THE UNIVERSITY OF MELBOURNE
Australia

UMP–98–19

School of Physics Computer System Manual 1998

Lloyd Hollenberg and Robert Scholten

With 1997 updates by: W. Langer and M. Munro

Contents

1. INTRODUCTION

This manual is designed as a quick reference guide to the computing facilities here at the School of Physics. The material contained herein is primarily directed to fourth year students, however, anyone who needs to gain a working knowledge of the system and the various applications available should find the information useful. In no way should this manual be considered an exhaustive reference for any of the topics – all areas are covered in a format whereby the reader learns by example. For detailed and more advanced information on the topics covered in this manual the reader should consult the references recommended in various sections.

1.1. The Baker Lab

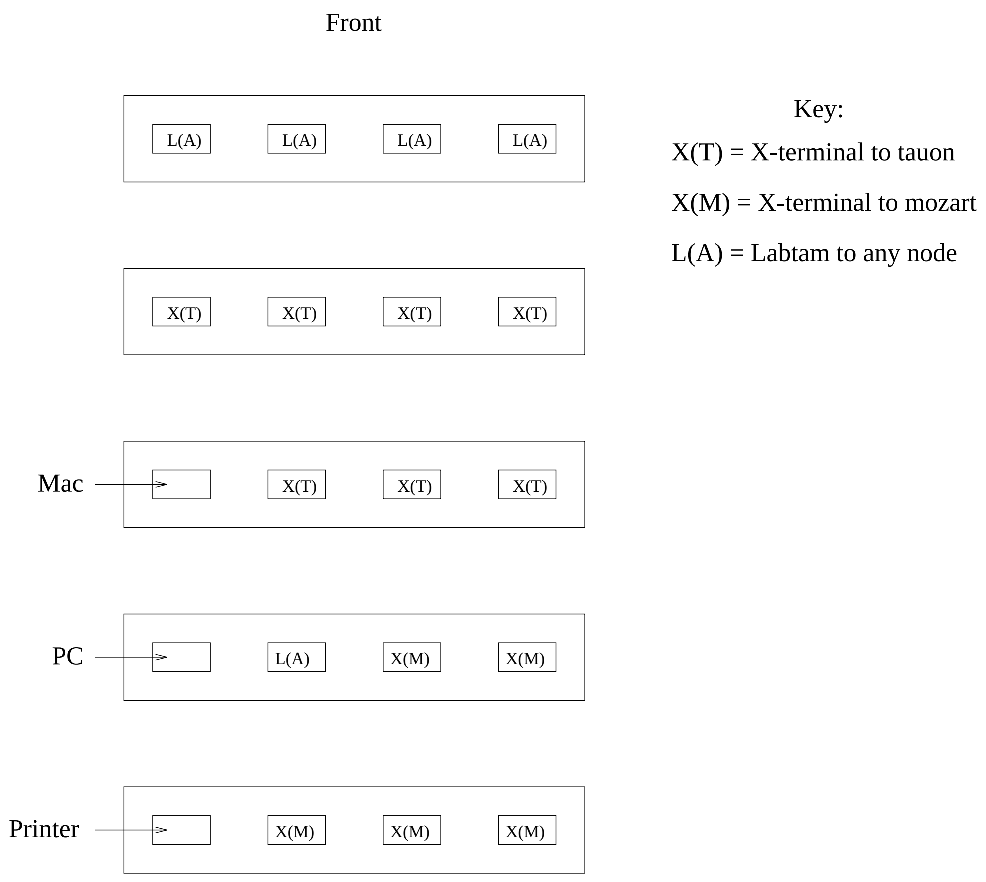
The computer system consists of a number of distributed workstations strategically located throughout the building all networked together and to the campus wide University and international networks. One of the school's key initiatives has been to introduce a Computer Laboratory to increase the availability of computing sites to researchers and as a teaching tool for undergraduates. This laboratory is located in the East wing of the Physics building on the 4th level. This lab is commonly called the **Baker Lab** after the Baker Foundation who graciously donated to its establishment.

Obtaining an account:

To obtain an account see the systems manager, Dr. Mark Munro, in the office at the back of the Baker Lab. Your account will be created on the School's main computers, **tauon** and **mozart**, plus any computers in your research group. Your username will be the same on all computers (and historically is constructed from your initials).

Whilst this manual will be written specifically for Baker lab users whose main accounts are on **tauon** (Digital Alpha Station) and **mozart** (IBM RISC Workstation), most of the material will be applicable to the other workstations on the School's network.

The Baker Lab terminals are set out in the diagram overleaf. You can connect to one of main computers from either a **X-terminal** or a **Labtam** terminal (the latter are faster).



As far as your account and file system is concerned, logging into **tauon** or **mozart** is essentially identical: i.e. **mozart** accesses the files on **tauon** by default. In terms of speed **tauon** is about 2-3 times faster than **mozart**. Although you will initially log onto either a **tauon** or **mozart** session to begin with, it is a simple matter to bring up a window for any other computer on the system, so you are not restricted to the computer designated by that particular terminal.

Users should be reminded of the following simple rules:

1. Your computer account is to be used by you and no one else but you. Do not give your password to any one else.
2. Choose your password carefully. Try to make it as cryptic as possible while still being able to remember it. If your password is simple or obvious then your security and the security of the whole system is compromised.
3. Do not go snooping around other people's accounts. Respect their privacy.
4. No material of an obscene nature is permitted on the system.
5. Treat ALL equipment in the lab with respect. Do not bring food or drink into the Baker Lab.

6. Try to be as quiet as possible. People congregating around a single X-terminal tend to make a lot of noise and distract others from working.

1.2. Logging in

The first requirement of any computing session is to login to **tauon** or **mozart** by entering your username and your password. Use the following procedure (depending on which type of terminal you are at):

1. Find a Labtam or X terminal (if Labtam choose a node to login to)
2. Make sure the mouse pointer is located over the login box.
3. At the Login prompt, type your **username** and **password**.

Tauon login

After typing your username and password correctly the Common Desktop Environment (CDE) will automatically start – this is the default desktop for the Digital Alpha Station. The CDE is a fairly straightforward windows system based on the mouse and has ample online help available.

Mozart login

Once you have performed a valid login sequence, the Message of the Day is displayed on the screen - make sure you read it as it may contain important information. Click a mouse button or press any key to start the X Window system and the OSF/Motif Window manager.

Several windows should now appear on your screen including helpful X clients like **xclock**.

You have successfully performed a login sequence and started the Motif Window manager.

1.3. Logging out

Tauon log out

To log out click on the small exit box on the command window at the bottom of the screen.

Mozart log out

To logout, move the cursor to a free space in the background root window. Press mouse button 3 (right hand side) in this root window and hold it down – the **root menu window** should now appear. With button 3 still pressed, drag the cursor down till **End Session** is selected, move the cursor to **ok** and release the button. A login window should now appear identical to the one you logged in on. You have successfully logged out.

2. BASICS OF UNIX

Unix is an operating system which allows many users to access the resources of a computer or network simultaneously. Although the user interface can appear unfriendly to begin with, it can be quickly customised to match the needs of each user, making it a flexible and powerful system. Conceptually the UNIX operating system can be divided into three parts. The *kernel* does the actual work, speaking directly to the computer and dealing with the allocation of computing resources as it carries out various tasks. The *shell* is the interface between the kernel and the user. The shell accepts input from the user, interprets this input, and relays it to the kernel. The *utility programs* or "commands" allow you to ask the shell to carry out certain tasks, for instance, to display, print, copy, move, search, delete, or otherwise manipulate files. Some of the more common commands are described in this section - a more comprehensive list can be found in the glossary at the end of this manual and in the references cited.

2.1. The shell

The shell is a utility which processes your commands, relaying this information to the kernel which then carries out the required task. Several different customised shells exist, so that users can vary their interface with the computer if desired. Although all the shells interface with the same UNIX kernel, your UNIX account will behave differently depending upon which shell you are running.

The shells can be distinguished by their different *prompts*. Your account will initially be set up to run under **csh** (pronounced "c-shell") which has a % as the prompt. Updating to more advanced shells is very simple and can be done at a later date.

2.2. Getting help in UNIX

If you know the name of the command you wish to use, then you can type the following command (do not type the prompt, %)

```
% man <name of command>
```

This will bring up on-line information describing the function of this command, how to call it up, the various options or "flags" which can be used to alter its function, and often some examples of its use. If this information takes up more than a page it will be "piped" to the **more** utility, a program which displays information one page at a time - hit the space-bar to read the next page. If you wish to know more about how to use “more”(for instance how to scroll backwards), try “man more”. The information provided by the man command is often confusing, however it is a good place to begin. If you’re still stuck, or can’t remember the exact name of the command you want to find out about, consult the glossary at the end of these notes, a text on UNIX, or ask someone else in the lab.

2.3. Customising your account

One thing which the shell interfaces all seem to share is the principle that “No news is good news”(Sobell, 1984). Unlike menu-driven systems, they do not prompt you for commands, and they give very little information about whether a task has been carried out successfully. Making good use of the “man “ command to gain information about the commands you are using can alleviate this to an extent. To get around the problem of terse and not terribly mnemonic command names, you can use the **alias** command, which allows you to make up your own name for a command or sequence of commands (see section 2 2.4).

Each shell is also a programming language, in which you can construct “shell scripts”, which allow you to combine sequences of commands to carry out tasks which you may require but which are not already included in the utility packages. If you are interested in this aspect of Unix consult a text to find out more.

An important example of shell-scripts are the “startup” files which contain scripts which are run automatically each time you log into your account. These contain important information such as the location of various important directories, and other account defaults. The names of these “startup” files always begin with a full-stop, and are sometimes called “hidden” files since they are not listed when you use the “ls” command, even though they are always there. To list all files including hidden files type “ls -a” at the prompt.Once you know how to use the *vi* or **emacs** editors, you can use one of them to display and edit the contents of these files, just as you would any other file. For instance, if you wanted to set up a permanent alias for a command you used all the time, you would edit your .cshrc file, adding to the list of aliases which it contains (see also section 2 2.4).

WARNING Before making any more complex alterations to these files, always make a copy of the original, and enlist somebody who has done it before to help when you first try it.

2.4. Shortcuts for typing in commands

It is useful to rename a complicated command which you might use often, to something shorter to type. For example the file listing command **ls -la** could be *aliased* to simply **dir** by entering the following command (again, do not type the prompt, %)

```
% alias dir ls -la
```

The command **dir** now replaces the more cumbersome **ls -la**. Aliasing can be done automatically everytime you log on or create a new window, by editing your **.cshrc** file which can contain, among other useful things, a list of aliases. There will be a file **.cshrc** already on your account and it will contain a default list of aliases which should serve as a guide.

Unix has a "wildcard" pattern matching feature, useful for avoiding typing long filenames, or for performing the same action to a whole set of similar files. An asterix in a name is interpreted by Unix as a wild-card, standing in for some unknown set of zero or more characters. For instance,

```
% ls file*.txt
```

will list only those files in the working directory which have the form "file(something).txt", so you might get a list such as:

file1.txt
file2.txt
filenames.txt
fileNew.txt

Typing **ls *.tex** will list all the files with the extension ".tex", and so on. If for some reason you wanted a printout of all the text files in your directory, you could type **lpr *.txt**.

A great deal of typing can be saved by using the *history* of commands. Typing the **history** command will produce a list of the most recent commands entered in that window. Any of these commands can be accessed (but not necessarily repeated) by using the cursor keys to scroll through and around the list of previous commands, or repeated by numerical reference – e.g. command #10 is repeated by typing

```
% !10
```

Alternatively, the last command starting with a particular letter(s) can be repeated by typing the first letter(s) of that command before the exclamation mark, e.g. the last *vi* command can be repeated by typing

```
% !vi
```

Be careful that you do not repeat an unwanted command which happens to start with the same letter – you may have to specify several letters to avoid making costly mistakes.

