Scientific Computing

Scientific Computing is the use of computers to solve problems in Science, Engineering and Mathematics. It often involves numerical calculations.

Some Uses

**Solving** Systems of equations, differential equations

**Simulation** Describe/predict behaviour of a complex system based on basic physical laws for its elementary parts.
Weather Forecast

Source: Unisys
Stability prediction

(Finite elements method)

Source: F. Rieg
Airflow

Source: IAM Freiburg
Data Analysis

Transformation of experimental (real world) data into interpretable data.

Tomography
Accelerators

Source: CERN
Genome Analysis
Encephalograms

Source: U. Wisconsin

Visualization Graphical Interpretation of a large set of numerical data (In GS511)
This course

- Working with a computer
- Structure of a Computer
- Basics of Numerical Analysis and Computer Algebra
- Solving equations
- Standard techniques
- Performance Improvement
- Audience requests
Course Structure

- Lectures Monday, Wednesday
- Laboratory on Friday
- One homework sheet per week.
- End of semester project (30% grade)
Introduction to UNIX

UNIX is the de-facto standard operating system for workstations. As you will have to work on (various) UNIX machines for the homework and projects the following slides give a short introduction.

The main way to communicate with a UNIX machine is text-based (Similar to the DOS window under Windows.):

You have a shell prompt, type in a command (followed by ENTER), the shell then executes the command (respectively: starts the program, whose name is the first part of the command with given the parameters):

A52 /home/hulpke > ls -lt

```
total 60848
drwxrwxr-x 4 hulpke hulpke 4096 Aug 3 15:21 sc
drwxr-x--- 7 hulpke hulpke 4096 Aug 3 14:32 mystuff
...```

JAH 13
Unless you work on the computer itself, you have to connect to it first.

telnet  older program, unsecure, often disabled by now

ssh  Encrypted connection (no eavesdropping on password)

For file transfer, the programs are:

scp  (secured)

ftp  (Ws-FTP on the Math PCs also has an scp option)
Common Commands

man  Online manual page: man ls, man -k word

more  View a file page by page: more mynicefile  (Newer version: less

ls  Lists the contents of a directory

cd  Changes directories

mkdir  Creates a directory

rm  Deletes a file

cp  Copies a file

mv  Moves a file (or directory)

exit/logout  Ends the login
Parameters

Commands take

- Control parameters, typically starting with a − sign.
- Parameters (typically file names) to work on.

For example: ls −lR mydirectory

Often programs can work on multiple files

mv myfile subdirectory
mv myfile yourfile subdirectory
Regular expressions

Regular expressions are a way to describe a pattern that will fit (several) filenames (or – in a text-processing context – letter sequences).

They are an extremely powerful tool. The basics are:

* matches any sequence of characters

? matches a single character

\([a - z]\) matches a single letter in the range a-z

They are very useful to work on multiple files:

mv *.txt subdir

mv filenr?.dat subdir

mv file[a-z].dat subdir
IO, Redirection and Pipes

Many commands take an input and an output file, which default to stdin and stdout.

You can change this to other files using the < and > operators

```
ls -l >filelist
```

The >> operator appends the output to a file

```
cat appendix>>existingfile
```
Using *pipes* it is possible to build a chain of commands, one program processing the other programs output

```
ls -l | sort +4 -n | less
```

It can be useful to run a big program that does not produce screen output in the background. This is done by appending a `&` to the command:

```
mv manyfiles* subdir &
```
Standard UNIX Editors might behave a bit different than what you might be accustomed to. However it is necessary to learn to use at least one of them:

**vi** The oldest of the three editors listed here. Available on virtually any system, but can be cryptic. If you want to learn how to use it, there are some links on the web page.

**emacs** Big and powerful, easy to get started, but hard to use in perfection.

**pico** Probably the most beginner-friendly, but not available on every system.
Some Editor commands

On the course links web page you will find several vi tutorials.

It is also worth to know the following (emacs/vi) commands since many programs use the same:

\[ \uparrow \text{<ctrl-P>}, \text{k} \]
\[ \leftarrow \text{<ctrl-B>}, \text{h} \quad \rightarrow \text{<ctrl-F>}, \text{l} \]
\[ \downarrow \text{<ctrl-N>}, \text{j} \]
\[ \text{<ctrl-A>}, \text{^} \quad \text{respectively} \quad \text{<ctrl-E>}, \text{$} \quad \text{go to start and end of a line.} \]
Each file has permissions to read, write, execute.

These permissions can be set independently for yourself, your group (say: Graduate Students, or Programming Department) and others.

A long format directory listing shows these permissions:

```
-rw-r----- hulpke fac 3042 file1.txt
-rwxr-xr-x hulpke fac 921 myprog
```

One can change permissions using the `chmod` command. Either one gives a new permission as a 3-digit number. The digits are composed binary from the values 4 (read), 2 (write), 1 (execute). For example:

```
chmod 644 myfile
```
Alternatively one can vary permissions by letters:

```bash
chmod o+rx myfile
chmod go-w myfile
```

To save setting permissions differently for each file, the `umask` command sets a default for all new files. It gives a number that excludes a bit mask of permissions.

