---help

DENNIS VAN DOK

Nikhef Computing Course, Wednesday 2022-06-22

UNIX FOR PHYSICISTS

Speaker notes

The purpose of this talk is to give a few helpful pointers to aspects of the general Linux computing environment that may be beneficial to know for a student of physics, i.e. not someone who purposely explores the realm of computing but rather sees this as (at best) a useful tool or (at worst) a necessary hurdle to overcome.

In contrast with the other talks given at the computing course, which focus on various aspects of how computing is best approached at Nikhef, this talk is more generally applicable outside our lab. Nevertheless, I've tried to aim for the target audience of prospective PhD students.

THE PHILOSOPHY OF UNIX

What? Unix has a philosophy?

Speaker notes

The single most important message I have for you is in the title and is about getting help. The Unix environment has never been known to be particularly user friendly and it can be daunting to get started. The general philosophy seems always to have been that with a basic set of skills you should be able to find documentation and apply knowledge on your own, and learn from your peers as you gain experience.

These days there is Stack Overflow, but the hardest part about that is knowing which question to ask. The question 'how do i do *X*' is easy to answer, but knowing that *X* can be done at all, or that *X* is a useful step along the way to achieving your overall goal, is not always obvious.

The student is expected to know the tools of the trade, and understand what power they wield. How to apply that power in their own work is then a matter of developing one's skills.

DEVELOPING SKILLS

Speaker notes

The background against we should consider this presentation is this poignant question: should you really learn any of this at all? As this lecture is not going in depth in any of the topics it touches on, it is up to you to investigate further, exploring and trying out to see if these are techniques you should adopt; as time is a limited resource and there is a lot of work to be done, you need to be able to make a fair judgement whether the investment is worth the cost.

SHOULD YOU LEARN A NEW SKILL?

T	time normally spent on related tasks
I	investment
R	rate of productivity increase

Speaker notes

This is the only piece of mathematics I'll discuss here and it is a piece of pseudo-science, but this formula does reflect the key point to learning a new skill. If this skill applies to your daily work and makes you, say, twice as fast at accomplishing a task, it is easy to see that the investment is sound if it cost less then half of the time normally spent.

It is also immediately obvious from the formula that if you spent very little time using a skill, no matter how high R gets, you will never recoup your investment.

The big problem with this knowledge is that the three quantities are not obvious to estimate let alone know exactly.

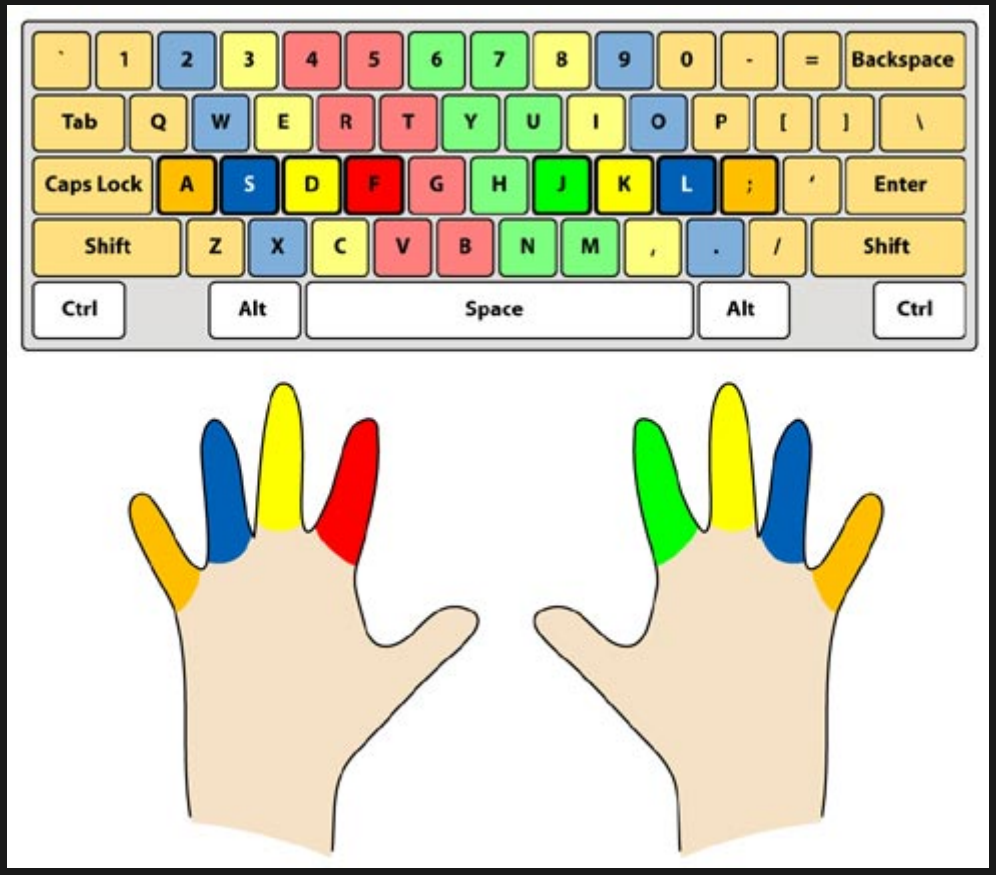
SHOULD YOU LEARN A NEW SKILL?

T	time normally spent on related tasks
I	investment
R	rate of productivity increase

Learning a skill is worthwhile if

$$T \geq I + \frac{T}{R}$$

SHOULD YOU LEARN TOUCH TYPING?