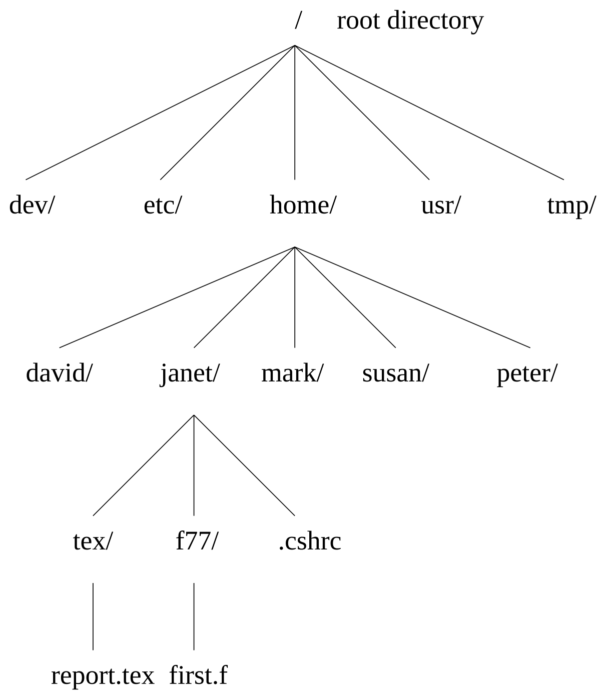
Another time saver is file-completion: under **csh** the escape key will automatically complete a half written filename (if unambiguous).

3. THE UNIX FILE SYSTEM

The Unix file system is a hierarchical one consisting of files and directories. The file system is similar to other operating systems such as DOS or VMS but with extensions for security and access permissions. A file can be a program, data file or text file. A **directory** is special type of file which contains other files. Directories are extremely useful in keeping common files grouped together rather as opposed to having unrelated files all mixed together.

3.1. The File System and basic UNIX commands

The typical filesystem on a UNIX system is set out as follows:



The directory separator character is /. All file systems have a **root** or starting point from which all directories branch. The **root directory** is known as /. In the example above, **home** is a subdirectory of /, **janet** is a subdirectory of **home**, **f77** is a subdirectory of **janet** and **first.f** is a file residing in the **f77** directory.

By convention, all systems operating with Unix generally have the following standard directories:

/home containing the home directories of the various users on the system.

/bin and **/usr/bin** which contain most of the standard Unix programs. Conventionally, the more common utilities are kept in **/bin** and more obscure commands or programs specific to the installation are kept in **/usr/bin**.

/dev files that represent devices, such as the printer, are kept here.

/etc contains miscellaneous files, including the read-only **passwd** file, which lists all the users who have permission to use the system.

/tmp temporary files.

Here are some useful UNIX commands to help you get around the filesystem (use the **man** command to get additional help on any of these and related commands).

Finding your location:

To find out where you are in a filesystem type the following command (do not type the prompt, %)

```
% pwd
```

For example if Janet in the above diagram was in her **f77** directory the result of the **pwd** command (print working directory) would be

```
% pwd
/home/janet/f77
%
```

Creating a directory:

To create a directory called **papers** for example, type the command

```
% mkdir papers
```

Moving around:

The **cd** command is used to move around the directory tree structure. For example to move to the newly created directory called **papers**, type the command

```
% cd papers
```

To move up a directory type the command

```
% cd ..
```

To move to a directory using absolute path names, the command would be something like

```
% cd /home/janet/f77
```

Listing files and their properties:

To list the files in the directory **papers** for example, type the command

```
% ls papers
```

(If you are already in the directory you need not type the directory name.)

More information about the files, such as privileges, size in Kbytes and date of last modification, can be displayed by adding options to the **ls** command. For example in the directory **janet**, the command

```
% ls -la
```

will produce the following output

```
drwxr-xr-x   3 janet      theory       512 Dec 22 18:40 ./
drwxr-xr-x   3 janet      theory       512 Dec 22 18:40 ../
-rwxr-xr-x   2 janet      theory      4539 Nov  6 09:29 .cshrc*
drwxr-xr-x   2 janet      theory      1024 Dec  3 03:00 f77/
drwxr-xr-x   2 janet      theory      1024 Dec  4 02:00 tex/
```

The information in the first ten columns, for example **drwxr-xr-x**, corresponds to the type of file and its access privileges. The **d** in the first column indicates a directory. The first series of privileges **rwx** indicates that the *owner* has read, write and execute privileges to that file. The second series **r-x** indicates that members of the same group (in this case the **theory** group) have read and execute privileges only. The third series **r-x** indicates that all other users of the system also have read and execute privileges only. Privileges can be changed using **chmod** (careful you retain your user privileges!).

Creating and editing files:

To create an empty file called **myfile**, type the command

```
% touch myfile
```

To write text to a file (new or old) use the **vi** editor (see the attached notes on the **vi** editior) by typing the command

```
% vi myfile
```

The **touch** command can be bypassed by the **vi** command.

Typing out a file:

The contents of a file can be displayed to the screen using the **more** command,

```
% more myfile
```

The spacebar scrolls through one screen at a time, whilst the enter key scrolls one line at a time.

Copying files:

To make a copy of a file use the **cp** command.

```
% cp file1 file2
```

The above command assumes that a file called **file1** exists and makes an identical copy of this file and calls it **file2**.

Deleting files and directories:

To delete a file use the **rm** command.

```
% rm file1
```

This command completely removes the file called **file1** from the file system. Be careful of this command as it can be dangerous. Your account has been set up so that the **rm** command asks for confirmation before deleting the file. To delete a directory there is a special command

```
% rmdir mydir
```

which will delete the directory **mydir** if and only if the directory is empty and contains no files.

Renaming files:

To rename a file use the **mv** command.

```
% mv file1 file2
```

This command renames the file called **file1** to a file called **file2**. This command has a special meaning if a file is moved between directories

```
% mv file1 mydir/file2
```

This command copies the file **file1** to the directory **mydir**, renames it to **file2**, then deletes the file **file1** (ie. the file is moved!) .

3.2. Printing

To print a file using the **bakerps** printer in the Baker Lab the file is usually in **postscript** format. To print a file already in postscript format (e.g. the file **paper.ps**) the command is

```
% lpr -h -P bakerps paper.ps
```

The option -h removes the header page which is of no practical use and wastes paper. To save paper you can print double sided using the command

```
% lpr -h -P bakerps-duplex paper.ps
```

In order to print a plain text file (e.g. **paper.txt**) it must first be converted to postscript. This is easily done using the **psf** command.

```
% psf paper.txt > paper.ps
```

The text file **paper.txt** has been converted to a postscript file and the output of the **psf** command has been directed to a file called **paper.ps** and can be printed using the **qpr** command. Alternatively the text file can be printed in one command using a **pipe** as follows.

```
% psf paper.txt | lpr -h -P bakerps
```

The output of the **psf** command is “piped” into the **qpr** command.

The status of a printing job can be checked using the **lpq** command.

3.3. Copying and pasting with the mouse

The mouse buttons allow you to copy and paste text on the screen, either in line mode (UNIX commands etc) or when editing in *vi*. Dragging the mouse while holding the left button down copies the text. Pasting at the cursor is done by pressing the middle button.

4. THE *VI* EDITOR

There are several editors available on the UNIX system (e.g. **vi**, **emacs**, **xedit** and **axe**). Here we will concentrate on the **vi** editor.

Suggested reading: "Introduction to the UNIX Operating System", by V.Y. Hansper (copies available in the Baker Lab)

To edit a file (e.g. the file **first.f**) using **vi** the command is

```
% vi first.f
```

Moving around the file:

The cursor keys will allow you to move around the file when you are not in insert mode. Some other commands are:

w → move forward one word.
b → move backward one word.
$ → move to end of the line.
shift g → go to end of file.
:20 → go to line 20.

Changing text:

i → insert at the cursor.
esc key → escape from insert mode.
o → insert below current line.
dd → delete line at cursor.
cw → change word at the cursor.
dw → delete word at the cursor.
r → replace character.
yy → yank the line of text at the cursor.
p → paste yanked lines below cursor.
shift j → join next line to end of current line.

Saving changes, quitting and file handling:

shift zz → exit *vi* session **saving** changes.
:q! → quit *vi* session **without saving** changes.
:w! → save the file but stay in *vi* session.
:r letter.tex → read in file **letter.tex** (give full pathname if it is not in the working directory).

Searching, substitution and miscellaneous:

/string → find the next occurrence of the word "string".
n → repeat the last find command.
:%s/string1/string2/gc → substitute string2 for string1 globally (option *g*) and confirm each substitution (option *c*).
. → repeat the last modification.

Note that you can specify multiple actions: i.e. **3dd** deletes three lines starting from the cursor position, **5yy** yanks five lines from the cursor position, **10w** moves forward by ten words, etc.

The **.exrc** file has defined function keys **F1** to **F4** as a fast cut, copy and paste facility. The specific functions are

F1 → mark start of cut or copy.

F2 → cut lines from start mark to the cursor line.

F3 → copy lines from start mark to the cursor line.

F4 → paste lines after cursor line.

These function keys are more convenient than using the mouse or the yank and put commands.

5. THE EMACS EDITOR

`emacs` is a powerful and ubiquitous editor. For example, you can compile FORTRAN, run LaTeX on your documents, ftp files, do email, compress and uncompress, tar and untar files, and many more things, all from within `emacs`. However, five minutes is enough to learn the essential. It works from any terminal, and is menu-driven when run in an X window.

Starting and quitting `emacs`

To start `emacs`:

```
emacs <filename> &
```

(the character & at the end of the line indicates that the job will be run in background) or alternately just:

```
emacs &
```

and then type away (or input a file; see below). `emacs` can also be run within an existing `xterm` window:

```
emacs -nw
```

where `-nw` means "no window".

In `emacs`, anything you type will go into your document/file/program, unless you use the $\boxed{\texttt{Ctrl}}$ key or $\boxed{\texttt{Alt}}$. These give you access to commands, for example `C-f` which means hold down the `Ctrl` key, then tap the `f`, then release the `Ctrl` key. `M-v` means "Meta"-v, where the `Meta` key is labeled `Alt` on most keyboards. If your keyboard doesn't have an `Alt` key, use the `Esc` key (don't hold it down), then the `v`.

To exit `emacs`: `C-x C-c`

Note that you will be prompted if any files need to be saved.

Moving around

You *can* use the arrow keys, but you'll be better off using the keys right under your fingers:

`C-f`	`forward`
`C-b`	`backward`
`C-n`	`next` line
`C-p`	`previous` line
`C-a`	beginning of line
`C-e`	`end` of line
`C-v`	one screen forward
`M-v`	one screen backward

Undo

`C-u`	`undo` `emacs` allows you to undo things infinitely.

Files

`C-x C-f`	`file open`	you will be prompted to enter a filename
`C-x i`	`insert file`	you will be prompted to enter a filename
`C-x C-s`	`save file`	

A useful "file" to open is a remote ftp site. As in, `C-x C-f //ftp@tauon:/` which will open tauon.