```bash
umask 027
```

forbids writing to group and everything to others.

Execute permission for a directory is needed to go into a directory (for example to read files there or to go into subdirectories).
The shell

The prompt you get when logging in is produced by a program, the *shell* (there are various flavours) which takes in keystrokes and executes programs.

It also processes the parameters, expands regular expressions for files, and handles pipes and redirection.

The shell also has some built-in commands and a built-in programming language (See manual).
Shell and Environment variables

The commands `set` or `setenv` give a list of variables that determine some program behaviour:

`path`  A list of all directories in which the shell will search for programs.

`MANPATH`  A list of all directories in which `man` will search for documentation.

`EDITOR`  The default editor to be used.

`DISPLAY`  The name of the (X-Windows) display on which programs are run.
Shell scripts

The shell has its own simple programming language that can be used for small programs:

```bash
foreach i (*.bak)
    echo "Deleted " $i
    rm $i
end
```

It is possible to put such programs in a file (so that calling this file executes the program. For this:

- The program is put (with line breaks) in a text file
- The first line of the text file must be the name of the shell to execute:

  ```bash
  #!/usr/local/bin/tcsh
  ```
- It is possible to use `perl`, `awk` or other programs instead of a shell
Startup

When you log in, the computer automatically executes the commands in the files `.login` and `.cshrc` or `.bashrc`.

(You can see such files by `ls -a`)

These files set many of the shell variables and probably also other default behaviour, for example `alias` commands.

(Warning: Changing settings you don't know might disable future logins!)
Useful tools

**elm/pine**  Two common email programs

**(f)grep**  Finds patterns in file(s):
   
   ```
   grep "Ompaloo" Mail/*
   ```

**diff**  Reports differences among files

**tar**  UNIX archiver (Also see gzip compressor)

**screen**  Permits to “detach” a login session and reattach it later. You can log out and continue at the same point the next day.

**make**  Resolves dependencies in bigger projects and (re)compiles only the necessary files.

**rcs/cvs**  Revision control system (when working on bigger projects)
Compiling a program 1: The compiler

There is a hierarchy of steps to get a program compiled. (I pick C for the examples, but the process is always the same)

The easiest case is a single file program:

```
cc myprog.c
```

produces a binary `a.out`. (both Compilation and Linking).

One can run the program by `./a.out`.

Alternatively:

```
cc -o myprog myprog.c
```

produces a binary `myprog`
Compiling a program 2: Compilation Only

Larger programs consist of several files which get compiled separately and then are *linked* with a library of runtime routines.

```
c -c file1.c
c -c file2.c
c myprog file1.o file2.o
```

**Make**

The files from which a program is built are listed in a file called `Makefile`. A call

```
make myprogram
```

(or sometimes simply `make`) will do all compilations and linking.
The Makefile also lists dependencies and causes recompilation of the necessary parts if single files changed:

```
# Makefile for `myprog'
file1.o: file1.c
    cc -o file1.o -c file1.c
file2.o: file2.c file1.h
    cc -o file2.o -c file2.c
myprog: file1.o file2.o
    cc -o myprog file1.o file2.o

default: myprog
```

Often a Makefile also contains extra targets, for example for testing the program, building the documentation and installing the program for all users.

test: myprog
myprog exampledata >testout
diff testout correctout

install: myprog
    install ....

clean:
    rm *.o testout

Thus a typical installation would go:

make myprog
make test
make install
Compiling a program 3: System Dependencies

Not all computers are the same and often some things (say file paths) have to be adapted. This is often done by editing the Makefile or a file `config.h`. Over time this became extremely tedious and an automatic method was developed:

- The program author describes these dependencies with certain variables (For example: Names of the C compiler)

- A preprocessor produces a program (actually a shell script) `configure` which will check all the system dependencies.

- The user simply calls `./configure`. This will create a Makefile with properly set variables.

- Compilation then continues with `make` as before.

There are many variants.