of various kinds as these almost always will try to interpret and convert your data to something like dates or currencies or something similar.

At first, you might feel uncomfortable with the command line but it is a very efficient and powerful tool. Moreover, many programs to analyze NGS data can only be used from the command line and to run them, you need to know the basic syntax. This tutorial gets you familiar with some of the most basic command line tools and shows you how they allow you to transfer files and how to extract information from big files fast and easily. All tools and programs are open source - so you will have free access to them also at your home institution or on your private computer.

1 Remote connection

To connect remotely to one of our computers, you will get:

- A password (e.g. PWD213)
- A username (e.g. user)
- An IP address (e.g. 127.0.0.1)

[*] University of Nordland, Norway

On 'Mac' and 'Linux' computers, the `ssh` (*Secure Shell*) tool is generally installed by default. To connect to a remote server, thus, just open the command line/terminal and type:

```
ssh user@host
```

Replace 'user' and the IP address 'host' with your own username and the IP address of one of our remote computers.

- Computer 1: 158.39.60.180
- Computer 2: 158.39.31.63

If this is the first time you want to access the remote server, you will likely get the following warning

```
The authenticity of host '127.0.0.1 (127.0.0.1)' can't be established.
ECDSA key fingerprint is 38:59:f7:22:e5:85:ec:c3:9c:90:7x:c3:e4:ae:88:18.
Are you sure you want to continue connecting (yes/no)?
```

Just type 'yes' and hit enter[1]. You will be asked to enter your password. You will not see the password when you enter it, so just type it blindly and hit enter.

```
user@127.0.0.1's password:
```

When you connected successfully, you will see a *welcome-text* similar to:

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-35-generic x86_64)
```

If you don't want to only use the command line on the remote host, but also want to use the Graphical User Interface (GUI) to open figures, GUI-based programs or web browsers, you should add the -X option to your command:

```
ssh -X user@host
```

If you use PUTTY on Windows you need to install Xming (http://www.straightrunning.com/XmingNotes/) and follow the guidelines on http://www.geo.mtu.edu/geoschem/docs/putty_install.html).

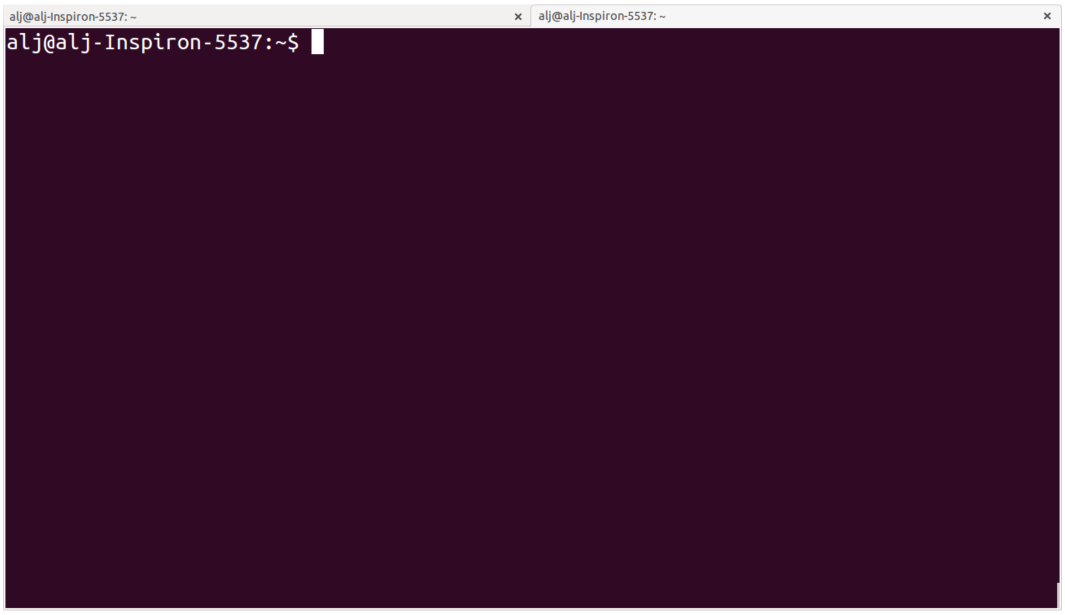
To open applications with a graphical user interface remotely on a MAC, you need to install XQuartz.

[1] This is an oversimplification. In general you should not simply ignore warnings like this, but it's too much off topic for us to explain here. Note though, that you shouldn't see this warning more than once, and if you do, you might want to read up on 'man-in-the-middle attacks'.

There are also a range of other options that let you access Unix/Linux servers from Macs or Windows that will let you use the command line. However, these often require specific software to be installed on the server and you can't rely on this being the case. In general, life is just easier if you run Linux on your own computer; it is acceptable if you run MacOS and just painful if you're stuck on Windows[2].

2 Basic orientation in the command line

The command line appears as a window/terminal similar to Fig. fig:terminal:



```
alj@alj-Inspiron-5537:~$ 
```

Figure 1: Command line window

The first line starts with your user name, followed by an `@` and then the name of the computer you are working on[3]. The line usually ends with a `$` or `>` - after this sign you can enter your commands.

The Unix cheat sheet (see at the end of this file) provides an overview of the core commands to navigate and operate in the Unix command line. Most commands allow you to adjust their behaviour with a variety of so-called arguments or flags. Most of the commands, for example, display help information if you use them with the `--help` flag.

For example, if you type `ls --help` [4], you'll get an overview of the common usage of the command `ls` and of the flags that can change the behaviour of this command. The `--help` option doesn't provide much help for the `ssh` command. In such cases you can try the `man` command. It opens

[2] This is part opinion and part fact. There are ways to use Windows to communicate with Unix machines that are not painful and in many ways the situation is improving. But the statement is nevertheless pretty much true and if you are going to spend some time doing informatics you might as well get rid of Windows as soon as you can.

[3] The beginning of the command line is referred to as the 'prompt' and like many things, it can be changed by changing an environment variable (in this case the `PS1` variable).

[4] Complete the command by presseing the `Enter` key (also called the return key and often denoted by down and left arrow).

a manual page of the specified tool. For example, try `man ssh`. If you want to leave the manual page, just hit `q`. To search within the `man` program, simply type `/` followed by your search term and hit enter. For example if you want to find something about colour just type `/color` and hit enter[5]. For more details try `man man` Before we will work on some sequencing data, let's have a look at the commands that allow you to change directories and how to get an overview of files that were saved in these directories.

2.1 Directory navigation

Navigating through your directories[5] is a big hurdle if you are new to the command line and are used to 'clicking' your way in and out of folders. To understand how to move in and out of folders (directories on Unix/Linux) and to look at the content of folders is an essential step to analyse your data on the command line.

2.1.1 Conventional directory layout

> On a UNIX system, everything is a file; if something is not a file, it is a process.

This is a simplification but it is mostly true. A directory on a Unix system is just a file that contains names of other files. Also programs, images and documents are all files. These files are hierarchically ordered in a conventional tree-like structure (see Fig. fig:linuxfiletree)

The root (represented as `/`) is the top-most level of this hierarchy. When you connect to a Unix computer, you are automatically located in your user home directory (`/home/username/`) and this is the only one you have write access rights to in this course. Many of the programs and scripts that you will use in this tutorial are located in the `/usr/` directory, generally in `/usr/local/bin/`.

Applications that are located in `/usr/local/bin/` can usually be run by any user by simply typing the name of the application since this directory is automatically specified in the so-called `PATH` variable of every user. The `PATH` variable is simply a variable that specifies the directories where executable programs are located. You will meet this `PATH` variable when you learn more about running programs.

2.1.2 Moving in and out of directories with `cd`

`cd` stands for 'change directory'. with this command you can navigate in and out of your directories. To see what your present working directory is, simply type `pwd` (for 'present working directory') and hit enter