Directories

You can open a directory too – and then open files by moving the cursor to the file, then hitting `f`. Other things you can do when editing a directory:

`f`	`file open`	
`m`	`mark`	mark multiple files, then copy/delete/etc.
`u`	`unmark`	
`d`	`delete`	
`C`	`copy`	
`R`	`rename` (move)	
`Z`	compress/uncompress	
`+`	add new directory	
`h`	`help`	to see many more options

Delete

`C-d`	`delete` forward
`C-h or Backspace`	delete `backward`
`C-k`	`kill` to end of line
`C-w`	`wipe` (region; see Marking, below)
`C-y`	`yank` back last thing deleted

Mark (cutting and pasting)

`C-@ or C-SPC`	`mark`

Here `SPC` means "space". Once you have marked a point, you then move your cursor somewhere else and for example:

`C-w`	`wipe`	(cut)
`M-w`	`copy`	like `wipe` but leaves existing text
`C-y`	`yank`	(paste)

When you `wipe` a block of text, it is copied into another buffer (called the `kill` buffer), so you can then `yank` it back into your text somewhere else. You can yank back multiple times.

Search and Replace

`C-s`	`search` forward	
`C-r`	`reverse` search	
`M-%`	search and replace	`emacs` prompts for input

Rectangles

`emacs` allows editing of rectangular regions. This is particularly useful for deleting columns of numbers. Mark the top left of the rectangular region, then move the cursor to the bottom right, and:

`C-x r k`	`kill`	delete rectangular region
`C-x r y`	`yank`	paste rectangular region back
`M-x clear-rectangle`		replace region with spaces

Keyboard Macros

Very simple macros can be quite useful:

`C-x (`	start recording macro
`C-x )`	finish recording macro
`C-x e`	execute recorded macro

6. PROGRAMMING IN FORTRAN

Most computer programs can usually be broken up into three distinct parts: input, operation (e.g. calculations) and output. In these notes, as a quick introduction to the fortran language, a template program will be presented which includes these essential features. This document is designed not as a reliable reference to the fortran language, but as a quick guide to getting started. For more detailed and complete explanations you should refer to the fortran manual in the Baker lab (**not** to be removed!).

Suggested reading: "Fortran 77", by L.P. Meissner and E.I. Organick.
"Problem Solving and Structured Programming in Fortran 77", by E.B. Koffman and F.L. Friedman.

6.1. Editing, compiling and running code

Consider a file called **first.f** which contains the following lines of fortran.

```
c       first program
        implicit none
        real*8 x,f,a
        a = 2.0d2
        write(*,*)' Code to calculate f = ((x * 200)/(x + 200))**2 '
        write(*,*)' input x: '
        read(*,*)x
        f = (x*a/(x+a))**2
        write(*,*)' f = ',f
        end
```

This code takes a real number x and computes the function

$$f(x) = \left[\frac{200x}{200+x}\right]^2$$

Note that the lines beginning with a "c" do not form part of the executable program but allow for comments. All other lines begin at the first **tab** mark (8th column for the X-terminals in the Baker Lab). A detailed description of the code follows.

Line by line description:

```
implicit none
```

This command guards against typographical errors - each variable has to be declared. The debugging time saved by this device will become apparent as you gain more experience.

```
real*8 x,f,a
```

The variables **x,f,a** are defined to be real numbers represented by 8 bytes (64 bits). This is called **double precision** as opposed to **single precision** which corresponds to 4 bytes. A double precision real number has 19 places. Variables can also be declared as single or double precision **integers**. To avoid rounding errors one usually works with double precision integers and real numbers. Variable names can be long to help make the program easier to write and debug; e.g. **integral_of_f** can be a variable name.

```
a = 2.0d2
```

Here the variable **a** is assigned the value 200.0; i.e **2.0d2** means the same as 2.0×10^2 for double precision numbers. For single precision numbers the same value would be written as **2.0e2**.

```
write(*,*)' Code to calculate f = ((x * 200)/(x + 200))**2 '
```

This is an example of how to write to the screen. The first argument of the write command (i.e. where the first asterix is) specifies where the data is to be written - eg to a file or to the screen. The asterix directs the data to be written to the screen. The second argument specifies how the data is to be written - in this case the asterix means that the data is to be written in free format (the default setting). One can direct the write command to output the data in a special format - i.e. specify the number of decimal places of numbers, tabulate etc. (see the template program in section 4.4). The data in this case is a **character string**.

```
write(*,*)' input x: '
```

Same idea as above. This line will appear below the previous line on the screen.

```
read(*,*)x
```

The variable x is read in from the screen (the default setting determined by the first asterix) in free format (set by the second asterix). More variables could be read in by separating with commas. One can modify the arguments so that the variable is read from a file and/or in a predetermined format (see the template program in section 4.4).

```
f = (x*a/(x+a))**2
```

An example of basic mathematical operations. Note the use of parentheses.

```
write(*,*)' f = ',f
```

The variable **f** is written to the screen. Note that the character string contained in the ' ' marks and the variable **f** are separated by a comma.

```
end
```

Indicates the end of the program.

Once this code has been entered and saved in the file **first.f**, it can be *compiled* using the command

```
% f77 -o first first.f
```

The **f77** command checks the syntax of the fortran code and if correct creates an executable file called **first** (without the -o option the default executable file is called **a.out**). The program is executed (or run) by typing the name of the executable file

```
% first
  Code to calculate f = ((x * 200)/(x + 200))**2
  input x:
```

6.2. Running code in background

An executable code **first** can be run in background using the command

```
% first < in > out &
```

The file **in** is a text file containing the input required for the code which is normally entered from the screen. All the output from the code written to the screen in an interactive run will be written instead to the file **out**. The character **&** at the end of the line indicates that the job will be run in background, leaving the terminal free – i.e. you can logout and the job will continue to run.

The status of any job can be checked using the **top** command. To abort an interactive job hit Ctrl-C in that window.

To kill a batch job, first find the process identification number (PID) using the **ps** command (or **top**). For example

```
% ps -u janet
 UID   PID    TTY  TIME CMD
 316 10297  pts/2  0:01 aixterm
 316 11655  pts/6  0:03 vi
 316 2735   pts/8  4:25 first
```

The batch job **first** can be stopped by typing **kill** followed by the PID number corresponding to the job (in the above example this is 2735), i.e.

```
% kill 2735
```

If for some reason the job still continues then repeat the **kill** command with the option **-9** (this is a kill-with-extreme-prejudice command).

It is important to reduce to priority of your batch job in order that interactive users are not unduly affected by background number crunching. The **renice** command will alter the priority (0 is the highest, 20 is the lowest) of any given PID. For example, the job **first** in the above example can be reduced from the default level 0 down to level 10 by typing

```
% renice 10 2735
```

The priority level of a batch job will show up on the **top** command output.

6.3. Optimization

Once a program is debugged and running correctly it can be compiled with the optimizing option **-O** which will increase the speed by a factor of two in most cases. The compiling command would look like

```
% f77 -O -o first first.f
```

6.4. Subprograms: functions and subroutines

Often one needs to evaluate a function many times or to repeat a certain number of steps in a program. In order to make programs easier to organise one can use sub-program units to perform these tasks which can be called from the main program. The following code shows how this can be done for the simple case of the program **first.f**.

```
c       second program
        implicit none
        real*8 x,f,c
        c = 2.0d2
        write(*,*)' Code to calculate f = ((x * 200)/(x + 200))**2 '
        write(*,*)' input x: '
        read(*,*)x
        call calculation(x,c,f)
        write(*,*)' f = ',f
        end
c------------------------------------
        subroutine calculation(x,c,f)
        implicit none
        real*8 x,c,f,r
        f = r(x,c)
        return
        end
c------------------------------------
        double precision function r(x,c)
        implicit none
        real*8 x,c
        r = (x*c/(x+c))**2
        return
        end
```

Line by line description:

```
call calculation(x,c,f)
```

This statement executes the lines contained in the subroutine **calculation** with three arguments (i.e. variables that the subroutine requires as inputs or returns) **x**, **c** and **f**. The value of the variables **x** and **c** have been set in the main program and are used in the subroutine whereas the value of the variable **f** is set in the subroutine and returned to the main program.

```
subroutine calculation(x,c,f)
```

The subroutine is a seperate program unit with variable declarations. The values of the variables **x** and **c** are given in the arguments of the subroutine call from the main program. In this case the value of **f** is set in the subroutine (via a function call to **r(x,c)**. The variables **x** and **c** remain unchanged. The **return** statement at the end of the subroutine passes control back to the line in the main program after the subroutine call as well as the values of the arguments at the point of return statement.

```
real*8 x,f,c,r
```

The variable **r** representing the function to be evaluated in this subroutine has been declared as double precision real. If a call to this function was made from the main program, the same declaration for **r** would have to made at the start of that program.

```
f = r(x,c)
```

In this statement the double precision function **r(x,c)** is called, passing the variables **x** and **c** to be used in the function subprogram to evaluate **r**. The variable **f** is set to the value of **r**.

```
double precision function r(x,c)
```

The function **r(x,c)** is also a seperate program unit with variable declarations. The function call returns a value for **r** which is treated as a variable in the main program.

6.5. Template program

The following program, **template.f** encompasses most of the basic fortran needed for ordinary programming. The program itself is of no practical use, but is merely intended as a template for building bigger and better code. It can be found in **/home/4thyear/template.f**.

This code evaluates the function

$$f(x) = \frac{a}{1+x} \qquad , -1 < x < 1$$

by power series approximation

$$\frac{a}{1+x} = a\{1 - x + x^2 - x^3 + x^4 + \ldots\}$$

to increasing orders and compares with the 'exact' answer (exact to double precision). Note the liberal use of **documentation**!

```
c       template program
        implicit none
        real*8 x,sum(0:100),term(0:100),exact
        real*8 cumulative_sum,abs_x,f,a
        integer*4 i,max_exponent,exponent_last_term
        common/some_variables/a
        open(unit=1,file='template.out',status='unknown')
        max_exponent = 100
        a = 2.0d0
        write(*,*)' Code to calculate f = a/(x + 1) by power series'
        write(*,*)' where a has been set to ',a
c--------------------------------------------------
c       -> input the number of terms in series
10      write(*,*)' input exponent of last term in series: '
        read(*,*)exponent_last_term
c--------------------------------------------------
c       -> check to see if exponent_last_term > max_exponent
        if(exponent_last_term .gt. max_exponent) then
        write(*,*)' last exponent too large, max is ',max_exponent
        goto 10
        end if
c--------------------------------------------------
c       -> input the value of x using subroutine input
        call input(x)
c--------------------------------------------------
c       -> compute the exact value of f = a/(1 + x)
        exact = f(x)
```

```
c-----------------------------------------------
c       -> use a do loop to calculate terms in the
c          series and store terms in the array term(i).
        do 20 i = 0 , exponent_last_term , +1
        term(i) = a*(-x)**i
20      continue
c       -> add up terms and store cumulative sum in sum(i)
        cumulative_sum = 0.0d0
        do 30 i = 0 , exponent_last_term
        cumulative_sum = cumulative_sum + term(i)
        sum(i) = cumulative_sum
30      continue
c-----------------------------------------------
c       -> output
        write(1,*)' series expansion for f(x) = a/(1+x): x = ',x
        write(*,*)' series expansion for f(x) = a/(1+x): x = ',x
        do 40 i = 0 , exponent_last_term
        write(1,50)i,term(i),sum(i),exact
        write(*,50)i,term(i),sum(i),exact
50      format(' term #',i2,' : term = ',f8.6,
     &         ' : sum = ',f8.6,' : exact value = ',f8.6)
40      continue
c       -> close output file and end
        close(unit=1)
        end
c-----------------------------------------------
        double precision function f(x)
        implicit none
        real*8 x,a
        common/some_variables/a
        f = a/(1.0d0 + x)
        return
        end
c-----------------------------------------------
        subroutine input(x)
        implicit none
        real*8 x,abs_x
10      write(*,*)' input x: '
        read(*,*)x
        abs_x = dabs(x)
c       -> check to see if -1 < x < 1
        if(abs_x.ge.1.0d0) then
        write(*,*)' x out of range -1 < x < 1 : try again '
        goto 10
        end if
```

```
      return
      end
```