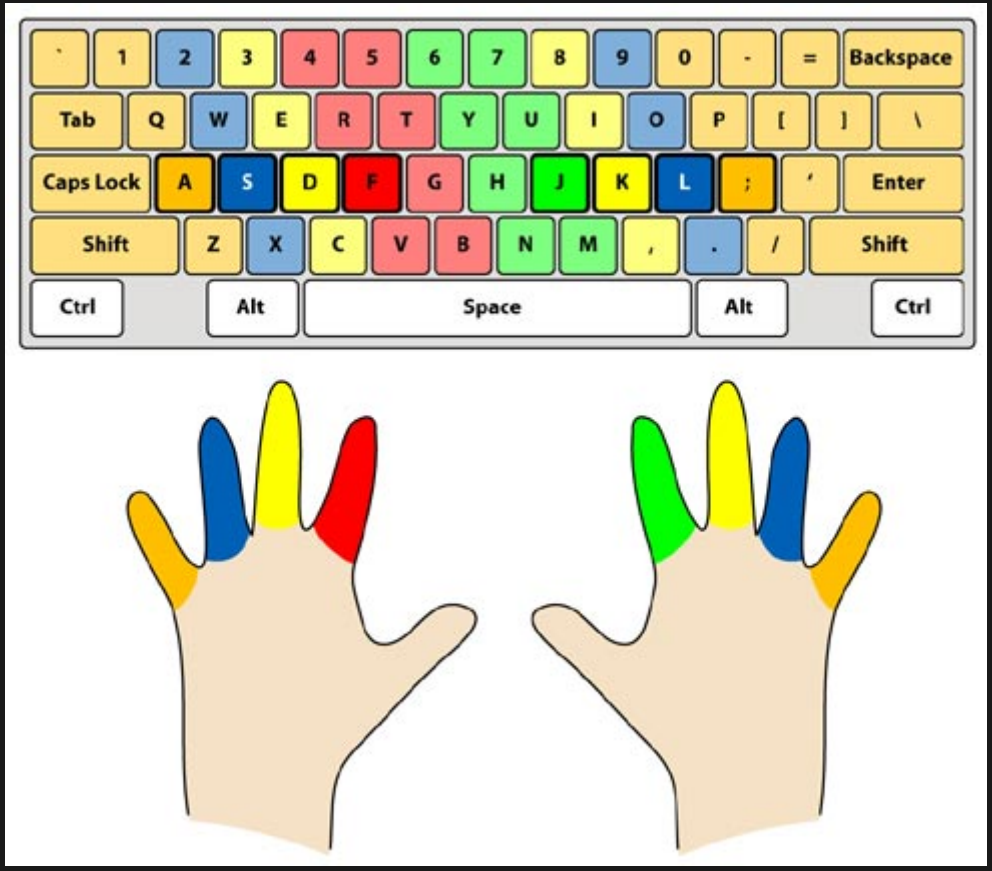


Speaker notes

As a simple example, the skill of being able to type with all fingers on a keyboard without looking at the keys and getting a reasonable 50 words per minute is something that is lacking from today's school curriculum, at least in the Netherlands. (We have so-called iPad schools which are as bad as they sound.) People are left to make this choice for themselves and many people don't.

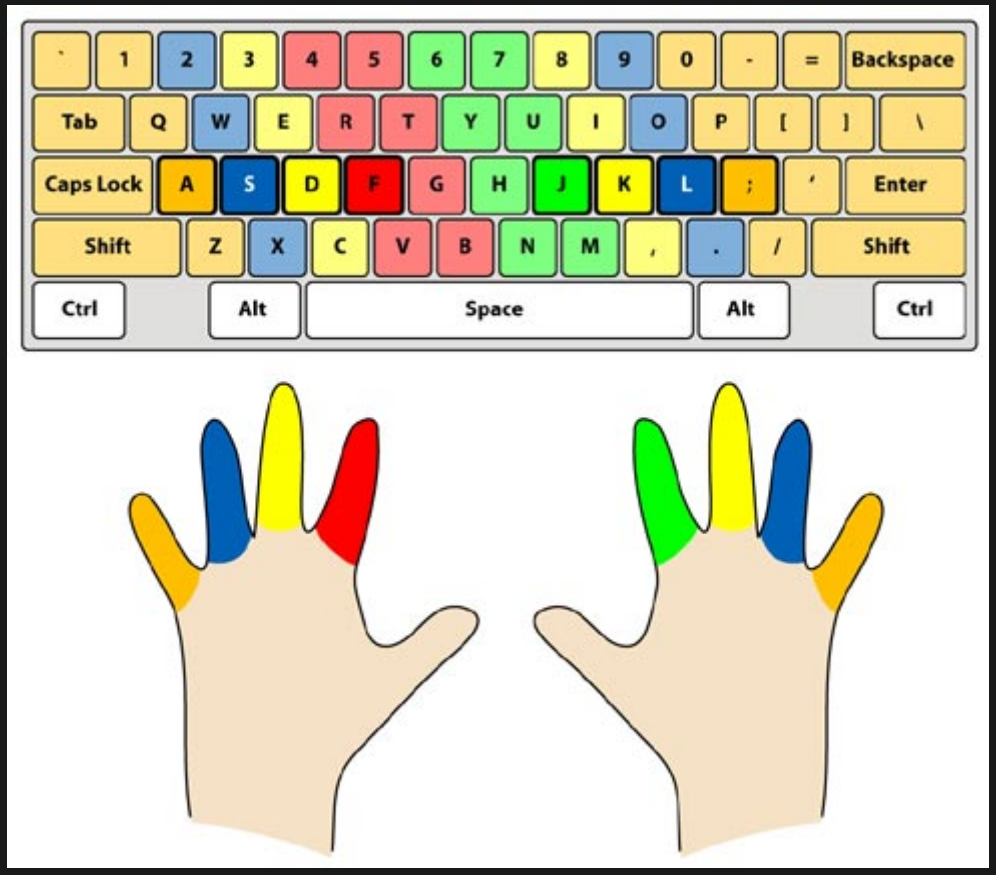
That is a shame because in most modern professions it is likely that you will be entering a lot of text via the computer keyboard and with some practice (10 hours according to one testimonial, the equivalent of half an hour per day for two weeks, not counting weekends) you can reach excellent performance and this will undoubtedly pay off.

SHOULD YOU LEARN TOUCH TYPING?



- $T \approx 1000\text{h}$
- $I \approx 10\text{h}$
- $R \approx 2$

SHOULD YOU LEARN TOUCH TYPING?