```
pwd
```

[5] Directories are what you might call folders. It's probably best to think of the file system as a hierarchical way of organising your files on the computer. Think of it as a tree with the branches representing directories and leaves files (although in a file system, leaves can also grow directly from the trunk or main branches).

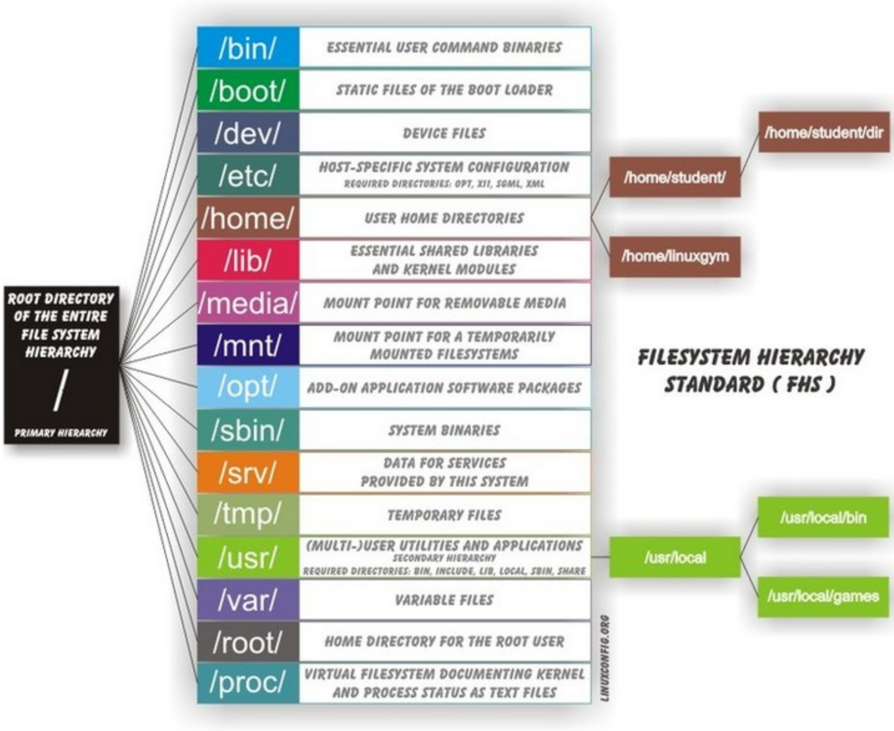

ROOT DIRECTORY OF THE ENTIRE FILE SYSTEM HIERARCHY
/
PRIMARY HIERARCHY
/bin/
ESSENTIAL USER COMMAND BINARIES
/boot/
STATIC FILES OF THE BOOT LOADER
/dev/
DEVICE FILES
/etc/
HOST-SPECIFIC SYSTEM CONFIGURATION
REQUIRED DIRECTORIES: OPT, X11, SGML, XML
/home/
USER HOME DIRECTORIES
/lib/
ESSENTIAL SHARED LIBRARIES AND KERNEL MODULES
/media/
MOUNT POINT FOR REMOVABLE MEDIA
/mnt/
MOUNT POINT FOR A TEMPORARILY MOUNTED FILESYSTEMS
/opt/
ADD-ON APPLICATION SOFTWARE PACKAGES
/sbin/
SYSTEM BINARIES
/srv/
DATA FOR SERVICES PROVIDED BY THIS SYSTEM
/tmp/
TEMPORARY FILES
/usr/
(MULTI-)USER UTILITIES AND APPLICATIONS
SECONDARY HIERARCHY
REQUIRED DIRECTORIES: BIN, INCLUDE, LIB, LOCAL, SBIN, SHARE
/var/
VARIABLE FILES
/root/
HOME DIRECTORY FOR THE ROOT USER
/proc/
VIRTUAL FILESYSTEM DOCUMENTING KERNEL AND PROCESS STATUS AS TEXT FILES
/home/student/
/home/student/dir
/home/linuxgym
FILESYSTEM HIERARCHY STANDARD (FHS)
/usr/local
/usr/local/bin
/usr/local/games
LINUXCONFIG.ORG

Figure 2: Conventional file tree structure on a UNIX system from linuxconfig.org

The response in my case is:

```
/home/alj/
```

When you want to 'move' to a different directory, the TAB key comes in handy as it auto-completes the possible sub-directories you can 'move' to. For example, when you type `cd` and hit the TAB key twice, you get an overview of all possible sub-directories. For example,

```
cd
.adobe/
Adobe/
.cabal/
.cache/
.compiz/
.config/
--More--
```

Hit ENTER to see more sub-directories in your shell or 'n' to leave the list of sub-directories.

If you know that your target sub-directory starts with the letters 'Do', you can type these after the `cd` command and then hit TAB twice (once is enough if there is only one sub-directory that starts with the letters 'Do'):

```
cd Do
```

```
Documents/ Downloads/
```

I, for example, have two directories starting with ‘Do’, `Documents` and `Downloads`. So, TAB completion helps when moving into sub-directories, but how to get out of them again? With

```
cd ..
```

you move one level up in your hierarchical directory structure. If you want to go to your home directory from wherever you are, use

```
cd ~
```

or just

```
cd
```

2.1.3 Tip

If there are empty spaces in your filepath, you need to precede them with a backslash (=)̄ in order to navigate to them, like in

```
/home/my\ directory/
```

or use quotation marks

```
cd "/home/my directory"
```

2.1.4 What files are present in the current directory?

Once you navigated with `cd` to your target directory and you want to look at the files and sub-directories that are located in it, you can use the command `ls` and hit enter. The tool `ls` comes with many options that refine the way that the results are shown; you get an overview of these options with:

```
ls --help
```

The combination of options that I use most frequently are

```
ls -lhcrta
```

The option

- `-l` provides additional information to the file or folder name
  - file permissions
  - user and group owners of the file
  - size
  - date of modification
- `-h` prints the file sizes in human readable format, like 100K instead of 102400
- `-c` sort by last modification of file status information
- `-r` reverse order while sorting (so that the newest files are the last that are printed)
- `-t` sort by modification time, newest first
- `-a` prints also the hidden files (starting with a dot '.').

2.1.5 Have a look at the directory structure

The command line tool `tree` prints the hierarchical structure of your files and directories (recursing into all sub-directories) to the screen.

To discriminate files from folders via colors, use the `-C` option

```
tree -C
```

To show only directories, use the `-d` option

```
tree -d
```

Try also the following command:

```
tree -sh
```

Here,

- `-s` provides the file and directory sizes
- `-h` prints the sizes in a human readable format

2.1.6 Tip

Besides the TAB-key, that allows for auto-completion of commands or filenames, the UP- and DOWN-arrow keys on your keyboard can save you some time. These buttons allow you to navigate through the history of commands that you have entered previously. Try it out.

2.1.7 Create, move and remove files and directories

New directories can be created with

```
mkdir directoryname
```

Here, `directory` is the name of the directory you want to create.

To create a new empty file, use the command `touch`:

```
touch filename
```

You can move or rename files with the command `mv`. For example:

```
mv file1 file2
mv file1 ../file1
```

The first command renames file1 to file2. The second command moves file1 one folder up[6].

If you don't want to move but copy a file, use the command `cp`.

```
cp file1 file2
```

Instead of renaming file1 to file2, as the `mv` command does, the `cp` command keeps file1 and creates a new file2 with the same content.

The most dangerous command that you learn to day is `rm`, which stands for remove. If you remove a file with this command, it is gone and you can not retrieve it. But if this is what you want, you can remove, for example, file2 that we created above with the following command:

```
rm file2
```

To remove an entire directory, use `rm` with the `-r` flag, like:

```
rm -r directoryname
```

[6] In unix, the `..` notation indicates the containing folder (i.e. one up in the hierarchy).

2.1.8 Tip

To get an overview of all the commands that you have used before, just type

```
history
```

and hit ENTER.

2.2 Data transfer between computers

Before you can work on a remote server with your own data, you first need to know how to transfer them. One of the best platform-independent GUI programs that allows you to up- and download files is FileZilla (Download and Documentation: https://filezilla-project.org/). In the following lines I want to introduce the command line tools `rsync` and `sftp/lftp`, that allow you to transfer and synchronize files.

2.2.1 rsync

`rsync` stands for "remote sync". This powerful tool has plenty of options. Here is the most basic syntax to transfer files from a *source* (SRC) location to a *destination* (DEST) with `rsync`. (Text in square brackets denotes optional arguments, in this case optional options!)