Line by line description:

```
real*8 x,sum(0:100),term(0:100),exact
```

In this declaration the variables **sum** and **term** are one dimensional arrays (vectors) with indices ranging from 0 to 100. The array **sum** is composed of variables **sum(0)**, **sum(1)**, **sum(2)**,..., **sum(100)**. For repetitive calculations one usually tries to define variables as arrays since these can be easily called in do-loops.

```
common/some_variables/a
```

This defines a **common block** of variables (just one variable **a** in this case). The name of the common block is "some_variables" (it could have been called anything) to distinguish it if there were more common blocks. The variable **a** will now have the same value in any sub-program unit (main program, function, subroutine etc) which has this common block statement at the start. Note that the common block statements must occur after the declaration of variables and before any executable statement. The common block is a useful way of reducing the number of arguments in function and subroutine calls.

```
open(unit=1,file='template.out',status='unknown')
```

The file **template.out** has been opened for output and is used later on in the code. The **unit** number allows the program to refer to this file in a write statement. The **status** indicates that the data is written to a new file. Each time the program is executed the contents of **template.out** will be written over. The status option can have different definitions on other systems so one should refer to the fortran reference manual to be sure.

```
if(exponent_last_term .gt. max_exponent) then
write(*,*)' last exponent too large, max is ',max_exponent
goto 10
end if
```

If **exponent_last_term>max_exponent** then the two lines contained before the **end if** will be executed otherwise the program skips to the line after **end if**. Note that there is a **goto** statement which passes control back to line 10 - i.e. if **exponent_last_term>max_exponent** then the program will require another value of **exponent_last_term**. The complete set of simple logical operations are given by (eq, lt, gt, le, ge, ne) corresponding to $(=, <, >, \leq, \geq, \neq)$. More complicated logical operations can be constructed by the inclusion of **or** and **and** in the statement. See the fortran manual for a more complete description.

```
call input(x)
```

The value of the variable **x** is given by the subroutine **input** which asks for input from screen and tests the value for the condition -1 < x < 1. Normally one would include this sort of input code in the main program, but here we put it in a subroutine, merely as an example.

```
exact = f(x)
```

The exact value of the function $f(x)$ is obtained by calling the double precision function **f(x)**.

```
        do 20 i = 0 , exponent_last_term , +1
        term(i) = a*(-x)**i
20      continue
```

In this **do loop** the loop variable **i** is increased from 0 to the value of **exponent_last_term** in increments of +1 (one can leave the increment off in this case since the default is +1) and for each value of the loop variable, **i**, the lines contained in the do loop, i.e. those before the end specified as line 20, are executed.

```
write(1,*)' series expansion for f(x) = a/(1+x): x = ',x
write(*,*)' series expansion for f(x) = a/(1+x): x = ',x
```

In the first write statement the character string and variable **x** is written in free format (indicated by the asterix in the second argument) to the file specified as unit 1 (in this case it is the file **template.out**). The second statement writes the same data to the screen in free format.

```
write(1,50)i,term(i),sum(i),exact
write(*,50)i,term(i),sum(i),exact
```

The variables **i**, **term(i)**, **sum(i)** and **exact** are written to the the file **template.out** (first write statement) and to screen (second write statement) using the format specified by the format statement in line 50.

```
50      format(' term #',i2,' : term = ',f8.6,
     &          ' : sum = ',f8.6,' : exact value = ',f8.6)
```

The **format** statement gives control over the output of data. In this case we have one line of output which starts with the character string **term #** followed by the first variable in the write statement list, i.e. the variable **i**, which is written out as an integer of 2 places as specified by the argument **i2** (the “i” meaning integer). The remaining three double precision real variables **term(i)**, **sum(i)** and **exact** are in floating point format f8.6 - specifying a total of 8 places including the decimal point. See the fortran manual for a complete description. Note that since this format statement was too long to contain on one line it was broken into two lines by the use of the character **&** in column 5.

```
close(unit=1)
```

The file specified by unit 1 is closed.

```
double precision function f(x)
```

The subprogram unit to compute the function $f(x)$.

```
subroutine input(x)
```

The subroutine which inputs **x** from the screen and checks the validity of the value obtained.

```
abs_x = dabs(x)
```

This is an example of an **intrinsic function**. The intrinsic function **dabs(x)** finds the absolute value of a **double precision** number (as distinct from the intrinsic function **abs(x)** finds the absolute value of a **single precision** number). Other intrinsic functions include **dsin(x)**, **dcos(x)**, **dsqrt(x)** etc. See the fortran language manual for a list of intrinsic functions and their argument types.

The template program can be compiled and executed by the following commands:

```
% f77 -o template template.f
% template
```

The file **template.out** will contain the output.

6.6. Debugging and checking fortran code

Once a program compiles and executes one must check that it is working correctly. One must be able to test the numerical output of the program and find and correct bugs in the code. To check and debug a program one must write out variable values at various stages throughout the program execution and make sure the values calculated and being used in the program are correct. This is hard and tedious work, but it must be done thoroughly to avoid the disastrous situation of publishing incorrect results due to a flawed program.

Whilst there are no easier strategies in general for debugging a program than writing out variables and checking their values, there are facilities that make this job quicker by avoiding the need to constantly insert write statements and recompile the code. The debugging facility on UNIX used here is called **dbx**. It is invoked by compiling the code with the option **-g**, i.e.

```
% f77 -g -o first first.f
```

To run and debug the program one then enters the debugging environment using the **dbx** command:

```
% dbx first
```

A more advanced X windows version is **xde** for which the same compiler option **-g** must first be used.

7. EDITING, COMPILING AND RUNNING C CODE

Consider a file called **first.c** which contains the following lines of C:

```
/* first.c                                                              */
/* Purpose:     Simple demo C program - input, calculate and output     */
/* Author:      R. Scholten                                             */
/* Date:        6 July 1996                                             */
#include <stdio.h>            /* include standard I/O function defns  */

main ()
{
  float x,f,c;                /* define real-number variables */
  c=2.0e2;                    /* initialise c=200 */
  printf("Calculating f=((x*200/(x+200))**2\n");
  printf("Enter x: ");
  scanf("%f",&x);             /* get x from keyboard */
  f=(x*c/(x+c));              /* calculate term */
  f=f*f;                      /* and square it */
  printf("f = %f\n",f);

  return 0;                   /* program ended OK */
}
```

This code takes a real number input, uses this to calculate something and displays the result. The relationship between input and output is given by the algebraic formula

$$f(x) = \left[\frac{200x}{200+x}\right]^2$$

Line by line description

```
/* first.c                                                              */
/* Purpose:     Simple demo C program - input, calculate and output     */
/* Author:      R. Scholten                                             */
/* Date:        6 July 1996                                             */
```

All good programs should begin with a short comment section, including:

- program name
- program purpose
- author

- date

These lines are simply comments, intended to help you – and us – remember what this code is for. In C, anything between `/*` and `*/` can be disregarded, even if the `/*` and `*/` are on different lines.

```
#include <stdio.h>
```

This is a *compiler directive* which means it's a statement to the compiler, not an actual line of code. This example tells the compiler to include some information about the standard input/output library. You should always include this line.

```
main()
```

The program is a function called `main`. All C programs will have a `main` function somewhere, which is what will be executed when the program is run.

```
{
```

The left curly brace will have, somewhere later, a matching right curly brace `}`. These braces mark out the extent of the function `main`.

```
float x,f,c;                  /* define real-number variables */
```

All variables must be declared. In this case, `x`, `f`, and `c` are defined as real numbers. Names are case-sensitive, so `x` is **not** the same as `X`.

Also note the "`;`". The semicolon marks the end of the *statement*. In C, statements (code instructions) can be split over several lines, or there can be many statements on one line. The important separator is the semicolon.

Finally, note the comment on the same line. This would normally say something useful about what the variables are for.

```
c=2.0e2;                      /* initialise c=200 */
```

This statement sets the initial value of the `c` variable to 2.0×10^2, or 200.

```
printf("Calculating f=((x*200/(x+200))**2\n");
printf("Enter x: ");
```

These two similar statements print something on your screen. They are examples of *function calls*. They are instructions to C to call a library function, called `printf`, as defined in the standard input/output library specified by `<stdio.h>`. The stuff between the parentheses is a list of *arguments* which are passed on to the library function. In this case, there is only one argument in each call – a string defined by the double quotation marks, "".

The first string includes some straight text, and \n which tells C to output a *newline* after the message. Note that the second printf statement does not include the \n and so after displaying the message, the screen cursor will be just after the colon, waiting for your input.

```
scanf("%f",&x);                  /* get x from keyboard */
```

The scanf command is used to input something from the user (to "scan" the keyboard). The first argument, %f, is again a conversion specification. It instructs the scanf function to look for a floating point number.

The second argument is **almost** the name of the variable in which the entered number is to be stored. However, note the & ampersand. This means that scanf is not passed the value of x but rather the *address*; that is, the memory location associated with the x variable. Hence, when scanf gets the floating point (real) number, it knows *where* to store the result.

Forgetting the & is a very common error!

```
f=(x*c/(x+c));                   /* calculate term */
f=f*f;                           /* and square it */
```

Ah ha! Now the program is doing something. This statement calculates the term in parentheses in our equation and sets the value of the f variable to the result. The second line calculates the square.

```
printf("f = %f\n",f);
```

Finally, the program outputs the result. Note the %f which is a *format specifier* which means "output number as a floating point". The value of the second argument to printf, that is, variable f, is displayed assuming f is a floating point.

The floating point format specifier is more generally of the form %W.Df where the width W and decimal places D are specified, in this case 10 characters wide and 6 digits after the decimal place. The %d specifier is used for integers; others can be found by using the Unix man page for printf (try man printf at your console).

```
return 0;
```

At the end of every C function (in this case main) you can return a value. Returning zero indicates that everything was OK. It's not strictly necessary in this case, but good programming practice. You might use return -1 when there is an error.

```
}
```

Here's the matching right curly brace that marks the end of the `main` function – and the end of the program.

7.1. Compilation

Once the program has been entered and saved to a file (e.g. `first.c`) it can be *compiled.* This is readily done with:

```
  $ cc first.c -o first -lm
```

The `-lm` isn't strictly necessary here, but will be for later programs that use *math library* functions such as `exp`, `pow` and so on.

If your program is bug-free, you should now find an *executable* program named `first` in your directory, for example:

```
  $ ls -l
  -rwxr-x--x   1 scholten pfg        24064 Jul  7 19:03 first
  -rw-r--r--   1 scholten pfg          823 Jul  7 19:03 first.c
```

This listing shows the executable file `first` and the C source file `first.c`. You can execute the program just by typing in the name, `first`:

```
  $ first
  Calculating f=((x*200/(x+200))**2
  Enter x: 100
  f = 4444.444336
  $
```

7.2. Functions

The program above made good use of C's ability to compartmentalise tasks into discrete *functions.* The program itself, `main`, is a function, and it called two other functions, `printf` and `scanf`. You can also define your own functions. Here's how we might rewrite the above program to make use of functions.

```
/* second.c                                                          */
/* Purpose:     Demonstrate functions in C                           */
/* Author:      R. Scholten and L. Hollenberg                        */
/* Date:        7 July 1996                                               */
#include <stdio.h>           /* include standard I/O function defns  */

float input(float c);                          /* function declarations */
float r(float x, float c);

main ()
{
  float x,f,c;                 /* define real-number variables */
  c=2.0e2;                     /* initial c=200 */
  x=input(c);                  /* go ask user for x, for given c */
  f=r(x,c);                    /* calculate function, given x and c */
  printf("f = %f\n",f);

  return 0;                                /* program ended OK */
}

float input(float c)
/* display value of c; input value and return */
{
  float temp;                              /* local storage */
  printf("Calculating f=((x*200)/(x+200))**2\n");
  printf("Enter x: ");
  scanf("%f",&temp);                       /* get x from keyboard */
  return temp;
}

float r(float x, float c)
/* given x and c, calculate (x*200/(x+200)) */
{
  float temp;
  temp=(x*c/(x+c));                       /* calculate term */
  return temp*temp;                       /* return square */
}
```

Line by line description
The first lines are similar to those in the previous example. Then we find:

```
float input(float c);                          /* function declarations */
float r(float x, float c);
```

These are *function declarations*. They don't actually do anything, except tell the C compiler what's coming. They warn of two functions, `input` and `r`, both of which return `float` floating-point numbers. `input` will require one float argument (called `c`, although the name is completely arbitrary here); `r` wants two arguments, `x` and `c`.