- $T \approx 1000\text{h}$
- $I \approx 10\text{h}$
- $R \approx 2$

(Yes, absolutely)

SO HOW WILL I KNOW WHAT TO LEARN?

T, I, and R can only be learned from experience.

Speaker notes

It is hard to judge these quantities for most skills, as life in general is complicated.

On the slides ahead I have compiled a collection of techniques and hidden gems of the Linux systems that I believe can improve your productivity as you are working on the Nikhef computing facilities.

But keep in mind that the theme of the talk is that you should use your own judgement in deciding whether you are going to try this.

There is the law of diminishing returns at work here. Although some of you will find joy in exploring the full width and breadth of the toolstack and becoming an absolute wiz on the UNIX command line, a little knowledge already goes a long way. In that sense you could consider this talk a a showcase of so much of what is out there to make your life easier, and you can come back later to explore at leasure to see what you like.

UNIX

Getting Linux on your laptop:

- http://get.debian.net/
- https://www.ubuntu.com/download
- https://getfedora.org/
- https://software.opensuse.org/

Speaker notes

Since you are going to see so much Unix on Nikhef systems, it makes no sense to run anything else on your laptop. Luckily this is easy enough to install in the last decades.

I'm listing four of the more popular flavours in no particular order, and I can't really say that any of them (cough Debian cough) is better than the others.

Not listed above are the many distributions which are for advanced users and/or Linux freaks, because if you feel these are your cup of tea then this lecture is probably not for you.

APPLE HARDWARE

- OS X = Unix
- VirtualBox/VMWare
- hard-core install Linux anyway

Speaker notes

Aficionados of overpriced hardware need not worry; although Apple tries to make it harder to install Linux on your devices, at the core of OS X is a POSIX compliant Darwin kernel. Some of the commands behave slightly differently because it has a BSD pedigree and Unix was never a single unified OS. Virtual machines are another option.

The modern M1 chip found in newer Apple hardware only recently attained Linux support, which makes the native option really hard-core geeky.

MICROSOFT WINDOWS

Often the best choice when there is but one choice. In that case:

- Dual Boot
- VMWare/VirtualBox
- CygWin
- WSL 2

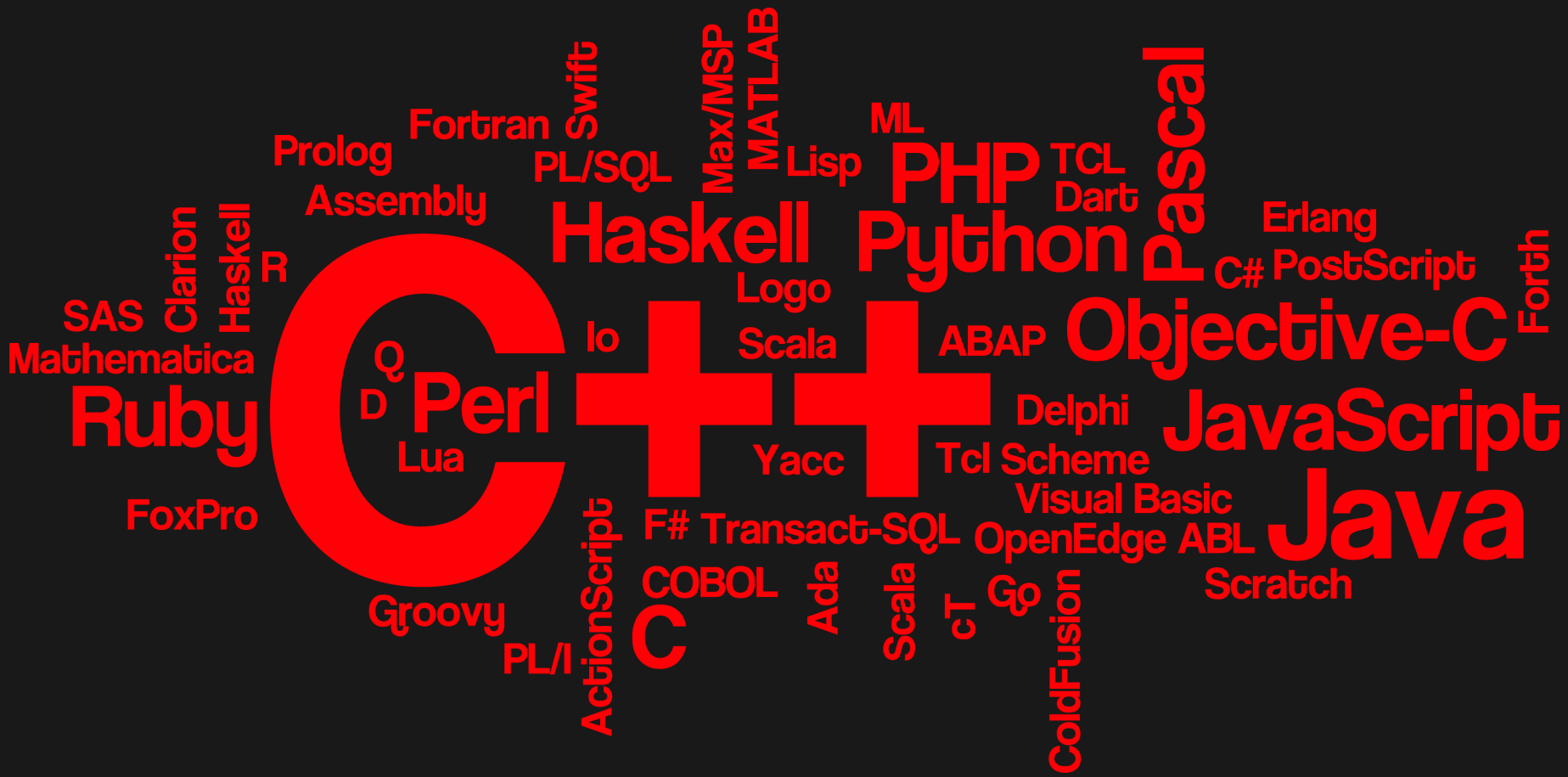
Speaker notes

If you are stuck on Microsoft's OS for whatever reason you need not despair. Any modern Linux installer will be able to turn your system into a dual-boot machine. For the less adventurous there is still the virtual machine option and most of the common tools are available anyway through CygWin (this is not Linux or even Unix, but since much of Windows is POSIX-compliant it makes a good effort to behave the same).