```
rsync [OPTIONS] SRC DEST
```

SRC and DEST can either be files or folders. For example, to transfer the file 'file.txt' from your local home folder to a remote server, you can type:

```
rsync --progress /home/user/directory/file.txt user@host://home/user/
```

Here, you need to change `/home/user/directory/` to your own filepath and `file.txt` to your own filename. In '=user@host=', `user` represents your username on the remote server and `host` the IP address of the remote server. The `--progress` option will indicate the progress of the file transfer - which is useful when transferring big files.

If you want to transfer files from the remote server to your local computer, just swap the source and destination path specifications:

```
rsync --progress  user@host://home/user/file.txt /home/user/directory/
```

If you want to transfer all files that are located in your local folder `/home/user/directory/`, you can use the following command

```
rsync -avz --progress /home/user/directory/ user@host://home/user/
```

Here,

- `-av` will transfer the files in 'archive mode' (which combines several options, including recursing into directories)
- `-z` will compress the files durig the transfer

Note the trailing slash after the source directory: `/home/user/directory/`. If you do not use this trailing slash, like `/home/user/directory`, then `rsync` will create a folder with the name `directory` at the destination and copy all files from the source folder into it.

Ok, that's all we need to know to get the sequencing data from last week to the remote computer. As we need the data in the following tutorials, it is best if you upload them now.

2.2.2 sftp/lftp

rsync is a wonderful tool, but its power makes it complex and it can be difficult to remember how to do even simple things (try `man rsync` if you don't believe me!). When using rsync you also need to know and remember where the files and directories that you wish to synchronise are located. My preference is for using the programs similar to the old ftp command line client (which even Windows has). This provides an environment very similar to the normal Unix shell, where you change directory using `cd`, list the files using `ls`, find out where you are using `pwd` and so on. However, the ftp protocol is inherently insecure; it may not matter that the data is transmitted without encryption, but you should be concerned about sending your password in plain text across the ethernet. Not good. Hence, these days we use the sftp (secure file transfer protocol) instead. On Mac and Unix systems you will essentially always have the sftp command line client installed. On Windows, well, you can use Putty or other third party tools. On Linux systems you may also have the lftp command line client installed. Its usage is almost identical to the usual sftp and ftp clients but it comes with extended functionality that allows you for example to mirror (i.e. synchronise directories) between the remote and local computers.

To use the sftp program, simply type:

```
sftp hostname
```

into your terminal. The hostname may need to be specified as the IP address (a load of numbers) or can be a simple name depending on your setup. After the connection is made you will be asked for your password. The sftp program assumes that you will be using the same username as you are using on the local computer. If this is not the case you can specify your username by:

```
sftp username@hostname
```

After having successfully logged in to the remote computer you can move around the directories as if you were logged in over a shell session (i.e. using `ls`, `cd` and so on). If you wish to change the directory on the local machine, simply use the `lcd` command. You can also run commands in your local shell by prefixing these with an !, eg. `!ls` or `!pwd`. You can create directories on the remote computer with `mkdir`, and on the local machine with `!mkdir`. To transfer files from the remote to the local computer use `get fname`. You can use globbing (*) to expand the file set, eg. `get *.fa` for all files ending in '.fa'. (For this you may need to use `mget *.fa` on some implementations, this used to be true on the old ftp command line client). Similarly you can upload files using `put`.

As mentioned lftp is almost identical in its operation. However, when starting the program you need to specify that you wish to use the sftp protocol as it defaults to the standard ftp protocol (with an anonymous user). Hence use something like:

```
lftp sftp://username@hostname
```

lftp also allows you to mirror whole directory structures using the `mirror` command which can save you a lot of time. Finally, when I started using lftp, the standard ftp and sftp clients did not provide tab completion, and this was a big advantage of lftp at that time. These days most if not all of the clients provide this functionality, so it is not quite as big a deal as it was in the long past.

2.2.3 Tip

If you want to transfer in one go, all files that have some common characteristic in their name you can use the asterisk `*`, which stands for 'any character'. The `*` is one of the most commonly used wildcard symbols that stands for a continuous string of characters. To specify a set of filenames with wildcard characters is also referred to as *globbing*.

For example, if you want to transfer all fasta files at once, you can use

```
rsync -avz --progress /home/user/directory/*fasta user@host://home/user/
```

This means that any characters can precede the `fasta` file ending.

If you want to transfer all files that belong to a certain population and are, for example, marked with 'Pop1' in the file name, you can use:

```
rsync -avz --progress /home/user/directory/*Pop1* user@host://home/user/
```

This means that any characters can precede or follow the `Pop1` character in the file name.

3 Running programs (and the PATH variable)

3.1 From the shell

When using the shell you normally run a program by simply typing the program name and any required arguments. But how does the shell know what program to run and where to find it? On a typical Unix / Linux system executable files (i.e. programs) can be found in a range of standard locations (eg. `/bin/, /sbin/, /usr/bin/, ~/bin/`) as well as anywhere a user puts them. Normally when you run a program by simply typing its name, the shell will look for an executable file of that name in a list of directories specified by the `$PATH` environment variable. The first matching program is then run.

The user can also directly specify the location (path) of the executable; this is necessary if the program you wish to run is not present in any directory specified by the `$PATH` variable, or if multiple programs of the same name are present and you want to run one of the later matches:

```
/usr/local/bin/pg_ctl start
```

to start a version of the Postgresql database installed in /usr/local/bin specifically.

You can also specify a path that is relative to your current location. If for example your current working directory is `~/Documents/testPrograms/` and you wish to run a locally installed version of gcc (gnu C compiler) found in `~/bin/`[7]:

```
../../bin/gcc -o test main.c
```

(Remembering that ../ takes you up one level in the directory structure). To do the same you could also make sure that the `$PATH` contains ~/bin before other potential locations of gcc.

To check the current value of your `$PATH`, simply use the echo command:

```
echo $PATH
```

To learn how to extend your own PATH variable have a look in the hidden .basrhc or .bash$_{\text{profile}}$ file in your home directory. It usually gives a few examples. Failing that have a look at Google.

Finally if you've written a small script or installed a program in your current working directory you can run that by typing `./scriptname`. There is nothing special about that, it is merely how you represent the relative path to your current working directory.[8]

[7] The ~ (tilde) character is used as shorthand for your home directory.

[8] Previously you learnt that `../` represents the containing directory (one level up); the `..` is simply shortcut for the current working directory.

3.2 From a shell script

You may have noticed that the commands to run sequence analysis programs can be quite long and complex. This is because the program allows you to specify many different options directly when invoking the program rather than by interactively asking you through some sort of interface. This is convenient, as it allows you to do something else as soon as you have invoked the program. However, it can be difficult to remember the exact details of how to run something that you do not use every day. Fortunately you do not need to remember this yourself; there is a better way: instead of running the program from the command line you create a small text file that contains the commands that you wish to run, and then you ask the shell (i.e. the command line interface) to run the commands contained within your text file. Since your file can also contain comments this allows you to add explanations as to what the command does and why you chose to run it that way.

For example we could just type the following into the terminal to map sequences in `seqData.fastq` to the genome provided in `~/apps/STAR/STAR/genomeDir/Zv10/unmasked/` using the STAR program:

```
STAR --outFilterMultimapNmax 5 \
--outFilterMismatchNoverLmax 0.05 --outFilterIntronMotifs RemoveNoncanonicalUnannotated \
--genomeDir ~/apps/STAR/STAR/genomeDir/Zv10/unmasked/ --readFilesIn seqData.fastq \
--runThreadN 6 --outSAMtype BAM SortedByCoordinate
```

But that is both difficult to remember and to type fully without making any mistakes. It is much easier to create a small text file, maybe called, `map_seq.sh` [9] that contains the code with some comments to explain what does what:

```
#!/bin/bash

## this command will use STAR to map the sequences in seqData.fastq to
## the unmasked version of the Zv10 (zebra fish) genome sequence

STAR --outFilterMultimapNmax 5 \
--outFilterMismatchNoverLmax 0.05 --outFilterIntronMotifs RemoveNoncanonicalUnannotated \
--genomeDir ~/apps/STAR/STAR/genomeDir/Zv10/unmasked/ --readFilesIn seqData.fastq \
--runThreadN 6 --outSAMtype BAM SortedByCoordinate

## --outFilterMismatchNoverLmax : the maximum proprotion of mismatches
## --outFilterIntronMotifs RemoveNoncanonicalUnannotated : don't report
##   weird splice positions
```

You can now run this script in two different ways. Either first make it executable by changing its permissions: `chmod +x map_seq.sh` and then run it from the directory where it exists, `./map_seq.sh`. Alternatively you can invoke the bash shell on the script by typing `bash`

[9] Shell scripts are normally named with a `.sh` extension; this makes it easy to identify them. However, Unix doesn't care, and you are free to name the scripts as you like.

map_seq.sh and hitting enter. In the latter case you do not need the first line of the script #!/bin/bash, but it is usually included for completeness. Note that comments in shell scripts are preceded by the # character. Anything that follows this on the same line is considered as a comment and will not be executed by the shell.

Writing a script file to run a single command like this may not look like it helps you very much, but there are a number of advantages over simply writing the command directly:

- You can spend some time to proof-read the command making sure you don't have any stupid mistakes in it.
- You have a record of how you ran the program. This is actually very important.
- You can look at this file next time you have something to map and remind you of how you use the program.

However, the real beauty of making a shell script is that the shell provides a simple programming environment and this means that you can automate the processing of large numbers of files. For example if I wish to run the above command in a set of sub-directories whose names all start with RZY-3 I can use the following script:

```
#!/bin/bash

dirs=RZY-3*

for d in $dirs
do
    echo "entering directory: $d"
    echo
    cd $d
    f=$d.fastq
    echo "Processing: $f $(date +'%b %D %H:%M:%S')"