These declarations are important, because when you use these functions, the C compiler will know *how* you should use them. It will know how many and what type of arguments, and what type of value is returned. This is important for debugging.

Several lines of code are again the same, and then:

```
x=input(c);                    /* go ask user for x, for given c */
f=r(x,c);                      /* calculate function, given x and c */
```

These are the function calls. The first sets `x` to be the value returned by function `input` when `input` is given as its argument the value of `c`. Thus we have "hidden" the `printf` and `scanf` statements. The second function call is similar.

Note that if we want to change the form of the function, we need only change the `r` function; the rest of the program would not change at all.

```
float input(float c)
/* display value of c; input value and return */
{
```

This is the start of the `input` function. Note the comment at the beginning so that the reader knows what the function is supposed to do. In this case it is trivial, but that is unfortunately not generally true.

```
float temp;                                 /* local storage */
printf("Calculating f=((x*200)/(x+200))**2\n");
printf("Enter x: ");
scanf("%f",&temp);                          /* get x from keyboard */
```

Here is the function itself. The `float temp;` declaration defines a *local* variable called `temp`. Anything done to change the value of `temp` within this function will not affect any other variable called `temp` in another function, including `main`.

```
  return temp;
}
```

Once `temp` is set to the value entered from the keyboard, it is returned as the value of the function `input`.

Function `r` is very much the same.

7.3. Program flow

Very few programs are so straightforward. One of the most powerful aspects of computers is their ability to execute some instructions many times over in a *loop*. For example, to take the sum of a series of numbers.

It is also necessary to be able to change the order of execution, depending on some critical decision. For example, you might want to take the square root of a number, but not if it is negative.

More complex data structures are also needed, beyond the floating-point numbers introduced so far. These include integers, double-precision floating points, and arrays.

Here's a typical example program that encompasses many of the basic building blocks needed to efficiently program in C. These include `if/then/else`, `do-while`-loops, and `for`-loops.

This program should form a good template for your own C programs; hence it is called `template.c`. It evaluates the function

$$f(x) = \frac{a}{1+x} \qquad -1 < x < 1$$

by power series approximation

$$\frac{a}{1+x} = a\{1 - x + x^2 - x^3 + x^4 + \ldots\}$$

to increasing orders and compares with the 'exact' answer (exact to double precision).

```
/* template.c                                                             */
/* Purpose:     Illustrate C conditional program flow, arrays            */
/* Author:      R. Scholten                                               */
/* Date:        7 July 1996                                               */
#include <stdio.h>                /* include standard I/O statement defns */
#include <math.h>                          /* include math (pow and fabs) */

double input();                                  /* function declarations */
double f(float a, float x);

main ()
{
  double x,                                          /* x=scratch variable */
         sum[101],            /* 101-element array,  sum[0] to sum[100] */
         term[101],
       exact,                                      /* exact result */
       cumulative_sum,                              /* etc etc etc */
       a;
long int i,j,k,                                 /* similar comments */
       max_exponent,
       exponent_last_term,
       nogood;     /* in C zero implies false, while non-zero is true */

char outname[80]="template.out";              /* name of output file */
FILE *outfile;                             /* pointer to output file */

/* First we open the file which will hold our results */
if ( (outfile=fopen(outname,"w")) == NULL) {
  printf("Life is bad.  Couldn't open file %s.\n",outname);
  return -1;
  }

max_exponent = 100;
a = 2.0e0;
printf("Code to calculate f = a/(x + 1) by power series,");
printf(" where a has been set to %0.2f\n",a);

/*---------------------------------------------*/
/*   input the number of terms in series       */
/*---------------------------------------------*/
do {
  printf("Input exponent of last term in series: ");
  scanf("%ld",&exponent_last_term);
  /*-----------------------------------------------*/
  /* check to see if exponent_last_term > max_exponent */
  /*-----------------------------------------------*/
  if (nogood=(exponent_last_term > max_exponent))
    printf("Last exponent too large, max is %d\n",max_exponent);
} while (nogood);

x=input();         /* input value of x using function "input" */
exact=f(a,x);      /* compute exact value of f=a/(1+x) */


printf("Exact value for f(x) = a/(1+x) (a=%f, x=%f) is %f\n",a,x,exact);

/*---------------------------------------------*/
/* use a for loop to calculate terms in the    */
/* series and store terms in the array term[i]  */
/*---------------------------------------------*/
```

```
  for (i = 0; i<=exponent_last_term; i++) {
    term[i]= a*pow(-x,(double)i);      /* Note the "cast" i.e. (double)i */
  }


  /* add up terms and store cumulative sum in sum(i) */
  cumulative_sum = 0.0e0;
  for (i = 0; i<=exponent_last_term; i++) {
    cumulative_sum = cumulative_sum + term[i];
    sum[i] = cumulative_sum;
    }

  /* output results to screen and to file */
  fprintf(outfile, "Series expansion for f(x) = a/(1+x): a=%f, x=%f\n",a,x);
  printf("Series expansion for f(x) = a/(1+x): a=%f, x=%f\n",a,x);
  for (i=0; i<=exponent_last_term; i++) {
    fprintf(outfile,"term # %2d  term = %10.6f  sum = %10.6f  exact=%10.6f\n",
            i,term[i],sum[i],exact);
    printf("term # %2d  term = %10.6f  sum = %10.6f  exact=%10.6f\n",
           i,term[i],sum[i],exact);
    }
  /* close file and end */
  fclose(outfile);
  return 0;
}

/*-----------------------------------------------------------------------*/
/* f(a,x)                                                                */
/* Purpose:     calculate a/(1+x)                                        */
/*-----------------------------------------------------------------------*/
double f(float a, float x)
{
  double temp;
  temp=a/(1.0+x);
  return temp;
}

/*-----------------------------------------------------------------------*/
/* input()                                                               */
/* Purpose:     input a floating-point number x and check -1 < x < 1    */
/*-----------------------------------------------------------------------*/
double input()
{
  double x,abs_x;
  int nogood;
  /* check for valid input */
  do {
    printf("Input x: ");
    scanf("%lf",&x);     /* note the "l" for "long" float (i.e. double) */
    abs_x = fabs(x);
    if (nogood=(abs_x >= 1.0e0))
      printf("x out of range -1 < x < 1 : try again\n");
  } while (nogood);
  return x;
}
```

Line by line description
Note the extensive use of comments to describe what the program, and each part of the program, is doing. Liberal use of such documentation is em essential.

```
#include <stdio.h>             /* include standard I/O statement defns */
#include <math.h>                       /* include math (pow and fabs) */
```

Note that in addition to `<stdio.h>` we also include `<math.h>`. The `<math.h>` header describes several mathematical functions that are needed, in particular `pow(x,y)` and `fabs(x)` which calculate x^y and $|x|$ respectively.

```
double x,                                         /* x=scratch variable */
```

This line is the first of several which declare some variables as type `double`. The exact definition of `double` is machine-dependent, but generally means an 8-byte number with about 13 decimal places of precision.

```
sum[101],            /* 101-element array,  sum[0] to sum[100] */
term[101],
```

These lines continue the `double` variable declarations. In this case, `sum` and `term` are one-dimensional arrays (vectors), with 101 elements (each `double`). These elements are accessed via indices ranging from 0 to 100, thus the array `sum` is composed of `sum[0]`, `sum[1]`,...,`sum[100]` (**not** `sum[101]`).

```
char outname[80]="template.out";               /* name of output file */
FILE *outfile;                               /* pointer to output file */
```

The first line here defines another array, this time of type `char`, which is used to hold a string of characters. It is pre-assigned the value `"template.out"` which is the name of an output file which is used later. The `FILE *outfile;` line defines a *pointer* to a file, which is opened for writing by the next few lines:

```
if ( (outfile=fopen(outname,"w")) == NULL) {
  printf("Life is bad.  Couldn't open file %s.\n",outname);
  return -1;
  }
```

Here the pointer `outfile` is set to point to the file opened by the function `fopen`. `fopen` is passed the name (`template.out`) and `"w"` which means "open `template.out` for writing". The `"w"` means that any existing file `template.out` will be written over. `fopen` can also be used to `"a"` (append to existing file) or `"r"` (read).

If `fopen` sets the pointer `outfile` to the special value `NULL` then `fopen` was unable to open that file (perhaps because the disk was full). The program outputs a message to that effect, and quits.

```
do {
  printf("Input exponent of last term in series: ");
  scanf("%ld",&exponent_last_term);
  if (nogood=(exponent_last_term > max_exponent))
    printf("Last exponent too large, max is %d\n",max_exponent);
} while (nogood);
```

This construct is a `do {} while` loop. The code between the two braces `{}` will be executed over and over again until the variable `nogood` is "false". `nogood` is set to `(exponent_last_term > max_exponent)`; that is, if `exponent_last_term` is greater than `max_exponent` then `nogood` will be true, in which case an error message is displayed and the `do {} while` loop is re-executed.

The *conditional* operator (`>`) can be replaced with other tests such as `==,!=,<,>,<=,>=`. Beware the "equal to" test which is a *double* `==`! More complicated tests can include `&&` (**and**), and `||` (**or**).

```
scanf("%ld",&exponent_last_term);
```

It is worth commenting on this line especially. `exponent_last_term` was declared as a `long int` and so when inputting a value from the keyboard, `scanf` must be told to convert the key entries to a `long int`. This leads to the

```
x=input();          /* input value of x using function "input" */
exact=f(a,x);       /* compute exact value of f=a/(1+x) */
```

The `input()` function is called to input a value; note there are no arguments. The exact value (to double precision) is calculated by the function `f(a,x)` using the input value of `x`.

```
for (i = 0; i<=exponent_last_term; i++) {
  term[i]= a*pow(-x,(double)i);       /* Note the "cast" i.e. (double)i */
}
```

This is an example of a `for` material between the braces `{}` (the body) is executed while the loop variable `i` $\leq$ `exponent_last_term`. `i` is increased by one **after** the body is executed, as specified by the `i++`. Other examples are `i--` and `i=i+7` which count down by one, and count up by 7, per loop iteration. `i` can also be changed in the body but this is not good programming practice because mistakes can be hard to find.

The body includes a couple of oddities. First of all, to calculate x^y in C you need to call a function, in this case `pow(x,y)`. Now, `pow` is a double precision function, and it expects double precision arguments for `x` and `y`. However, in this example the exponent y is an integer, `i`. This is converted or *cast* into a double precision floating point with the `(double)` type cast. Similar things can be done with `(float)`, `(int)` and so on.

```
fprintf(outfile,"term # %2d  term = %10.6f  sum = %10.6f  exact=%10.6f\n",
        i,term[i],sum[i],exact);
printf("term # %2d  term = %10.6f  sum = %10.6f  exact=%10.6f\n",
       i,term[i],sum[i],exact);
```

The `printf` statement has a cousin, `fprintf`, which prints to a file. In this case, the first argument is the *pointer* to the output file which was opened earlier, `template.out`.

```
fclose(outfile);
```

All open files should be explicitly closed before the program finishes.

```
abs_x = fabs(x);
```

The last remaining code of note call to the `fabs` function, which calculates the (double precision) absolute value of its (double precision) argument. There are many similar mathematical functions which can be found in a C manual. The Unix `man` command can be a valuable aid with such things, for example `man fabs` which not only describes the `fabs` function but also many similar functions, with references to other related material that can be obtained online.