The modern option, however, is to go for the Windows Subsystem for Linux, which allows you to run Linux natively under Windows. This is Microsoft's own product and guarantees to integrate well.

Nikhef

PROGRAMMING LANGUAGES



Speaker notes

In the course of your time here you will not only be expected to operate and run computer programs, but also to write your own programs and scripts for doing analysis. You may already have heard of a number of programming languages and you may be wondering which of these you should learn well, which you can get away with learning only superficially and which ones you can avoid completely.

SCRIPTING LANGUAGES

- No compilation required
- Easy prototyping
- Can be used interactively
- Ideal to build workflows

Examples:

- Bash
- Python
- Perl

Speaker notes

Let's start by cutting the field into two large portions. The first is made up of the languages that run via an interpreter, i.e. a special program that reads the program statement by statement, translates these into actions and executes these actions until the script ends, usually by reaching the end of the file, or an explicit call to end, or an error condition from which it cannot recover.

The Bash interactive shell doubles as a makeshift programming language but it is probably the ugliest and least efficient one out there. Since nearly every 'call' to a routing is really forking another pipeline the overhead is enormous. Still, it is a useful control language for batch jobs, preparing input files, etc. and it prototypes so naturally because everything can also be done on the command line.

Perl never really went away, but it is now eclipsed by the massive uptake of Python as a scripting language.

COMPILED LANGUAGES

- Translate down to the CPU instruction level
- High performance
- Various degrees of abstraction away from the underlying architecture

Examples:

- C/C++
- Fortran
- Go
- Rust

Speaker notes

The other portion are the languages that take an intermediate step: a compiler translates the statements down to snippets of machine code particular to the architecture of the CPU that the program is supposed to run on; it optimizes constructs to make use of certain features of the CPU. These programs are much faster for processing and computational tasks.

You probably won't end up choosing your own here, as much of the work that has come before has already been done in one way or the other. The newcomer with the most potential at this moment looks to be the Rust language.

All of the heavy lifting and advanced compiler tech happens in C++, which is more of a conclomerate of language features than an actual design.

MOST LIKELY COMBO

Python/C++

(Special recommendation: Jupyterlab)

Speaker notes

There is a place for both types so you have to learn at least two. For compiled languages it is hard to escape C++. For scripted languages it is hard to beat the double-barrelled power of having an interactive shell that is also a programming language, more about Bash later.

The shortcomings of Bash as a data processor are obvious, and here Python jumps in the gap. With the power of notebooks (a.k.a. Jupyter) this could be an excellent tool for workflow-building and note keeping.

A NOTE ON C++

There is a decade of architectural development between current CPUs (AMD EPYC 7H12) and what we still had last year (Intel Xeon E5-2650). The clock speed, however, is still in the same ballpark.

Principally, your C++ program will compile to both. Technically, to make use of all the advancements in processor design it takes a lot of insider knowledge of both the CPU and compiler optimization.

Speaker notes

This is beyond the scope of this talk but just wanted to mention it: newer compilers tend to do a lot better on modern hardware. Typically somebody in the project will have set this up so you just need to source a single shell script to have access to the latest and greatest.

SSH

- secure remote shell
- Passwordless
- versatile

Speaker notes

The standard way to get around on systems is by way of a secure remote shell, a.k.a. ssh.

SSH will set up an encrypted communication channel between your computer and the remote server; even if the traffic across this channel is intercepted, the actual data will be indecipherable for the interceptor.

What most people don't realise right away is how versatile ssh really is.

SETTINGS

.ssh/config

```
Host *.nikhef.nl
    ControlMaster auto
    ControlPath /tmp/%h-%p-%r.shared
Host *
    ForwardAgent yes
    User yournamehere
    HashKnownHosts yes
```

Speaker notes

Connection sharing means that only the first connection needs to authenticate and subsequent ssh actions to the same host will happen much quicker. I'd always recommend this with the caveat that if the master session dies, so do all the clients.

Hashing of the known hosts file is generally a good idea, but it is not a productivity gain; it is a security measure to prevent others from seeing which machines you regularly log on to.

Read the man page of `ssh-keygen` to see how to find and remove old entries.

Agent forwarding is discussed below.

These settings go to your laptop and your Nikhef home directory.

SSH PUBLIC/PRIVATE KEY

```
ssh-keygen
cat ${HOME}/.ssh/id_rsa.pub > authorized_keys
scp authorized_keys login:.ssh/authorized_keys
```

Permissions:

```
drwxr-xr-x .ssh/
-rw-r--r-- .ssh/authorized_keys
-r--r--r-- .ssh/id_rsa.pub
-r-------- .ssh/id_rsa
```

Speaker notes

Instead of having to type your password every time it is possible to set up a public/private key pair. The private part stays with you and you alone; there is a password on it for good measure.

The public part you can spread everywhere.

Keep the private key on your laptop. Don't copy it anywhere else!

If you also work at Nikhef desktop systems, make another ssh key there and add it to the authorized keys file as well.

Nikhef

AGENT FORWARDING

```
ssh-add -l       # list keys in the agent
ssh -A login     # login with agent forwarding
```

Speaker notes

Logging in through a chain of servers is easier with an ssh agent. Normally an agent is already started for you.

The forwarding means that the agent can be reached through a backchannel.

This saves so much typing of passwords that this should almost be considered mandatory.

PROXY FROM OUTSIDE NIKHEF

```
Host stbci2.proxy
    Hostname stbc-i2.nikhef.nl
    user yournamehere
    CheckHostIP no
    ProxyCommand ssh -q -A login.nikhef.nl /usr/bin/nc %h %p 2
```

Speaker notes

This little trick so useful that recent implementations of ssh have now incorporated this functionality so you could try the `ProxyJump` option instead. See the man page for `ssh_config`.