##    STAR --outSAMstrandField intronMotif --outFilterMultimapNmax 5 \
##    we have directional libraries and should not use the
##    outSAMstrandField intronMotif function
    STAR --outFilterMultimapNmax 5 \
         --outFilterMismatchNoverLmax 0.05 \
         --outFilterIntronMotifs RemoveNoncanonicalUnannotated \
         --genomeDir ~/apps/STAR/STAR/genomeDir/Zv10/unmasked/ \
         --readFilesIn $f --runThreadN 6 \
         --outSAMtype BAM SortedByCoordinate

    echo "      Done: $f $(date +'%b %D %H:%M:%S')"
    cd ..
done
```

Here we first use a wild card notation dirs\=RZY-3* to create a list of directory names. We then use a loop (for d in $dirs) to go through each directory in turn. For each directory we first use cd to change our current working directory, then assign the the file name (f=$d.fastq, in this case the filenames all have the same prefix as the directory name, so we can simply concatenate the dirname and the .fastq suffix together) to the $f variable and then call the command to run the program. After STAR returns (i.e. it has finished running) we go back to the starting directory (cd ..) and go to the next one.

When we define a variable we simply write the name of it without any decoration; (eg. f=$d.fastq to define the $f variable). However, when we use the variable we have to decorate it with the $ sign.

We use echo throughout the script to print information about the process of the script. This ouput, in addition to that printed by the STAR program can be redirected to a file (a log) so that we can leave the process to run and then come back and see if there were any errors encountered during the processing. To run the script, first chmod it to an executable, then:

```
./map_seq.sh > map.log &
```

This will run the commands in the shell script and redirect STDOUT to the file map.log. The & at the end of the line returns control back to the terminal so that you can do other things rather than just waiting for the process to complete. If you are using a remote computer you should probably do :

```
nohup ./map_seq.sh > map.log &
```

as that will now allow you to log off, and do whatever you like. As long the server doesn't crash (or an administrator stops your process) the process will complete without any input from you.

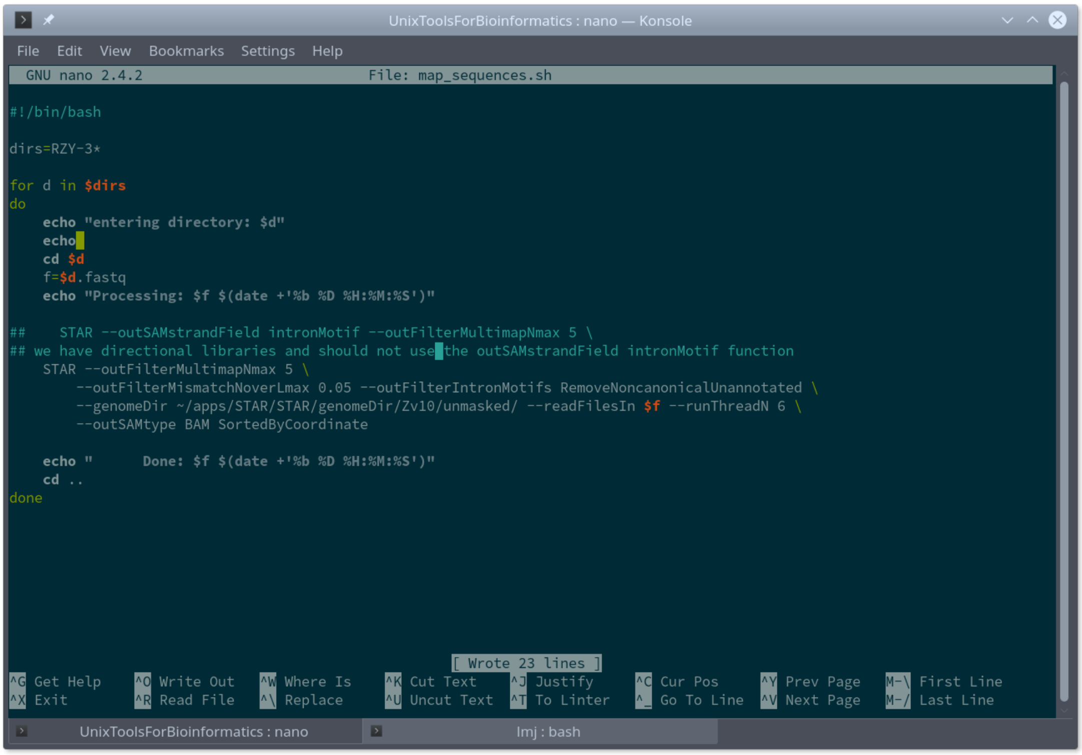
If you wish to follow the process of the mapping, you can use the tail program to view the output of the shell script. If you invoke it with the -f option it will follow the file you specify, printing the new lines of it as it is written to:

```
tail -f map.log
```

To stop following the file you will need to hit Ctrl-C (i.e. the control and C keys simultaneously).

3.3 Writing a shell script

To write a shell script you will need to use some sort of editor. There are very many to chose from; for experienced Unix / Linux users the two most commonly used editors are probably emacs and vi (these days perhaps vim). These editors are very powerful tools for writing computer code, but can take a certain amount of time to learn. For this course we recommend you to try the nano editor. To edit or create a new file with nano simply type nano filename and start typing. Commands to save, exit, cut, paste, etc. are indicated at the bottom of the screen. ^ is shorthand for the Ctrl key.



UnixToolsForBioinformatics : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.4.2
File: map_sequences.sh

```
#!/bin/bash

dirs=RZY-3*

for d in $dirs
do
    echo "entering directory: $d"
    echo
    cd $d
    f=$d.fastq
    echo "Processing: $f $(date +'%b %D %H:%M:%S')"

##    STAR --outSAMstrandField intronMotif --outFilterMultimapNmax 5 \
## we have directional libraries and should not use the outSAMstrandField intronMotif function
    STAR --outFilterMultimapNmax 5 \
        --outFilterMismatchNoverLmax 0.05 --outFilterIntronMotifs RemoveNoncanonicalUnannotated \
        --genomeDir ~/apps/STAR/STAR/genomeDir/Zv10/unmasked/ --readFilesIn $f --runThreadN 6 \
        --outSAMtype BAM SortedByCoordinate

    echo "      Done: $f $(date +'%b %D %H:%M:%S')"
    cd ..
done
```

```
                                                      [ Wrote 23 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    ^Y Prev Page  M-\ First Line
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Linter  ^_ Go To Line ^V Next Page  M-/ Last Line
```

4 Retrieving basic information from common NGS files

Now that we know how the commandline works, how we can change directories and transfer files, it's time to look at NGS data output and to learn how to open and summarize information from such files - like, for example, the number of sequences in a fasta file.

The folder PracticeFiles contains the following files:

- HTS.fasta and HTS2.fasta, fasta files with sequence identifiers and sequences
- HTS.fastq, a file with sequences and associated base qualities
- HTS.sam, an alignment file

4.1 Look at the content of a file and search for patterns

The tool `less`[10] can be used to display the content of text files one line or page after the other. Since it doesn't read the entire content of a file at once, it is very useful for looking into large files.

Let's have a look at a fastq file with the command:

[10] `less` is very similar to the more basic program `more`. It's name is a bit of a joke on 'less is more'. Habits die hard, and at least one of the authors of this document has `more` hardcoded into his fingers.

```
less Fastqfile.fastq
```

Once you have opened a fasta file, for example, with `less` ...

```
less Fastqfile.fasta
```

... you can search for patterns, like the nucleotide sequence ‘GCTC’, with `/`, like

```
/GCTC
```

hitting `n` repeats this search on the remainder of the file.

To show only those lines in the file that match the nucleotide sequence ‘GCTC’, type this sequence after the `&` sign:

```
&GCTC
```

To go to the last line of the file, just type `G`, to go to the first line, type `g`. To close the file again, hit `q`.

The `less` command has more options than this. You get an overview of these with the `--help` flag:

```
less --help
```

The `head` command, followed by the name of a text file, prints by default the first 10 lines/rows of the file to the terminal. The `-n` option allows to determine the number of rows that shall be printed. For example, to extract the first sequence-id along with the nucleotide sequence from HTS.fasta, you can select the first two lines with:

```
head -n 2 HTS.fasta
```

When the line number `K` is preceded with `-`, then all but the last `K` lines are printed. For example, the command to print all but the last ten lines from a HTS.fasta is:

```
head -n -10 HTS.fasta
```

The `tail` command, in contrast, prints by default the last 10 lines of a file to the terminal. Also here you can select the number of lines with the `-n` option. When the line number `K` is preceded by a `+`, then all but the first `K` lines are printed. For example, to exclude the first two lines from HTS.fasta

```
tail -n +2 HTS.fasta
```

To extract specific lines from a file, the tool sed can help you. To print all lines between line 234 and 236 from HTS.fasta, for example, use:

```
sed -n '234,236p'
```

4.2 Counting words, lines, and characters with 'wc' and searching for patterns with 'grep'

If you want to get a rapid overview of the number of lines in a file, the wc command is the right tool. In output-files where every line represents a sequence, for example, wc -l is all you need to count the number of sequences.