8. PV-WAVE: GRAPHICAL DATA ANALYSIS

PV-WAVE, or just `WAVE`, is a powerful commercial program for analysing data, used by advanced research groups around the world. It can do simple 2D plots or sophisticated five-dimensional plots, and includes image processing, the IMSL numerical libraries, statistical and mapping packages, signal processing libraries, widgets and many others.

While powerful, it can also be reasonably straightforward to use. Some very simple examples of how to use it are included below; use the built-in `help` to find out more (including a tutorial).

To start `WAVE`, just enter the command:

```
  % wave
```

This assumes your environment variables have been set correctly; see section 8 8.6.

To close the program, just type:

```
  WAVE> quit
```

A demonstration program, written to provide an overview of `WAVE` abilities, can be run by entering:

```
  WAVE> demo
```

`WAVE` has built-in online hypertext Help:

```
  WAVE> help
```

and in fact the complete set of manuals, including the tutorial manual, can be accessed online with:

```
  WAVE> help,/doc
```

`WAVE` is interactive, so commands can be entered directly, and the language is very much like FORTRAN. Often, you will wish to enter a long series of commands. Usually, rather than entering them directly, it is best to treat `WAVE` in the same way that FORTRAN is used: that is, by editing a "program" outside `WAVE` which is then compiled and run inside `WAVE`.

Here's an example which reads a file of (x,y) data pairs, and plots the points with straight lines between them. In your text editor of choice, edit a file called `simp_plot.pro`. Type in the lines below (case is not important), and save it.

```
; WAVE program to read data file with two columns
; of numbers and plot as x,y graph.
PRO simp_plot
fname=' '                  ; name of data file
print,'Enter filename'
read,fname                 ; get filename from user
status = dc_read_free(fname, x, y, /Column)     ; read data
plot,x,y,title='File: '+fname, xtitle='x', ytitle='y', psym=-4
end
```

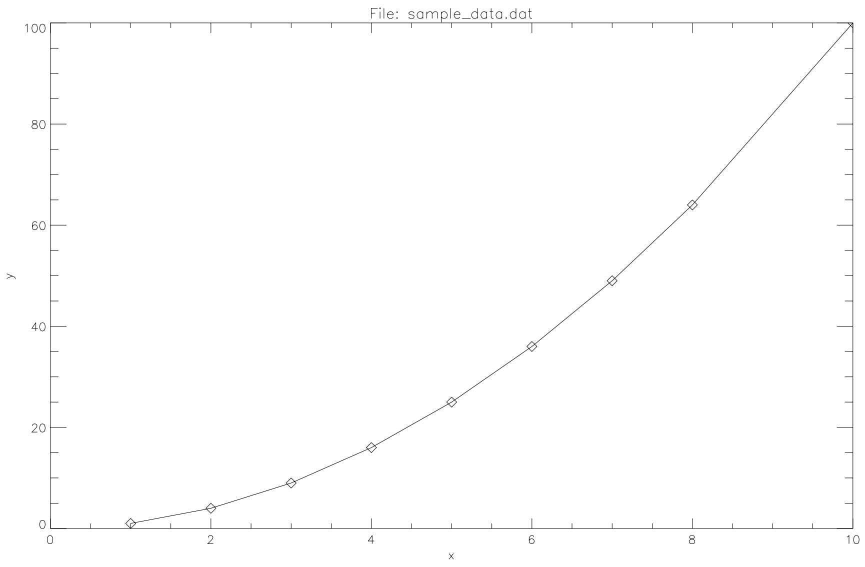
To run this script, start `WAVE` and compile it with the command:

```
  WAVE> .run filename.pro
```

where `filename.pro` is the name of the file which has the `WAVE` code. Make sure you are running `WAVE` from the directory in which the file is saved! Once that is done, you can run the program by typing the name of the procedure:

```
  WAVE> simp_plot
```

The output might look like:



There are many options to the `plot` command, and many other useful commands. We look at those options after some details on compiling and running `WAVE` code.

8.1. Compiling and executing

WAVE is rather like FORTRAN in that you need to first compile your code, then run it. However, WAVE can also simply interpret your code (compile a line, run it, compile a line, run it, etc.). For most applications, writing a short program and compiling it is the best way to go.

Compiling: .run

When you type commands at the WAVE prompt, these commands are interpreted and executed immediately. When you "run" an external program, for example .run progname.pro, those commands are also interpreted and executed immediately.

However, if your external program is actually a procedure, starting with a PRO declaration, then the procedure is compiled but **not** executed. The procedure becomes another new command that WAVE will then understand. To execute it, you must call the procedure by typing its name, just as you type the plot command name and so on. Your file can include multiple procedures and functions, all of which will be compiled when you .run the file, and then available as new commands within WAVE. You need to re-compile them very time you re-start WAVE but you can do this with your wavestartup program (see section 8 8.6).

Suppose you have a procedure that looks like:

```
PRO myplot,x,y
; procedure to plot stuff my way
plot,x,y,title='myplot'
END
```

which is saved in a file called myplot.pro. You can compile this and execute it as follows:

```
WAVE> .run myplot
WAVE> myplot,x,y
WAVE> myplot,a,b
WAVE> myplot,c,d
```

When errors occur: retall

If WAVE finds an error when compiling or executing your code, it is useful to use the command retall. This tells WAVE to give up and start again.

Command-line editing: the cursor keys

You can use the up and down arrow keys to re-enter previous commands, and the left and right arrow keys to edit those commands.

Comments and long lines: ; and $

Comments in WAVE programs are prefixed with the semicolon ; which can occur anywhere on a line (so anything following a ; is ignored). Long commands can be split using the $. Here's an example:

```
  ; this line is an example of a comment
  plot,x,y,title='Hubble expansion',xtitle='Distance', $
    ytitle='Expansion'      ; extension of previous command
```

Stopping wave temporarily: Ctrl-Z and fg

It can be very useful to stop WAVE temporarily, so you can type a few Unix commands (for example, to print a file, or display it with ghostscript). Use Ctrl-Z to temporarily get back to Unix, and the Unix command fg to get back to WAVE.

save,/all

If you're in the middle of a complicated WAVE session, and you want to save everything and return to it later, you can use the save,/all command. For example:

```
WAVE> save,/all,filename='mysave.dat'   ; save all variables
-- User exits and then enters a new PV-WAVE session --
WAVE> restore,'mysave.dat'              ; restore all saved variables
```

8.2. Printing with WAVE

WAVE normally draws on the screen. To print, it is first necessary to tell WAVE to draw to a Postscript file. Once the plotting is done, the Postscript device is closed, and the file can be printed.

Note that it is essential that you first plot your stuff on-screen. Then tell WAVE to send future drawing commands to a Postscript file, then tell WAVE to close that file and return to drawing on-screen. Here's an example:

```
  WAVE> plot,x,y                ; it is ESSENTIAL that you first
                                ; plot it on-screen!!
```

```
WAVE> postscript,'myplot.ps'   ; redirect output to file 'myplot.ps'
WAVE> plot,x,y                 ; normally this would draw on-screen
WAVE> postscript,/close        ; close the Postscript file
```

A reminder: **IT IS ESSENTIAL THAT YOU FIRST PLOT ON-SCREEN!**

The `postscript` command has some other options to change the size of the plot and so on. Help on these is available by typing:

```
WAVE> postscript,/help
```

Once your plot has been output to the file, `myplot.ps`, it can be printed or viewed with the usual Unix commands. Return to Unix with `Ctrl-Z`, and then:

```
% gs myplot.ps
```

or from within `WAVE`,

```
WAVE> spawn,'gs myplot.ps'
```

If satisfactory, it can be printed:

```
WAVE> spawn,'qpr myplot.ps'
```

8.3. Reading data

Data is normally read from text files which are formatted into columns. This is easily done with the `dc_read_free` command. To read three columns, for example, try:

```
fname='myfile.dat''
status=dc_read_free(fname,x,y,dy,/column)
ploterr,x,y,dy
```

Simply add more variable names if there are more columns. Note that `WAVE` likes to use vectors and matrices. In this example, `x`, `y` and `dy` will be vectors. `WAVE` knows that when you say `ploterr,x,y,dy` you mean plot each of the elements in the vectors. You can add, subtract, multiply vectors and matrices with simple commands, for example `z=x+y`, where `z` will be a vector result.

Reading image data

Various image formats can be read. The most efficient is raw 8-bit binary data, one byte per pixel. For example, to read a 256×256 image and display it in greyscale:

```
  fname='myfile.raw''
  image=bytarr(256,256)
  openr,1,fname
  readu,1,image
  close,1
  tvscl,image
```

Standard image formats can also be read, for example GIF files:

```
  test=image_read('test.gif')
  img=test('pixels')
  tvscl,img
```

`test` will be an associative array with all sorts of good stuff; for more info on `test`, try:

```
  WAVE> info,test,/full
```

8.4. 2D plots with WAVE

The `plot` command and its variants have many many options. For convenience, a few are detailed here; more info can be found with the `WAVE help` command.

Choosing your axes

`plot` can be used with one parameter (e.g. `plot,y`) or with both coordinates (`plot,x,y`). The `plot` command can be replaced with the log/lin variations `plot_io` (log for y axis), `plot_oi` (log for x axis) and `plot_oo` (both log).

Adding titles

These options are used to specify the title and axes labels for the plot. For example,

```
WAVE> plot,x,y,title='Hubble expansion',xtitle='Distance',ytitle='Expansion'
```

Modifying the range of your axes

Normally `WAVE` will pick nice axis limits, but if you want more control, you can choose specific ranges, for example:

```
  WAVE> plot,x,y,xrange=[-1.5,3],xstyle=1
```

Here the `xstyle=1` option tells `WAVE` to use exactly `[-1.5,3]` rather than rounding it of to `[-2,3]`.

Plotting one set of data over another

To plot another set of data on the same set of axes as a previous set, use the "over-plot" command:

```
  WAVE> plot,x,y ; first data set
  WAVE> oplot,x,z ; second data set
```

Setting the symbol for your plot

This specifies the symbol. Some common examples are:

0	No symbol – points connected by lines
1	+
2	*
3	. (dot)
4	◇
5	△
6	□
7	×
8	user-defined

Use negative values of `psym` to connect points with straight lines. The `symsize` option can be used to get larger or smaller symbols, for example `plot,x,y,psym=2,symsize=3`.

Setting the colour for your plot

Colour is specified with the `col` option. Colours are initially shades of grey, unless you use the `tek_color` command (see sec. 8 8.6. Alternatively, the `loadct,n` "load colour table" command can be used to set colours up. Assuming you've given the `tek_color` command, then you can use simple red/green/blue colours with something like the code below.

```
  WAVE> plot,x,y,/nodata ; plot using standard colours
  WAVE> oplot,x,y,col=2 ; replot just the curve, with a new colour
```

The first few standard colours are:

0	white
1	red
2	green
3	blue
4	cyan
5	magenta
6	yellow
7	orange

Note that white on-screen will be printed as black. You can see the full colour table with the command `color_palette`.