In combination with Agent forwarding this means you get to log on to Stoomboot from anywhere in the world without typing your password once.

SSHFS

Fuse mount your remote home directory locally:

```
sshfs login.nikhef.nl: /tmp/login
ll /tmp/login/
fusermount -u /tmp/login
```

Speaker notes

Copying files by SSH can be done with `scp`, but there is a really convenient way under Linux using the FUSE file system driver.

The sshfs command mounts a remote server directory based on your ssh authentication. It appears just like an ordinary directory.

Be mindful of the abstraction and realise this is all going over a single TCP connection before you run a recursive directory listing or a `find` command.

COMMAND LINE SHELL

- tell the computer what to do, one line at a time
- most powerful way of direct interaction
- also used for scripting and fast prototyping
- ideal for taking notes as you go

Speaker notes

Although the graphical desktop environments on Linux systems have evolved and matured over time to be among the best in the industry, any serious work that involves scientific programming and analysis will require working with a command-line shell.

WHICH SHELL DO I NEED?

select your default shell at https://sso.nikhef.nl/chsh.

`/bin/bash`	**YES**
`/bin/zsh`	**YES**
`/bin/csh`	**NO!**

Speaker notes

If you believe there is any semblance between the C programming language and the C shell, you are regrettably misled.

Nikhef offers two perfectly reasonable shells but for the remainder of this presentation we are going with `bash`. I cannot vouch for productivity gains using `zsh` as I am not using it myself, but there are plenty of people who do and you may ask them.

TUNING

- everything can be tuned
- but you must resist
- use only the common enhancement

Speaker notes

Everybody wants to optimise their environment according to personal preference. The Unix attitude takes this into consideration and allows a million settings to suit everybody’s whim. The collective wisdom accumulated over four decades have delivered some tricks that everybody will want to know.

STARTUP FILES

login shell	`.bash_profile`
non-login shell	`.bashrc`

This distinction is outmoded.

Nikhef

.bash_profile

```
if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"
fi
```

Speaker notes

rc files are 'runcom' files, a holdover from a more beautiful era.

When a shell is started, a couple of files are sourced both system-wide and personal, and that is where to put your preferences.

Put all of your favourite settings in `.bashrc` and leave everything else to the system-wide `/etc/profile`.

PATH

`.bashrc`

```
if [ -d "$HOME/bin" ] ; then
   PATH="$HOME/bin:$PATH"
fi
```

Speaker notes

The `PATH` variable is a list of directories which are searched when you type a command.

Stick your own programs and scripts in `${HOME}/bin`.

Do not put `.` in your `PATH` and certainly not at the beginning! This poses a security risk because you will not be sure that you are not running a program from a local directory that you did not intend to run. It is better to adopt the notation of `./program` for local programs.

COMPLETIONS

- pressing `TAB` will auto-complete your command line
- works better with the package `bash-completions` installed

Speaker notes

I mentioned touch typing before and how much of a difference that will make. But *command-line completion* will have much more of an impact because it will save you from having to make all those keystrokes in the first place.

This functionality comes standard with your shell. Any partly written command or argument followed by a tap on the `TAB` key will either:

- complete the command if it is unique, or
- list the possible completions if there are multiple matches

Install the bash-completion package to unleash the full power.

Most programs (like `git`) come with their own collection of completion-fu so as to make the command line git-aware.

HISTORY

.bashrc

```
# don't keep more than one copy of a repeated command
HISTCONTROL=ignoredups
# append to the history file, don't overwrite it
shopt -s histappend
# keep plenty of history
HISTSIZE=65000
# useful on systems with shared home directories
HISTFILE=${HOME}/.bash_history-$(hostname)
# keep track of time
HISTTIMEFORMAT='%F %T %Z # '
```

Speaker notes

Let bygones be bygones but learn from your mistakes. Keeping a record of commands you ran earlier is quite useful. By default bash will keep track of this but there are a few useful enhancements.

For instance, the size of how much of history to keep can be tuned and modern systems can easily deal with thousands of lines.

On a shared home directory system such as Nikhef's it is sensible to have a history file per host, because you run different commands on on them.

Bash does not record the time at which a command was run by default, but this is also useful information to keep.

HISTORY RECALL

- Arrow up/down cycles through previous commands.
- `Ctrl-R` reverse search in history

Speaker notes

This is one of those little gems that you either stumble upon or go for years without until somebody points it out to you.

I've seen people furiously rapping at the `up` key to find an earlier command and it is so sad.

- Type `Ctrl-R`
- type a few letters from the command; this will start a reverse search through the history
- type `Ctrl-R` again to cycle back through matches
- or type more characters to refine the search term
- press enter to rerun the found command
- or press arrow keys to edit the command line

Nikhef

RECALL THE LAST ARGUMENT

Seeing is believing.

```
stat /some/path/to/file
# now I want to run cat on the same file
cat <ESC><.>
cat /some/path/to/file
```

Speaker notes

This is something I use surprisingly often. E.g.

```
stat /some/path/to/file
cat /some/path/to/file
```

Instead of recalling the history for the second line, simply type `cat Meta-.` to stick the last argument of the previous line on the end of the current line. The meta-key may be `Alt`, or the Windows key, or just us `ESC`. Repeat the command to cycle back through earlier commands.

These are ingrained in muscle memory over time.

PROMPT

`.bashrc`

```
PS1='\u@\h:\w \A $(__git_ps1 " (%s)")\$ '
```

This shows:

```
a07@lena:/project/newton 11:24 (master)$
```

Speaker notes

The prompt is displayed to indicate that the shell awaits your next order. Did you know you can enhance the prompt, e.g. to indicate the time, host name, and current path? Or even the current git branch name?

Letting the computer tell you something useful here is pretty sensible, but I've seen people go overboard on what they display in their prompts. That is entirely up to you of course, but keep in mind the increase in productivity is probably no too much.