```
wc -l File.txt
```

The -l option specifies that you want to count the number of lines. The -m and -w options further allow you to count the number of characters or words.

To count the number of sequences in a fasta file, you have to limit the lines that are counted to those starting with a ">" sign because ">" precedes every sequence identifier:

```
>SEQ1_ID
GGATTCATAGAAACCATAGATACATAGATACATAGATTAGGGACAGATAATAG
>SEQ2_ID
GATTTGGGGTTCAAATTAGTATCGATCAAATAGTAAATCCATTTGTTCAACTC
>SEQ3_ID
AGATACAGAGAGACAAGACATAGACAGATAACAGAATAGAGATAGAGGAGAGG
```

grep allows you to extract lines that contain specific characters, like ">".

If you type

```
grep ">" HTS.fasta
```

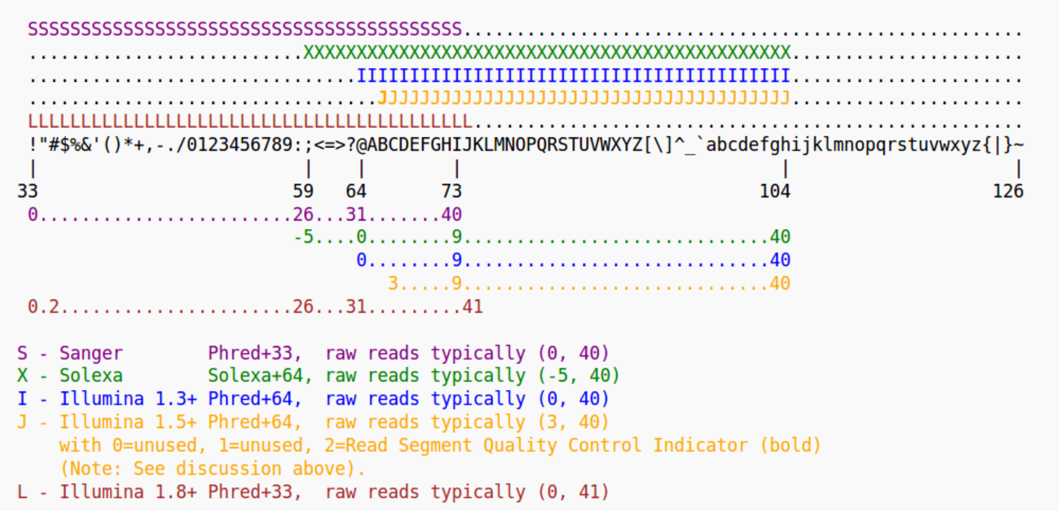
All lines in HTS.fasta that contain the ">" character are printed to the screen. You can stop the flow of output by pressing Ctrl+C. If you don't want to write these lines to the screen but want to count them, the | symbol provides a 'pipe' to pass the output from the grep command to the wc command. So, to count the number of sequences in HTS.fasta, you can use the following command:

```
grep ">" HTS.fasta | wc -l
```

Here a recap on what the commands mean: `grep` is used to search for `>` signs in the fasta file. All sequence-id's start with this character. Instead of printing all these lines to the terminal, we re-direct it to the `wc` command with the pipe symbol `|`. Using the `-l` option, `wc` counts all the lines. Here, `wc` doesn't need an input file as it reads from the output of `grep` [11].

Your turn. What command would you use to count the number of sequences in a fastq file?

If you are in doubt what quality encoding your fastq file has, `grep` can help you. Have a look at Fig. Fig:QC. If you find one of the ASCII characters 33 (character'!') to 58 (character ':'), you can be sure that the quality encoding is Phred+33.



```
  SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS.....................................................
  ..........................XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX......................
  ...............................IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII......................
  .................................JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ......................
  LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL....................................................
  !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
  |                         |    |        |                              |                     |
 33                        59   64       73                            104                   126
  0........................26...31.......40
                           -5....0........9.............................40
                                 0........9.............................40
                                    3.....9.............................40
  0.2......................26...31........41

 S - Sanger        Phred+33,  raw reads typically (0, 40)
 X - Solexa        Solexa+64, raw reads typically (-5, 40)
 I - Illumina 1.3+ Phred+64,  raw reads typically (0, 40)
 J - Illumina 1.5+ Phred+64,  raw reads typically (3, 40)
     with 0=unused, 1=unused, 2=Read Segment Quality Control Indicator (bold)
     (Note: See discussion above).
 L - Illumina 1.8+ Phred+33,  raw reads typically (0, 41)
```

Figure 3: The Nano editor. Quality score encodings

So, try if you find one of the Phred+33-specific quality characters in HTS.fastq. For example:

```
grep "!" HTS.fastq | wc -l
```

`grep` also allows you to search for the sequence of a specific gene-id and identify the line of the hit in a fasta file, if you use it with the `-n` flag. For example, if you want to know which line in the HTS.fasta file holds the sequence with the gene-id 'gi|612475216|gb|AZHG01011862.1|', you can use:

```
grep -n "gi|612475216|gb|AZHG01011862.1|" HTS.fasta
```

It is line 23724.

[11] When a program prints it's output to the terminal (i.e. the screen) it's normally printing to a stream referred to as `STDOUT` (standard out). When we use the pipe symbol (`|`) we can redirect this output to programs than can read from the `STDIN` stream. We can also use the `>` to redirect the output to files. Note that output printed to the `STDERR` stream will also be printed to the terminal, but will not be redirected using `|` or `>` (though you can use `2>` to redirect `STDERR`).

4.3 INFO on regular expressions

`grep` stands for *global regular expression printer* and is a command-line utility for searching plain-text data for lines matching a regular expression. With regular expressions you can match strings that are not identical but follow a specified pattern. We won't go into further detail here, but you can read more about regular expressions in A Tao of Regular Expressions and you can find a short introduction in the Perl section below. Also, here you will find a cheat sheet with essential regular expressions.

4.4 Combine the content of files with 'cat' and '>'

The most common use of the `cat` command is to redirect the contents of text files to other files or commands.

The following command, for example prints the content of HTS.fasta to the screen

```
cat HTS.fasta
```

With the > and » operators, you can print the content of files not to the screen but to other files. This allows you to rapidly combine two files, even huge ones. For example, in the following command `HTS.fasta` and `HTS2.fasta` are combined to `COMBINED.fasta`.

```
cat HTS.fasta > COMBINED.fasta
cat HTS2.fasta >> COMBINED.fasta
```

The > operator redirects the output of the `cat HTS.fasta` command (the content of `HTS.fasta`) to `COMBINED.fasta`. The » operator adds the output of the `cat HTS2.fasta` command to the `COMBINED.fasta`. If we would use the > operator instead of the » operator in the second line, the content of `COMBINED.fasta` file would be overwritten, not appended. So, the > operator (over) writes content to a specified file while the » operator appends content to a specified file. If you use the » operator, the specified file needs to exist already.

Note that you can achieve the same by:

```
cat HST.fasta HTS2.fasta > COMBINED.fasta
```

but we wanted to show you the difference between > and ».

4.5 Counting filtered reads in SAM files with 'awk'

Later in the course we will encounter specific programs that can filter SAM and VCF files. Here, I want to show you that we can also use basic command line tools to filter such files. The command line tool awk can extract single columns or apply a filter on column values in any file that is organized in columns - as SAM and VCF files are. The -F option allows you to specify if your columns are delimited by commas, spaces, tabs or any other character.

We learned this morning that SAM files (alignment files) are tab-delimited (\t and always contain the mapping quality in the fifth column ($5). Thus, to count mappings in a SAM file that have qualities > 20, we first strip off the header lines containing the @ character with grep:

```
grep -v "^@" HTS.sam
```

Here, the -v option inverts our search (all lines including @ at the beginning of the line - specified by the ^ sign - are excluded).

The above command would print all non-header lines to the screen. Instead, we want to pipe the output of this command to awk, in order to extract only those reads with a mapping quality >20 and then pipe this output to wc to count the lines:

```
grep -v "^@" HTS.sam | awk -F "\t" '$5 > 20 {print $0}' | wc -l
```

Here, $0 refers to the entire row, while $5 refers to column 5 of that row. -F just specifies the field separator, and \t sets it to the TAB character. Since we pipe (using |) the output of grep to awk, and then the ouput of awk to wc the lines are not printed to screen but directly counted with the wc command. Only the output of wc gets printed to the screen.

4.5.1 Variables in Perl

In order to handle information within a program we assign values to variables and then manipulate these according to the flow of the program. Perl provides three different types of variables:

- Scalar variables: these take a single value (usually a number or some text) and are denoted by a $ prefix, eg. $var.
- Arrays: these contain an ordered series of values that are accessed by their position. Arrays are denoted by an @ prefix, eg. @array. Individual values are accessed as scalars, using square brackets to indicate the position, eg. $array[3] accesses the fourth element of @array (the fourth rather than the third as we count from 0).
- Hashes (or associative arrays): these hold key-value pairs and are denoted by the % prefix, eg. %hash. Individual elements are again accessed as scalars, but this time using curly brackets, eg. $hash{key}. The key value can be anything that can be assigned to a scalar (numbers, text, and references).

4.5.2 Assigning variables

The values of variables can be assigned directly in the program's source code, but are more frequently assigned through the command line arguments (see below) or by the program reading input (data or configuration) files (see lower section). Scalars are the simplest:

```
$var1='hello';
$var2="world";
$var3=3.14;
```

Strings (i.e. text elements) can be assigned using either single ' or double " quotes. The use of double quotes expands variables within the quoted text such that:

```
$var4="goodbye $var1";
```

will assign the text "goodbye world" to the variable `$var4`. In contrast:

```
$var4='goodbye $var1';
```

will assign the text 'goodbye $var1' to `$var4` (without the quotation marks!). Double quotes also allow escape codes such as `\n \t` to be interpreted as newline and tab characters respectively.

Arrays can be assigned in a number of ways, occassionally directly in the code:

```
@ar1 = (1, 2, "three");
```

An empty array can also be created and then extended by adding elements. This can be done by either using the `push` function or by using subscripts beyond the range of the array:

```
## text following a # character are treated as comments

@ar1 = (); ## creates an empty array of length 0
push @ar1, "hello"; ##extends this array to have a length of 1

$ar1[2] = "three";
## the array now has a length of three, but an undefined value in the second position
## $ar1[1]
```

In most cases, elements of an array will be assigned to values found in input files containing the data to be analysed, rather than being defined directly in the code as above.

Hashes (associative arrays) that store key value pairs are defined in a similar way to arrays. Again the actual values are usually obtained from input files, but can also be defined in the code.

```
%kv1 = ();
```

```
## this creates an empty hash structure. It is actually not necessary to
## declare it, but one can directly assign elements of the hash:
$kv1{1} = "one";
$kv1{2} = "two";
$kv1{'three'} = 3;

## this hash could also have been created in a single line :
%kv1 = (1 => "one", 2 => "two", 'three' => 3);

## to access the elements of an associative array we obtain
## the keys of the hash using the keys command.

@keys = keys %kv1;
## print the first value associated with the first key:
print "$keys[0] $kv1{$keys[0]}\n";

## the \n simply defines a newline character
```

Scalars, arrays and associative arrays can be combined to create arbitrarily complex data structures. Hence you can have hashes of arrays and arrays of hashes and so on. To fully use more complicated data structures requires an understanding of the reference. A reference is a value that points to another piece of data by providing the memory address of that data. For example, an array of hashes is encoded as an array of references to hashes. To obtain the value of data referred to by a reference the reference must be dereferenced. Perl has a number of different ways in which this can be done, but these will not be explained in depth here as it can get a bit messy.

Semicolons: you may have noticed that in the above examples almost every line ends with a semicolon. In Perl (and in many other languages), the semicolon is used to denote the end of statements. This means that single statements can be spread across several lines and that a single line can contain a number of statements. This can greatly aid the readability of the code.

4.5.3 Data types

In the above examples we assigned values to variables without caring about what kind of data we used. For example consider the following:

```
$var1 = "one";
$var2 = 2;
$var3 = $var1 + $var2;
```

Here we have assigned the value of `$var1` to a piece of text (which we will refer to as a string from here on) whereas `$var2` has been assigned a numeric value. Perl is a dynamically typed language; that means that you do not have to explicitly define what type of value a variable contains. This is convenient when writing a script (essentially a small program), but this does make it easier to make mistakes in more complicated situations. In the above example, the third line doesn't make sense, and will generate an error. In this case it is obvious from the code, but in most real world situations the values will be read in from an external file produced by some other program or person in which case finding the reason for the problem may not be so simple.

Perl essentially has three data types, strings, numeric values and references. References are necessary for making more complex data structures and to allow variable values to be modified by functions. As mentioned above though, references will not be covered in much depth as they are more suitable for a more advanced course. The string and numerical data types are fairly straightforward, though there are a few potential problems (common to essentially all computer programming):

- Numeric values do not have infinite precision. For example (1/3) is not equal to (0.1/0.3).
- Numeric values can not be arbitrarily large. On my machine the maximum value Perl can handle is somewhere between 1e308 and 1e309. That's a pretty large number which you might think you will never need. However, it is smaller than the factorial of 171, and this is something you may run across in statistical equations.
- Mathematical operations can result in illegal numbers, eg. 1/0. If your program carries out any calcuations you need to be aware of this and how Perl handles the resulting values.
- Text is actually not that simple. From the beginning, the end of lines has been encoded differently in Windows (i.e. DOS), MacOS and Unix. In Unix an end of line is encoded with a newline character, on Windows, a newline character followed by a carriage return, and on MacOS it might be just a carriage return (to be honest I forget). This can cause trouble as text files are usually written and read line by line (i.e. new lines indicate a new section of data). The simplest way to avoid trouble is simply never to use Macs or Windows machines, but that can be difficult at times.
- These days text encoding is rather complicated, as it has been expanded to cater to a range of languages and character sets (eg. Arabic, Chinese, Japanese, Thai, etc..). This is not straightforward and several conflicting encodings have been developed. For bioinformatics you usually do not have to care; but you have to be aware of potential problems when handling text that contains unstructured descriptive data. Such text may contain names, or places written in glyphs that require Unicode encoding. Such descriptions may even contain characters that look like normal roman letters, but which have been encoded differently. Google, 'halfwidth fullwidth characters' to confuse yourself.
- Sorting. Numbers and strings are obviously sorted differently. Consider that `(12 > 8)`, but `('12' < '8')`. In the latter case we are comparing strings through a lexicographic comparison where the first character is the most significant for the sort. Since 8 is larger than 1, "8" is also larger than "12". In Perl sorting is lexicographic by default, and a numeric sort has to be explicitly specified. This is sometimes problematic when a mix of numerical and character based identifiers are used and the reason that you often see the following chromosome ordering: 1,10,11,12,...,19,20,21,3,4,5,...,9,X,Y.

4.5.4 Program flow: loops and conditionals

We use computer programs to automate repeated processes; that is to carry out the same or similar operations on a large number of data points. This is (usually) done by iterating over a collection of data until some condition is met. That condition is often simply that we have no more pieces of data to look at, but the condition can also be that a solution to some problem has been found, or anything that you can think of. This process is referred to as looping.

Similarly programs need to be able to handle the data differently depending on what it is. This is handled by conditional statements. Conditional statements are also used in lots of other cases including to control loops. Consider the following statement that checks for the equality of two variables.

```
## $a and $b are two variables whose values are specified somewhere else in the program.
if($a == $b){
  ## then do something. For example increase the value of $b
  $b = $b + 1;
}
```

There are a few things to mention here. The first is the use of the ==== operator. This tests for numerical equality. It is very important not to confuse this with the === operator which assigns values. Comparison operators can be thought of as returning a TRUE or a FALSE value. If a TRUE value is obtained then the conditional statement is carried out, and if FALSE not. Perl doesn't actually have explicit TRUE and FALSE values, but any non-0 value is considered as TRUE and a value of 0 is considered as FALSE. To confuse things the use of the assignment operator returns the value that was assigned and this can cause some rather specific problems. Consider:

```
$a = ($b = 10);
## $a is now assigned to the value of 10

## this conditional statement will always evaluate to TRUE
if( $a = 25 ){
  ## this will always be executed
}

## but this will never evaluate to TRUE
if($a = 0){
  ## this part of the program will never be reached
}
```

The second thing to mention is the use of the curly brackets ({and}). In Perl (and quite a few other programming languages) these are used to break the code up into blocks of code that can be conditionally executed (or looped over, which is kind of conditional). In Perl, blocks of code can have their own scope by using the my keyword. This means that a variable which is defined within a block of code is not visible outside of that block of code. This is very useful for more complicated programs where it is easy to accidentally use the same variable names to represent different properties. Consider the following snippet:

```
## We start in the global scope. Variables defined here will be visible and modifiable
## anywhere within the main body of the code (though not in external functions).
```

```
$a = 10;
{
  $a = 20;
}

print "a is $a \n";
## will print 20. However if we do:

{
  my $a = 30;
  ## $a will be equal to 30 only within this block of code
}

print "a is now $a \n";
## does not print 30, as we $a was declared using the
## my keyword.
```

It is good practice to use `my` and the related `our` keyword throughout the code as it will make it easier to catch a range of different types of errors. This can be enforced by `use strict;`. Google for more!

Looping can be used if, for example you have an array of values that you wish to obtain the mean value of. To do this we wish to find the sum of the values and divide by the length of the array. As always in Perl there are a number of ways in which this can be done:

```
## @ar is an array of values specified somewhere else in the program.
## ++ is an increment operator that increases the value of its operand
## by one each time it is called.
## += is an increment operator that increases the value of its left operand
## by the value of its right operand.