Plotting error bars

These are the same as `plot` and `oplot` but plot error bars on each point. For example,

```
  WAVE> plot,x,y,title='Plot with errors',/nodata
  WAVE> oploterr,x,y,dy
```

Note first of all the `plot,...,/nodata`. This plots the axes and title, but no data points. Then `oploterr` then plots (x,y,dy) data sets where the third parameter gives the size of the error bars.

8.5. 3D Plots with WAVE

The sequence of commands below is just to give you a taste of what `WAVE` can do.

```
  WAVE> a=fltarr(100,100)
  WAVE> for i = 0.0,99 do $
   - for j = 0.0,99 do $
   -   a(i,j) = sin(i/5 + j/5)
```

So far, all we have done is calculate an array of numbers obeying a sine rule. To see the result, use the command

```
  WAVE> shade_surf, a
```

to produce a shaded surface. The command

```
  WAVE> tvscl, a
```

will show you a view of your map from above on the same screen as your shaded surface. This will appear as a small box in the lower left-hand corner.

Alternatively, `WAVE` allows you to see contours of your 3D map. Use the command

```
  WAVE> contour, a
```

to see this.

8.6. Environment

`WAVE` needs to know how to find some things, and it uses certain environment variables to help. In your `.profile`, `.cshrc` or other initialisation file, put in:

```
export MY_WAVE_DIR=~/wave
export WAVE_STARTUP=~/wave/wavestartup
```

or, if you use a version of the Bourne or C-shells,

```
setenv MY_WAVE_DIR ~/wave
setenv WAVE_STARTUP ~/wave/wavestartup
```

You should also have a `WAVE` directory `~/wave` and a `WAVE` startup script. The latter can be many things, but here's an example that helps avoid many curly problems:

```
COMMON colornames, black, red, yellow, green, cyan, magenta, blue

device,retain=2 ; set backing store on for all windows
emacs_keys ; attempt to get emacs-like key action

xsize=40
```

```
ysize=40
window,xsize=xsize,ysize=ysize,colors=128,retain=2
wdelete

; set first 16 colours to simple things like black, red, green, ...
tek_color
```

8.7. Quick Reference Section

Getting Started

`wave`	gets you into `WAVE`
`quit`	gets you out of `WAVE`
`help`	hypertext help file
`help,/doc`	full help document
`demo`	runs demonstration

Plotting: 2D

`plot,x,y`	plots 2 columns
`oplot,x,y`	plots second data set over first
`ploterr`	plots with error bars
`oploterr`	over-plots with error bars

Options:

`/nodata`	plots axes, but no data points
`col`	select colour
`psym`	select symbol
`symsize`	sets symbol size
`plot_io`	plot with log for y-axis
`plot_oi`	plot with log for x-axis
`plot_oo`	plot with log for both axes
`title=''`	specify title of plot
`xtitle=''`	specify title of x-axis
`ytitle=''`	specify title of y-axis
`xrange=[]`	sets range of x-axis
`yrange=[]`	sets range of y-axis
`xstyle=1`	ensures `WAVE` doesn't round-off range
`ystyle=1`	

Plotting: 3D

`shade_surf`	plots a shaded 3D surface
`tvscl`	plots a view from above
`contour`	plots a series of contour lines

Working inside WAVE

`fname='mydata.dat'`	defines file name
`status=dc_read_free(fname,x,y,/column)`	reads data from a file in column format
`.run file.pro`	compiles a procedure
`postscript,file.ps`	redirects output to postscript file
`postscript,/close`	closes postscript file
`postscript,/help`	more postscript options
`spawn`	run Unix programs from inside `WAVE`
`spawn,'gs file.ps'`	view postscript file with ghostscript
`spawn,'qpr file.ps'`	print from within `WAVE`
`retall`	tells `WAVE` to give up and start again
`$`	splits a long command
`;`	indicates a comment
`Ctrl-Z`	temporarily stops `WAVE`
`fg`	gets back to `WAVE` after a temporary stop

9. FPLOT: BASIC PLOTTING AND CURVE FITTING

There are a host of relatively simple plotting packages (e.g. **fplot**, **gnuplot**, **xplot**, ...) now available on the system. All are reasonably self explanatory with on-line help facilities, however, for the purposes of this manual we will concentrate on one of the most comprehensive packages, **fplot**, which has been used for some time for the production of publication quality graphs. The program **fplot** is a general purpose plotting and fitting program which was originally written here at the School of Physics by D. Jamieson and further developed by R. Brown.

9.1. From data to graph

To enter **fplot** type

```
% fp
```

You will get an Xterm fplot window with the prompt

```
Fp >
```

A session will usually involve reading in data from a file, plotting to the screen, adding captions and printing a hard copy. Here is a typical **fplot** session.

```
Fp > make/2 test.dat
```

→ Read in data from file **test.dat** in x,y format (the filename can be anything).

```
Fp > dr/d d
```

→ Disable the drawing of data points (*dr/e d* enables drawing).

```
Fp > spline
```

→ Cubic spline interpolation (draws a nice smooth line through data – if too many points use **fun 9**).

```
Fp > dr/x
```

→ Draw to the screen only using default scales.

```
Fp > cap
```

→ Change the title of the plot.

```
Fp > cap/x
```

→ Change the x-axis caption (for y–axis use cap/y).

```
Fp > dr/xp
```

→ Draw to the screen and to the *postscript* file.

```
Fp > close/p
```

→ Close the postscript file so it can be printed.

```
Fp > exit
```

→ Exit from the **fplot** session (if the postscript file has not been closed it will be closed by exiting).

9.2. Printing Fplot graphs

The graph has been saved in *postscript* format to a file called **Fplot.ps**. Before printing to a laser writer you must make sure that everything is OK (so that paper and printing time is not wasted) by using the **ghostscript** or **ghostview** previewers. This will display your plot to the screen as it will appear in print. To view your plot using **ghostscript** type

```
% gs Fplot.ps
```

once satisfied simply kill the **ghostscript** screen by typing *alt-f4*. To print the plot contained in the file **Fplot.ps** you use the **qpr** command:

```
% qpr Fplot.ps
```

9.3. Miscellaneous commands

help → Displays a menu of help topics.

dr/# x → Prompt for user specified x and y ranges (# = x, p or xp).

dr/r → Redraw. Once you have drawn one graph to the devices specified (e.g. using **dr/xp x**) you can **make/2** another file and draw this data to the same plot using the redraw command.

label → Insert text (a label) – coordinates, orientation and character size will be prompted for.

fun → Displays a list of fitting functions (see the commands **np**, **par** and **mask**).

log/x → Logs the x–axis (**log/y** for y–axis).

def/# → shows the default settings of various parameters (# = x,p or xp) governing the output which can be modified (see template macro section).

dr/j f → multiply size of data points by factor *f*.

dr/n → changes the shape of the data points to shape *n*, where n is an integer between 1 and 8. For instance, *dr/5* will plot the data points as stars.

9.4. Macros

It often happens that you end up repeating a particular sequence of **fplot** commands each time you start the program. A lot of time can be saved by simply listing the commands in a file (a macro) as you would type them in a **fplot** session and reading that file in the **fplot** session. For example if a file called **macro.fp** has the following lines:

```
make/2 test.dat
cap
this is the title of my graph
cap/x
x-axis caption
```

```
cap/y
y-axis caption
dr/x
```

then the command

```
Fp > @macro
```

will read in the **fplot** commands from the file **macro.fp** (the extension **.fp** signifies that the file is a **fplot** macro).

9.5. Sample macro

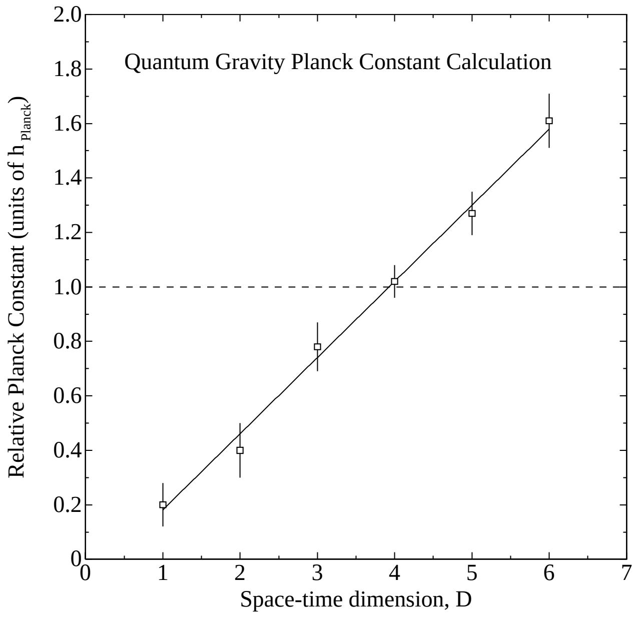
Here is a sample macro which plots some data from a fourth year student's (highly fictitious) quantum gravity calculation. This example incorporates a number of Fplot's commonly used features. Of course, it is difficult to include all the features of this plotting program, however, online help is available.

Consider a file **planck.dat** in (x, y, σ_y) format with the following data to which we want to fit a straight line, add captions and plot for publication.

```
Data from file: planck.dat

1    0.20    0.08
2    0.40    0.10
3    0.78    0.09
4    1.02    0.06
5    1.27    0.08
6    1.61    0.10
```

The final result from the macro described below appears overleaf.



The following macro file, broken into sections for ease of explanation, produced the plot shown.

```
make/3 planck.dat
def/x as = 1
def/p as=1 xo=4 yo=14 sz=12 wa=0 ds=planck.ps
wi 2.7
chr n
```

The data has been read with the errors on the y values in the third column ($make/3$). The aspect ratio for the plot to the screen has been set to 1 (*def/x as=1*) – it is usually easier for journals (e.g. Physical Review) to print square graphs. The postscript file has been designated **planck.ps** (*ds=planck.ps*) and the offsets (xo and yo), size (sz and orientation (wa) have been set so that the final printout will look reasonable. The line-width for the postscript file has been set (*wi 2.7*) and the native fonts of the printer accessed (*chr n*).

```
cap

cap/x
Space-time dimension, D
cap/y
Relative Planck Constant (units of h^D Planck^+)


dr/m
```

```
0
1
```

The caption has been set blank (the default caption is the filename and as the caption command is unpredictable we will set it blank and later use the label command instead). Captions for the x and y axes are set. The control characters which control subscripts and superscripts are: ^D *subscript text* ^+ and ^U *superscript text* ^−. Greek letters can be written as ^G *greek text* ^R. The characters ^R reset a given mode. The number of digits after the decimal place for x and y values have been set to 0 and 1 respectively (*dr/m*) – this is useful if the data includes integers.

```
fun 2
npar 3
par 2
0
ma 2
0
fit
dr/xp x
0 7
0 2.0
```

In fitting the data to a straight line the polynomial function is chosen (*fun 2*) – a list of possible functions can be obtained by typing fun. The polynomial form is $y = y_o + k_1(x + a) + k_2(x + a)^2 + k_3(x + a)^3 + ...$ The number of fitting parameters is set to three (*npar 3*) and the offset parameter a (parameter number 2) is set to zero and removed from the fit by $masking$ parameter 2 to zero. The straight line fit is carried out (fit). The data is plotted to screen and the postscript file with user defined ranges (*dr/xp x*) of $x \in [0, 7]$ and $y \in [0.0, 2.0]$.

```
lab
0.5 1.8
0
1.0
Quantum Gravity Planck Constant Calculation
```

A label has been defined at the position $(x, y) = (0.5, 1.8)$ with orientation of 0 degrees and text size 1.0.

```
def/xp lt=1
arc 1
0 1.0
7 1.0
close/p
```

The default line-type (*lt*) has been changed to a dashed line for the screen and postscript file (*def/xp lt=1*). A straight line has been drawn (*arc l*) starting at $(x,y) = (0,1.0)$ and ending at $(x,y) = (7,1.0)$ (i.e. a horizontal dashed line). The postscript file is then closed (*close/p*).