Nikhef

ALIASES

```
alias ls='ls --color=tty'
alias ll='ls -lhF'
alias rm='rm -i'
alias mv='mv -i'
```

Speaker notes

It’s safer to protect potentially dangerous commands with a mandatory interactive flag.

More fanciful shortcuts can easily be implemented with shell scripts.

Often used commands can be abbreviated by creating aliases. My advice: don’t overdo it on the aliases. Stick to some of the more usual ones. any alias can be overridden by putting a backslash in front.

KEEPING NOTES

- use `script` to capture an entire session
- run a jupyter notebook with a bash kernel
- emacs org-mode babel extension

Speaker notes

Interactive sessions help you work through certain problems in rapid short cycles. But it can be frustrating after a succesful bout of trial-and-error to retrace your steps.

One fix could be the use of `script`. Start it at the beginning of your session, and everything you type will be recorded in a file to peruse later.

A modern technique that is gaining popularity is the python notebook a.k.a. jupyter notebook. This can also be run with a bash kernel.

Finally a more fringe option is the Emacs org-mode capability of inserting and running code blocks inside documentation and capturing the results.

SCRIPTING

Write `myscript.sh`:

```
# my first script
echo "This is my first shellscript"
```

And then run it like

```
bash ./myscript.sh
```

Turn it into an executable like so:

```
#!/bin/bash
# my first script
echo "This is my first shellscript"
```

followed by

```
chmod +x myscript.sh
./myscript.sh
```

Speaker notes

The power of bash as a command-line tool is complemented by its power as a programming language. Without learning any more commands you can start writing a shell script by writing the commands to a file.

ESCAPING

Make a habit out of always quoting variables like so:

```
"${var}"
```

and you will never go wrong.

EVAL IS EVIL

Do **not** use `eval` ever.

Speaker notes

By the time you think you need eval, you need to switch to a real programming language.

PARSING COMMAND-LINE OPTIONS

```
#!/bin/sh
proxyhost=login.nikhef.nl
proxyport=8888
while getopts :h:p: OPT; do
    case $OPT in
    h|+h) proxyhost="$OPTARG" ;;
    p|+p) proxyport="$OPTARG" ;;
    *) echo "usage: `basename $0`"\
            "[+-h proxyhost] [+-p proxyport} [--] ARGS..."
       exit 2 ;;
    esac
done
shift `expr $OPTIND - 1`
OPTIND=1
ssh -n -N -f -D "$proxyport" "$proxyhost" "$@"
```

Speaker notes

The getopt utility can be used as well and supports long options.

DANGERS OF QUOTES

Jeff thoroughly tested the following code. Then he changed one line. What went wrong?

```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```

```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```


```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```

Speaker notes

This real-world example (slightly adapted for brevity) shows how tricky the shell can be when it comes to quotes. I've displayed the text with and without syntax highlighting, so you see how important that is.

The quote in the echo statement cancelled the quoted string, so the last quote actually started another one not cancelled until the ‘it’s’ in the comment further on.

This skips the setting of `$path` and that means that the cleanup script will run from ~/~… Oops.

You may not lose an entire Saturday over a single quote but it may be a good idea to read up on the subtleties between single and double quotes.

DEBUGGING SHELL SCRIPTS

You will find yourself at times pondering why your shell script went south. Here is what you do next.

Nikhef

DON’T IGNORE ERRORS

```
echo $?
```

Speaker notes

Decent programs (and your shell scripts, right?) exit with a zero (0) exit value when everything went OK, and non-zero otherwise. The special variable `$?` shows the exit value of the last command that was issued and it is prudent to inspect this value before carrying on.

FAIL EARLY AND GRACEFULLY

```
set -e
trap 'fail $LINENO' ERR
fail() {
    echo "error on line $1" >&2
}
```

Speaker notes

The default behaviour of a shell script is to carry on in the event of failures. It will only bomb out if it encounters a serious syntax error in the script, but no checking is done before it runs. Your script could be crawling with errors but as long as they aren't in the execution path, you’re fine.

Another approach is to be very strict about errors.

This will ensure that the execution stops when a non-zero return value is encountered and that the line number is printed.

INPUT, OUTPUT, ERRORS?

input	stdin	0
output	stdout	1
output	stderr	2

Speaker notes

Each Unix process has at least three standard data streams: one for input, and two for output

- stdin (fd 0)
- stdout (fd 1)
- stderr (fd 2)

It is useful to keep the normal output stream separate from the error stream.

REDIRECTIONS

Redirect both output streams to separate files.

```
run=`date -u +%FT%T`
./analysis.sh > "output.$run" 2> "err.$run"
```

Speaker notes

Common pattern to split normal output from diagnostics/warning/errors/info.

Use `>>` instead to append to a file instead of overwriting.

Nikhef

DEBUGGING STATEMENTS

```
echo "now starting the frobnicator" >&2
```

Speaker notes

Putting `echo` statements in your scripts may help with debugging. They should not be mixed with the standard output.

This means that stdout goes to the same stream as where stderr happens to go to.

TRACES

```
set -x
foo=somevalue
echo $foo
set +x
echo done
```

Renders:

```
+ foo=somevalue
+ echo somevalue
somevalue
+ set +x
done
```

Speaker notes

Use `set -x` judiciously throughout your code to print traces of all executed statements.

DEBUGGING—CHECK THE ENVIRONMENT

Dump the environment and check carefully:

- `PATH`
- `LD_LIBRARY_PATH`
- `LD_RUNPATH`
- `PYTHONPATH`
- `LANG`, `LC_*`

Speaker notes

Beware of funky influences of locale settings on the behaviour of some programs. When paranoia sets in, issue

```
export LC_ALL=C
export LANG=C
```

and try again. Also check the output of the `locale` command.

KEEPING IT IN ONE FILE

For completeness sake, here we compound stdout and stderr onto a single file.