## to loop through the values we can use a classic for loop:
$sum = 0;
for( $i=0; $i < @ar; $i++){
  $sum += $ar[$i];
}

## this will set the value of $i to 0, carry out the operation in the block
## following the for statement, then increment (increase) the value of
## $i by 1 and repeat as long as $i is less than the scalar value of
## @ar (which evaluates to the length of the array).

$mean = $sum / @ar;
## when an array variable is used in an expression it can can evaluate to either the array it
## or to a scalar value equal to its length. When it's not clear as to whether the scalar or
## value is indicated, the scalar value can be enforced by the scalar function.

## We can also use a range specified loop and make use of the fact that in Perl
## $#ar will evaluate to the higest index of an array (i.e. the length minus one)
```

```
for $i(0..$#ar){
  $sum += $ar[$i];
}

## we can also use a similar expression;
for $v(@ar){
  $sum += $v;
}

## alternatively we can use a while loop by specifying the index variable outside
## of the loop statement;
$i = 0;
while($i < @ar){
  $sum += $ar[$i];
  $i++;
}
```

These are not the only ways in which you can loop through values or data structures, but they probably represent the most common usages.

4.5.5 Reading and writing data

To read or write from a file we use a filehandle. This is just an identifier associated with the file and the reading or writing process. To write to a file we usually use the `print` function. Using `print` without specifying a filehandle will lead to the text being printed to STDOUT. In most cases this means your terminal screen, but STDOUT can also be piped to other processes as demonstrated previously in this guide. To open a text file and read a line at a time:

```
## we wish to read from a file specified by the variable $fname

open(IN, $fname) || die "unable to open $fname $!\n";
## here IN becomes specified as the filehandle (This is one of the few cases
## where we use an undecorated string literal as an identifier).
## The second half of the statement uses the '||' operator which simply means 'or'.
## If we are unable to open the file then the program will print out the warning statement
## following die and exit. $! is a magic variable that contains the error string.

## to read all of the lines we can make use of a while loop
while(<IN>){
  ## this will assign the text of each line to another magical variable, $_
  ## we can print this out to STDOUT by calling
  print;   ## without arguments this prints $_ to STDOUT

  ## normally we would do something useful first by processing the data in the line.
  ## but more of that later.
}
```

To write to a file we also use open, but modify the filename to indicate that we wish to write to a new file by prefixing the name with a ‘>’ character. If a file of the same name exists it will be overwritten. If we wish to append to an existing file we can use ‘»’.

```
## given that we wish to write something to a file specified by the
## $fname variable.
open(OUT, ">$fname") || die "unable to open $fname $!\n";
## write out the multiplication table (1..10) to the file
## first write out some column headers
for $i(1..10)\{
  print OUT "\t$i";
}
print OUT "\n";

for $i(1..10){
  print OUT $i;
  for $j(1..10){
    print OUT "\t", $i * $j;
  }
  print OUT "\n";
}

close OUT;
```

4.5.6 Regular Expressions

You have already come across regular expressions in this course; they are used by a number of Unix utilities like grep. The Perl implementation of regular expressions is perhaps one of the best and most powerful ones available and a large part of the power of Perl comes through its ability to make use of regular expressions.

As mentioned previously regular expressions are used to identify matches to generalised text patterns in strings. There are a very large number of tutorials on how to use regular expressions in Perl available on the net and we will only provide a very short introduction here.

In Perl, regular expression matching makes use of the == = operator, where the left operand contains the text to searched for matches to the pattern given by the right operand. Some examples:

```
## The left operand is usually a variable, but for clarity we'll use
## plain strings.

## The regular expression is usually written as follows:
## "some string to be tested" =~ m/ a regular expression /
##
## the character immediately following the m delimits the regular expression. If you wish to
## include this character within the regular expression it will need to be escaped by placing
## a \ in front of it. For regular pattern matching you do not need to specify the
## m if you are using the forward slash as the delimiter. This is the most common way to writ
## So to check if an expression looks like the name of a Hox gene we can do:

"HoxA3" =~ /hox[a-z][0-9]+/;

## Normal characters are matched directly, characters within square brackets [] represent a c
## class (any character specified will allow a match). In the above example, the regular expr
## will fail to recognise the left operand since the regular expression is case sensitive. To
## this we can do:

"HoxA3" =~ /hox[a-z][0-9]+/i;

## we could also specify a character class at each position, but this would be ugly:
"HoxA3" =~ /[hH][oO][xX][A-z][0-9]+/;

## which reads as: h OR H followed by o OR O followed by x OR X followed by a single characte
## followed by at least one number. But that is pretty ugly.

## if you wish to use a different delimiter, like the # character you can write it like:
"HoxA3" =~ m#hox[a-z][0-9]+#i

## this can be useful when trying to match directory names that contain lots of forward slash

## The above expressions on their own do nothing as we do not make use of the returned value
## To actually use a regular expression we make use of conditionals, eg...

if("HoxA3" =~ /hox[a-z][0-9]+/i){
  ## we have Hox gene, do something here..
}
## to substitute words we can use the s modifier. We may wish to substitute spaces within a
## a string with underscores.
$string = "Goodbye cruel World";
$string =~ s/ /_/g;

## here we also make use of the g (global) modifier to replace all instances rather than just
## match.
```

Regular expressions make use of a number of special characters and modifiers to represent textual patterns. The characters represent character classes, followed by a modifier specifying how many matches should be present to give a match. In Perl, the most widely used special characters are:

- . The dot. This matches any character.
- \d A numeric character. Equivalent to specifying [0-9].
- \s A space.
- \S Non space characters.
- \w Word characters (alpha numeric and some others).

- `\b` Word boundaries (tabs, spaces, newlines, punctuation).
- `\t` Tab characters.

A character may be followed by a modifier specifying how many times the character should be present in the text.

- `+` 1 or more.
- `*` 0 or more.
- `?` 0 or 1.
- `{N}` Exactly N times.
- `{n..N}` n to N times.

Other modifiers can be used to specify where a match should be present: `^` and `$` specify the beginning and end of lines respectively. Note that `^` inside a character class indicates an inverted character class (matches characters not present in the class).

Regular expressions can also be used to capture specific subsections of text. A very common example would be to extract a sequence identifier from a fasta file. This can easily be done in Perl.

```
## $line contains a line from a file. Identifiers begin with the > character.
if( $line =~ /^>(\S+)/ ){
    $seqId = $1;
}
## if brackets are used in the regular expression, the values matching within the brackets
## will be assigned to variables $1 - $9. (Ordered from left to right). If you wish to match
## brackets you will need to escape them with backslashes.
```

There's a lot more to regular expressions than this, but this may be enough to get started with.

4.5.7 Various operators

Operators are symbols that denote specific operations; like regular expression matching or regular mathematical operations. We have already come across a few of these, but there are more (and the following list is not complete).

- `+` The addition operator. Returns the sum of the left and right operand.
- `-` The subtraction operator.

- ++ The auto-increment operator. Increases the value of its single operand by 1. There are in fact two different increment operators; post-increment `$v++` and pre-increment `++$v`. The former increments the value after other operations, the latter before. Consider the difference between `$i=5; print $i++;` and `$i=5; print ++$i;`.

- -- The auto-decrement operator. Opposite of auto-increment.

- += The increment operator. Increases the value of its left operand by the value of its right operand.

- -= The decrement operator. Opposite of the increment operator.

- * Multiplication.

- / Division.

- *= Sets the value of its left operand to the product of the left and right operands. Identical to `$left = $left * $right`.

- /= As above but for division.

- ** Exponentiation. Returns the value of the left operand to the power of the right operand.

- . String concatenation. Concatenates left and right operands.

- .= Concatenates right operand to left operand.

- ==== Numerical equality operator. Returns TRUE if the value of the left and right operands are equal. Causes an error if either operand is not numerical.

- != Numerical inequality operator. Returns TRUE if the value of the left and right operands are not equal. Causes an error if either operand is not numerical.

- eq String equality operator. Returns TRUE if the strings specified by the left and the right hand operators are the same.

- ne String inequality operator. Returns TRUE if the strings specified by left and right hand operators are not the same.

- > Numerical greater than. Returns true if left operator is larger than the right operator.

- < Numerical less than. Opposite of above.

- >= Numerical greater than or equal to.

This is an incomplete list, but is sufficient to do rather a lot with. Note that some operators should be used with numerical values and others with strings (pieces of text). Using the wrong data types will sometimes raise errors, but can also result in the program silently doing something unexpected (which is the worst kind of behaviour as it can result in corrupt output).

4.5.8 A somewhat useful example

As an example of something potentially useful we can write a short script that reads in sequences from a fasta file and identifies sequences that contain a specific pattern within the first N bases. To do this we'll make use of most of the techniques outlined above, but we'll also need to be able to work out options specified by the user on the command line. The arguments specified to a Perl script are assigned to a special array called `@ARGV`, and we'll make use of this array to work out what the user wants to do.

The following segment contains a full script that you should be able to run, using the ./scriptname invocation.

```
#!/usr/bin/perl -w

## the first line is not really a comment, but is used to make the shell invoke the perl inte
## script.

## first check the command line arguments to make sure that the user has specified three argu
## the first argument should give the name of the fasta file containing the sequences to be se
## the second argument the pattern to look for, and the third argument the maximum distance fr
## beginning of the sequence.

if(@ARGV != 3)\{
  die "usage: script_name fasta_file pattern max_distance_from_edge \n";
}

## we could also use regular expressions to check if the arguments are of the correct type

$seqId = "";

## open the fasta file and read line by line.
open(IN, $ARGV[0]) || die "unable to open $ARGV[0] $!\n";
while(<IN>){
  chomp; ## this removes the end of line character from $_
  ## does the line look like it contains a sequence identifier?
  if( $_ =~ /^>(\S+)/ ){
    $seqId = $1;
    next;  ## go to the next iteration of the loop
  }
  ## if we have defined a sequence identifer, we will just assume that the rest of the text co
  if(length($seqId)){
    $seq{$seqId} .= $_;   ## extends the length of, or initialises a hash entry
  }
}

## We should now have read all of the sequences into an associative array where the keys are t
## identifiers. We now go through the sequences and check for the pattern.
## The identifiers of sequences which match are printed out to STDOUT.
## We could also print the matching sequences if we wished.
```

```
for $seqId(keys %seq){
  if( $seq{$seqId} =~ /^.{0,$ARGV[2]}$ARGV[1]/ ){
    print "$seqID\n";
  }
}

## end of the script!
```

This script probably has a few bugs in it. Working out where those bugs are is a pretty good exercise for honing your Perl skills. Note also that bad command line arguments can cause all sorts of problems as the script does not check the arguments given. The script is quite useful though, as you can use it as a sort of configurable grep to learn more about regular expressions in Perl.

Be aware that this is not a very memory efficient way of solving the problem as all of the sequences are read into memory before any processing is done. This is not only memory intensive, but it's also slower. It's been written this way to show the use of hashes and to keep it reasonably short. I've also avoided using custom functions as I've not included anything about how to write your own functions (subroutines in Perl). How to write your own functions is probably the first thing you should look at after this introduction if you wish to start using Perl seriously.

Good luck with Perl!

5 Bonus section on PERL

Perl is a useful programming language whose principles can be learnt within a short period of time allowing researchers not familiar with programming to quickly become able to automate a variety of processes. Although not an official acronym, Perl is often referred to as standing for, 'Practical Extraction and Reporting Language'; and this is pretty much what Perl makes easy.

Perl has been used extensively within the field of Bioinformatics (see Bioperl, http://www.bioperl.org) though recently it has been overshadowed to some extent by the use of R for statistical analyses of data. However, Perl remains widely used and several of the tools you will use in this course have been written in Perl. R is incredibly useful when you have regular data structures that can be expressed as arrays or matrices; however it is unsuitable for describing irregular types of data (eg. structures of genes, etc.) where it may be necessary to iterate through the elements of a data set. Compared to R, Perl is a much more general programming language that can be applied to a much wider set of problems.

The motto of Perl is, 'There is more than one way to do it'. And in Perl this is very true; the same logic can be expressed in a number of different ways and masters of Perl will sometimes delight in their ability to fit a very large amount of functionality into a small amount of code. This is kind of neat, but can lead to code that is difficult to understand and should not be encouraged for code that will actually be used. The flexibility of Perl also means that it can be difficult to read other people's code as they may use a very different style of coding to ones own. Perl can also be quite a dangerous language and it is often said that it gives the user more than enough rope to hang themselves with.

6 Recommended books

- Unix and Perl to the Rescue
- Computational Biology

7 Unix cheat sheet

7.1 FILE system

Command	Meaning
`cd DIR`	change directory to DIR
`cd ..`	go up one directory
`cd ˜`	to to your home directory
`pwd`	show present working directory
`ls`	list items in current directory
`ls -a`	list all items, including hidden ones
`ls -lhcrt`	list all items in long, human-readable format and sort in reverse order by modification time
`ls -F`	list all items in current directory and show directories with a slash and executables with a sta
`tree -C`	print hierarchical structure of your FILEs and directories (color-coded)
`tree -d`	print hierarchical structure of all subdirectories
`tree -sh`	print hierarchical structure of FILEs and directories with sizes (-s) in a human-readable forma
`mkdir directoryname`	make new directory named directoryname
`mv FILE1 FILE2`	rename FILE1 to FILE2
`mv FILE1 ../FILE2`	move FILE1 one directory up
`cp FILE1 FILE2`	copy FILE1 and save it as FILE2
`rm FILE`	remove FILE
`rm -r DIRECTORY`	remove directory and all of its contents

7.2 Opening FILEs and extracting information

Command	Meaning
`less FILE`	open FILE and scroll through it line by line
`wc -l -w -m FILE`	counting lines, words, and characters in FILE
`grep "pattern" FILE`	print lines from FILE that contain “pattern”
`grp -v "pattern" FILE`	print lines from FILE that do not contain “pattern”
`cat FILE > FILE2`	write the content of FILE to FILE2
`cat FILE » FILE2`	append the content of FILE to FILE2
`sed -n 11,12p FILE`	extract lines 11 to 12 from FILE
`awk -F "\t" ’$1 > 20 {print $0}’ FILE`	Print all columns of a line ($0) in FILE if the value in column 1 ($1) is
`unzip FILE.zip`	unzip the zip-compressed FILE
`gunzip FILE.gz`	unzip the gz-compressed FILE
`sort -n NUMBERS`	sort a row of NUMBERS numerically
`uniq -c FILE`	count unique lines in FILE
`nano FILE`	open FILE on the command-line
`xdg-open FILE`	open FILE with the standard program for its file type
`eog FILE`	open FILE (which is a figure) with the Eye of Gnome graphics viewer p

7.3 Data transfer

Command	Meaning
`rsync --progress -avz SRC DEST`	transfer from SRC to DEST, show the progress while FILEs are compressed d
`rsync FILE user@host://home/usr/`	transfer FILE to the folder /home/usr on the remote server user@host
`rsync -avz directory/ DEST`	transfer all FILEs saved in directory to DEST
`rsync -avz directory DEST`	create the folder directory in DEST and transfer all FILEs in this directory
`scp -r SRC DEST`	transfer all FILEs in SRC to DEST
`scp FILE DEST`	transfer FILE to DEST

7.4 Executing scripts and programs

Command	Meaning
`nohup ...  &`	execute ... in the background
`nohup ...  > FILE.txt &`	execute ... in the background and redirect output to FILE.txt
`ps -p ID`	print the status of a process with the specified process-ID
`kill ID`	stop the process witht the specified process-ID
`pkill NAME`	stop all processes with NAME (NAME could be for example 'R' or 'python')
`top`	provides an ongoing look at processor activity in real time

7.5 Networking

Command	Meaning
`ssh user@host`	connect to host as user
`ssh -X user@host`	connect to host as user with X11 forwarding enabled (you can open programs with graphical user

7.6 Help

Command	Meaning
`command --help`	Lists the options for command
`man command`	opens the manual page for command (exit with 'q')

7.7 Tricks

Pipe output from one command with `|` as input to another command.

Command	Meaning
`TAB key`	auto-completion of commands, FILE names etc.
`UP or DOWN arrows`	move through the history of your commands
`history`	Get overview of the commands you have used
`*`	Allows to generalize file names. For example, *fasta refers to all fasta files in a directory