```
./whatever.sh > all_the_output 2>&1
```

Mind the ordering. First you need to send stdout to a file, then you want to send stderr to the same stream.

COMMON UNIX TOOLS

“do one thing and do it well.”

- ––help
- man/info
- Google

Speaker notes

The Unix philosophy is “do one thing and do it well.” There are a couple of programs out there that implement the primitives for basic data manipulation.

In true self-documenting spirit, all tools have manpages. Start with `man man` and work your way up.

For everything else, there is Google.

REGULAR EXPRESSIONS

Find e-mail addresses:

```
grep -E -o "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}\
```


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Speaker notes

Regular expressions permeate much of the Unix toolchain and they can be a powerful tool. Then again, there are those who spent their entire lives not using a single regular expression and they look happy, too.

Perhaps it is better to know “that guy” than to be “that guy”.

SOME OF THE MORE COMMON TOOLS

Speaker notes

The remainder of this section is mostly there for reference.

TEXT MANIPULATION

cat	just listed here for the most useless use of cat award
sed	streamline editor with regular expression powers
awk	the duct tape of Unix tools
grep	find strings in files
sort	order lines

cut	select fields from each line
diff	show differences between files
head/tail	tail -f is actually useful
tar	roll directories into tarballs
gzip	compress files or data streams

FILE SYSTEM

`ls`	swiss army knife of file listings
`find`	most of the time you want to use locate instead
`touch`	create files out of nowhere, update timestamps
`cp`	copy
`mv`	move or rename
`ln`	link

`rm`	really remove
`rsync`	copy on steroids
`which`	where is my executable?
`stat`	what can we tell about a file
`du`	disk usage

SYSTEM PROCESSES

`ps`	list processes, like `ps aux` or `ps -ef`
`top`	who is eating my cpu and memory?
`kill`	sending signals
`bg/fg`	background/foreground programs
`lsof`	find open files
`vmstat`	memory, buffers and io
`free`	overview of memory

NETWORK

Nikhef

`ip`	swiss army knife of network tools
`ip addr`	show network addresses on this system
`ip route`	show the routing table
`ping`	see if we can reach a machine
`dig`	query DNS
`traceroute`	see which path takes us to a machine
`ssh`	secure shell

PACKAGE MANAGEMENT

`apt/dpkg`	Debian’s package manager
`yum/rpm`	Red Hat’s package manager
`pip`	Python package tool
`conda`	More general packaging

PIPELINES

Traditional Unix tools are designed to work with stream processing in mind. With ‘pipes’, the tools can be linked together like perls on a string.

Below are a few examples.

JOB MANIPULATION ON STOOMBOOT BATCH SYSTEM

Find running jobs owned by user id and delete them (you can only delete your own jobs, of course).

```
qdel `qselect -u dennisvd -s "R" `
```

FIND AND GREP

This traverses a directory and finds all files of a certain name and then tries to grep for a certain pattern in these files.

```
find . -type d \( -path \*/.svn \
    -o -path \*/.git \) -prune -o \
    -type f \( -name \*.txt \) \
    -exec grep --color -i -nH -e searchterm {} +
```

MANIPULATE A SET OF PREDICTABLY NUMBERED FILES

```
for i in `seq -f file-%03g.txt 1 100` ; do
    sort -t, -n -k2 $i | cut -d, -f2,4-8 | \
        tail -n 1 > ${i%.*}.ord
done
```

A set of 100 comma-separated data files is numerically sorted on the second field, cut to only output fields 2, 4, 5, 6, 7, and 8, and then the last lines are saved to an output file.

DISK USAGE REPORT

```
du -s * | sort -n
```

Show which file/directory uses the most disk space.

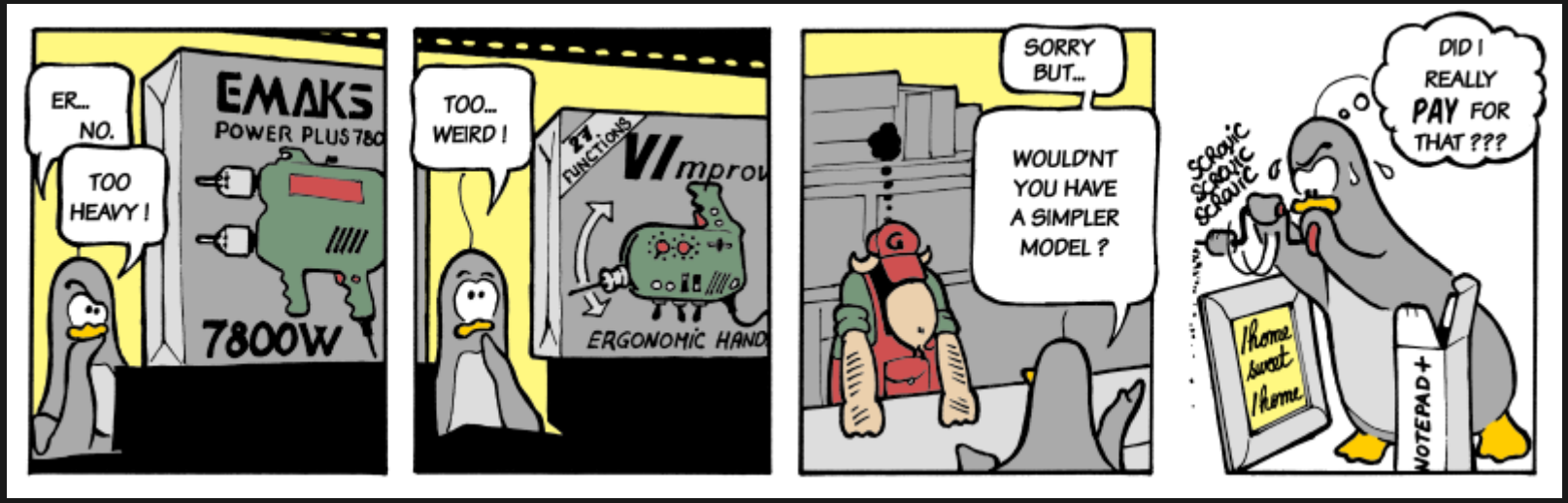
MOST RECENTLY CHANGED FILES

```
ls -lrt    # sort by timestamp
find . -mmin -10 -ls    # find files changed in the last 10 mi
```

EDITING FILES

At some point you will need to edit files: source code, LaTeX files, shell scripts, configuration files…

Modern Linux systems have plenty of editors to choose from.



Speaker notes

But smart people will stick religiously to only one of two: Emacs or VIM.

EMACS

- The thermonuclear word processor
- Everything and the kitchen sink
- Now with org-mode

Speaker notes

Emacs has a reputation for being slow and bloated, as well as overly complex. In truth, this editor has stood the test of time. There is active development and a ton of packages for every type of file and every type of workflow.

cons	pros
not generally installed everywhere	can edit files remotely
steep learning curve	built-in documentation
encourages heavy customisation	superbly extensible

EMACS

- The thermonuclear word processor
- Everything and the kitchen sink
- Now with org-mode

- $T \approx 1000$
- $I \approx \infty$
- $R \approx 100$

VIM

Originally `vi`, its pedigree going back to the original editor called `ed`.

Speaker notes

The original text editor of Unix. Nowadays it is actually “VI Improved” or VIM, which is much more powerful. The graphical version is called gvim. It can be personalised and extended.

cons	pros
editing modes require practice	powerful editing with very few keystrokes
limited extensibility	installed on nearly every system
strictly just an editor	Remote editing at lightning speed

VIM

Originally `vi`, its pedigree going back to the original editor called `ed`.

- $T \approx 1000$
- $I \approx 10$
- $R \approx 3$

SCREEN/TMUX

Sometimes you remote session should last longer than your workday. Or your laptop's battery.

The `screen` utility allocates a pseudo terminal attached to a background process independent of your session. You can run multiple shells in a screen and manoeuvre around with the Ctrl-A prefix. Type `Ctrl-A ?` for a help screen.

The tmux utility is a remake of screen, with modernised session handling, scripting, split screen, and ease of use. It is still less ubiquitous than screen so you may not have the option to run it unless you bring your own.

GIT

Version control of all your work, notes, programming, etc.

Nikhef has a public gitlab.

- $T \approx 100$
- $I \approx 10$
- $R \approx 2$

Speaker notes

Version control of your work is a game changer. Where you would think of the progress of your labour as a more-or-less linear affair, adding the ability to veer off into separate branches makes this a multi-dimensional experience. Going back to revisit earlier versions of the work makes you feel like a regular time-traveller.

All this new space opening up may cause some anxieties, a feeling of not knowing where to go or what to do. It is worthwhile to invest some time in learning the tools here, because they are especially powerful but not necessarily intuitive right away.

With git, even if you make a mess of it, there is nearly always a way to clean it up again (without throwing everything away and starting over).

WORKFLOWS

- gitflow
- OneFlow

(This may not be your choice to make.)

Speaker notes

To Rebase or not to rebase

It all comes down to which school of merging you wish to follow. One school follows the principle that merges should be proper merges as this renders a more faithful representation of the development history. The other school adheres to the idea that rebasing produces a cleaner, if somewhat synthetic, outline of the project's past.

SECURITY

Security considerations are usually not at the top of everyone's priority list. The adage: "Convenience, Speed, Security: pick two" might as well be

> *Convenience, speed, security: we know you will pick convenience and speed.*

RULE 1

Talk to the experts. At least once.

Speaker notes

It is unreasonable to expect everybody to immediately understand why things are security risks or even security sensitive. The experts actually appreciate it when you come and talk to them—they don’t get too much social interactions usually.

RULE 2—PASSWORDS

Treat passwords with extreme care.

Passwords are considered ‘something only you know’, but as soon as you write them down somewhere, on a piece of paper or in a file, you could inadvertently share this with others.

Never put passwords in a script. There is always a better way. Be aware that passwords typed on the command line will appear in your history file.

RULE 3—DATA

Where does this data go? Who has access to it? Since last year, a new EU directive went into effect governing the handling of personal information.

For Nikhef, personal data includes **user identities**.

This means that publishing the output of `qstat` on a personal web page is already a violation!

RULES 4 THROUGH ∞

- protect your security tokens (ssh private key)
- strong passwords
- different passwords everywhere
- do not log in from a public computer
- encrypt your phone
- encrypt your laptop
- encrypt your grandmother
- program with a deep mistrust of human beings

Speaker notes

Look, I could go on about all these rules. But the general gist is to be mindful about sensitive data. That could be your Facebook profile, but also your ssh private key. We hear about data breaches from large corporations nearly every week. If you use the same password everywhere, chances are it has been stolen.

It is generally a good idea to apply a principle of mistrust when programming. Processing data that comes from somebody else than yourself should be treated with utter paranoia.

And don't use eval. Ever.

TEMPORARY FILES AND DIRECTORIES

Established practice for safely creating temporary files is by using `mktemp`.

```
tmpfile=`mktemp`
tmpdir=`mktemp -d`
```

This takes care of creating a new file with a randomised name that is guaranteed to be owned by the user.

USING PASSWORDS IN SCRIPTS

Sometimes scripts need to use a password to authenticate or unlock. The script can read the password from `stdin` and keep it in a local variable for the time that it is needed.

```
stty -echo
echo "enter password:"
read passwd
stty echo
mkproxy --passin - <<<$passwd
unset passwd
```

Be aware that putting passwords on the command-line means that it will show up in the process list.

FINALLY

Learn just enough Linux to get things done

http://alexpetralia.com/posts/2017/6/26/learning-linux-bash-to-get-things-done

Learning git branching

https://learngitbranching.js.org/

Advanced Bash-Scripting Guide

http://tldp.org/LDP/abs/html/

Focus Hard. In Reasonable Bursts. One Day at a Time

https://www.calnewport.com/blog/2009/08/20/focu
hard-in-reasonable-bursts-one-day-at-a-time/

#Linux on Freenode.net IRC

https://freenode.linux.community/how-to-connect/

Gitlab server at Nikhef

https://gitlab.nikhef.nl/

Let me Google that for you

http://bfy.tw/FDe5

Emacs Org mode

http://orgmode.org/

Reveal.js

https://revealjs.com/