9.6. Multiple graphs

The macro below shows how to change the size of a graph and its position on the page. Using a line such as,

```
def/p xo=1, yo=10.5, as=1, sz=6, wa=0 ds=converg4.ps
```

you can set the x and y offsets, the aspect ratio (as), the size (sz), and so on. The *ds* command determines what the output postscript file will be called. In the case shown below, these commands are used in combination to place four different graphs on a single page. The resulting page is shown in the chapter on LaTeX, figure 2.

```
def/p xo=1, yo=10.5, as=1, sz=6, wa=0 ds=converg4.ps
  make/2 mconv1234.dat
  cap
MFDM - Gaussian Graded Core
  cap/x
^G a ^R-value
  cap/y
Calculated U-value

  dr/3
  dr/xp x

1.5 4.0


def/p xo=1, yo=18.5, as=1, sz=6, wa=0 ds=converg4.ps
  make/2 fconv8.dat
  cap
FDM - Gaussian Graded Core
  cap/x
L(micron)
  cap/y
Calculated U-value
  dr/3
  dr/xp x
```

```
1.5 4.0


def/p xo=11, yo=10.5, as=1, sz=6, wa=0 ds=converg4.ps
  make/2 mconv5.dat
cap
MFDM - Step Profile Square Core
  cap/x
^G a^R-value
  cap/y
Calculated U-value
  dr/xp x

1.5 4.0


def/p xo=11, yo=18.5, as=1, sz=6, wa=0 ds=conver4.ps
make/2 fconv3_4.dat
cap
FDM - Step Profile Square Core
  cap/x
L(micron)
  cap/y
Calculated U-value
  dr/xp x

1.5 4.0


close/p
```

10. LATEX AND REPORT WRITING

LaTeX (pronounced "lay-tech") has become the standard wordprocessing package in physics for writing reports and papers. Using LaTeX requires learning a relatively simple programming language, the functions of which give the package its strongest asset – the fact that mathematical typesetting will come out correct (modulo programming errors) without the author having to waste time "beautifying" expressions. Many journals will now accept electronic submission of manuscripts in LaTeX format. It has also become the standard to write up one's fourth year report in LaTeX.

The following section introduces you to some of the main features of LaTeX, and takes you step-by-step through the fourth-year template file. In conjunction with this you should read the "Essential LaTeX" guide, which can be accessed from the Part IV home page. This is available in the form of an unprocessed LaTeX file.Click on the link, and save the file to an appropriate directory, making sure you use the extension ".tex". You will then need to LaTeX it by typing

```
% latex essential.tex
```

You can now view it using the xdvi utility by typing

```
% xdvi essential
```

(See section the next section for the **xdvi** command.)

For more advanced LaTeX use, see the "LaTeX2e" user-guide, also available on the part IV homepage, or have a look at the "LaTeX User's Guide & Reference Manual", by L. Lamport (copy available in the Baker Lab). There is also a link to a TeX FAQ (a list of frequently asked questions and answers about TeX) on the part IV homepage - some of the questions relate to common problems encountered when using LaTeX.

10.1. Creating, compiling, previewing and printing LaTeX documents

Consider the following file **first.tex** which is an example of a simple LaTeX document with some mathematics.

```
\documentstyle[12pt]{article}
\begin{document}
```

```
This is the first paragraph so it appears automatically indented. The
amount of indentation can be controlled by a definition statement at
the beginning of the file.

Skipping a line automatically starts a new paragraph.\\
But two back-slashes gives a carriage return. You can make characters
{\large larger} or put them in {\bf bold face} or {\it italics}.

Mathematical expressions can be inserted in a line such as $E=m c^2$
by enclosing the expression in \$ signs. Note the backslash negated the
special function of the \$ sign.

An equation on its own can be defined as

\begin{equation}
\left[ -{\hbar^2\over 2 m}{\bf \laplace}^2 + V({\bf r})\right]
\psi({\bf r}) = E\psi({\bf r})
\end{equation}
\end{document}
```

To compile the file **first.tex** under LaTeX, use the **latex** command (which invokes the latest version of LaTeX)

```
% latex first
```

If all goes well you can now preview the document on the screen using the **xdvi** command

```
% xdvi first
```

The **.exrc** file has defined certain function keys for LaTeX within a *vi* session so that you need not leave the editor after making changes. For example **F11** will save and latex the file, and **F12** will bring up the xdvi previewer.

To create a postscript file **first.ps** for printout use the **dvips** command

```
% dvips -o first.ps first -pp 5-17
```

The "-pp 5-17" flag causes the **dvips** command to prepare only pages 5 to 17 in postscript form, so that you can print out only the pages you require. If you leave out this flag all pages are automatically converted to postscript. For very long documents, you may want to use the **dvi2ps** command which places two pages to a side.

The postscript file can be checked (to save time and paper) using either **ghostscript** or the more sophisticated **ghostview** commands. The **dvips** command has many useful options which are summarized in the manual page from the **man** command.

The printout of the file **first.ps** should look like this:

This is the first paragraph so it appears automatically indented. The amount of indentation can be controlled by a definition statement at the beginning of the file.

Skipping a line automatically starts a new paragraph.
But two back-slashes gives a carriage return. You can make characters larger or put them in **bold face** or *italics* or a combination.

Mathematical expressions can be inserted in a line such as $E = mc^2$ by enclosing the expression in \$ signs. Note the backslash negated the special function of the \$ sign.

An equation on its own can be defined as

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) \right] \psi(\mathbf{r}) = E\psi(\mathbf{r}) \tag{1}$$

The UNIX **ispell** command is useful to check the LaTeX file before the final compilation and printing.

```
% ispell -t first.tex
```

(The **-t** option tells the spell checker to take into account LaTeX commands)

10.2. Template fourth year report

Some typical uses of LaTeX, particularly in the context of writing up a fourth year report will appear in the template LaTeX file to follow. The template file that follows can be found in **/home/4thyear/report.tex**.

```
% Template fourth year report LaTeX document.
%-------------------------------------------------------------
```

```
% define the document style (using revtex macros)
\documentstyle[psfig,preprint,aps,epsf,tighten,floats]{revtex}
\topmargin 0mm                          %top of page margin
\headheight 0pt                         %running head height
\headsep 0pt                            %head to text separation
\textheight 230mm                       %text height
\parindent 0em                          %paragraph indentation width
\parskip 4mm                            %paragraph width
\oddsidemargin 0mm                      %left  side of page margin width
\evensidemargin 0pt                     %right side of page margin width
\textwidth 150mm                        %width of text across the page
\def\thesection{\arabic{section}}       %defines arabic section numbering
\renewcommand{\thesubsection}{\thesection.\arabic{subsection}}
%
\begin{document}
%----------------------------------------------------------------
% Insert the postscript of the Melbourne University logo at the top
\begin{figure*}
\vspace*{30mm}
\centerline{\special{psfile=/home/4th_year/UMCrest97.eps vscale=80 hscale=80
hoffset=-40}}
\end{figure*}
\rule{160mm}{0.5mm}
%----------------------------------------------------------------
% Now start the title page
%
\begin{center}
{\bf\large Unification of General Relativity and Quantum Mechanics}
\end{center}
\begin{center}
{\large S.O. Clever}
\end{center}
\begin{center}
{Honours Report, 1998}
\end{center}
\vspace*{10mm}
\begin{center}
Abstract
\end{center}

A new quantization of space-time geometry is developed which, for
the first time, unifies general relativity and quantum mechanics.
An amusing corollary on renormalization of the manifold summation
provides a detailed temporal evaluation of the Hubble constant and
the ultimate fate of the universe.
```

```
\begin{center}
Supervisor: Dr. A.R.C. Forms
\end{center}
\vspace*{10mm}

I authorize the Chairman of the School of Physics to make or have made a
copy of this report for supply to any person judged to have an
acceptable reason for access to the information, i.e., for research,
study or instruction.

\vspace*{10mm}
\begin{flushright}
Signature....................................
\end{flushright}
\newpage

\section{Introduction}

I will try to cram in most of the bits and pieces needed for a typical
report, namely

\begin{itemize}
\item sections
\item equations
\item references
\item tables
\item figures
\end{itemize}

Well, that covered itemization already (see also enumerate).

\section{Sections}

As you can see sections are pretty easy and self explanatory. The new
section is automatically numbered correctly. Subsections are just as easy
({\bf subsection} command).
A table of contents is constructed by the tableofcontents command.

\section{Equations}

This is one of the most important (sometimes fun, quite often
extremely frustrating) aspects of writing a report in physics.

Using the {\bf equation} environment, one line equations will be numbered
```

```
\begin{equation}
Z = \sum_{n=0}^{\infty} e^{-\beta E_n}.
\end{equation}

Bra and kets are easy. When you are including equations in a
sentence, such as

\begin{equation}
H|\psi\rangle = E |\psi\rangle,
\end{equation}

don't forget to use punctuation.

The most common problem to overcome is when equations get large and run
over more than one line. How to break the equation is an art form and
best left to the author. Use the {\bf eqnarray} environment and {\bf nonumber}
to turn off the numbering of every line. Here are some examples.

\begin{eqnarray}
\beta_{n}^{2}&=& n c_{2}N_{p}
+ {1\over 2}\,n\,(n-1)\left[{c_{2}c_{4} - c_{3}^{2}\over c_{2}^{2}}\right]
\nonumber\\
\nonumber\\
&+& {1\over 6}\,n\,(n-1)\,(n-2) \left[{-12 c_{3}^4 + 21 c_{2} c_{3}^2 c_{4}
- 4 c_{2}^2 c_{4}^2 - 6 c_{2}^2 c_{3} c_{5} + c_{2}^3 c_{6}
\over 2 c_{2}^5}\right] {1\over N_{p}}
+ \dots
\end{eqnarray}

The equation is split by aligning the = sign and the + sign with the
\& characters together with nonumbered carriage returns. Two returns
were put in to space the lines out a bit. Note how the left and right
brackets automatically size themselves for the content (braces ``{\bf \{}''
or parentheses ``{\bf (}'' are used in the same way). If you need to
break a line with matching {\bf $\backslash$left[} and {\bf
$\backslash$right]} brackets (or parentheses or braces) you can
instead use independent brackets tailored to the right size using
commands such as {\bf bigl[} for a big left bracket
{\bf Bigr$\backslash$\}} for an even bigger right brace, etc.

Normal spaces are ignored in math mode, but small spaces can be
introduced, e.g. by using backslash-comma and larger spaces can be put
in using {\bf quad}.
```

```
Sometimes, no matter how hard you try it just can’t be done elegantly
and you have to cut your losses (or make some drastic notational
changes):

\begin{eqnarray}
M_V(y) &=& {4\over {3\,y}}
- {(2y^2+3)^2(336y\sqrt{\Delta(y)}-1129y^4-1998y^2-432)
\over 36y^2\sqrt{\Delta(y)}(2y^6+114y^2-93)}\nonumber\\ \nonumber\\
&+& {2(2y^2+3)(90y^8-2426y^6-713y^4-7683y^2-288)(12y\sqrt{\Delta(y)}+3y^4
-32y^2-12)\over 27y^3(2y^6+114y^2-93)^2}
\end{eqnarray}

\section{References}
\label{refs}
References are handled using the {\bf cite} command. Let’s add a
reference now\cite{refname1} to the LaTeX manual. At the end of
this document the bibliography will be defined and next to {\bf
refname1} will appear the appropriate reference. Wherever {\bf refname1}
is cited in the text from now on it will be given the correct numeral,
in this case number 1 because it is the first reference of this
document. The next reference\cite{nextone} will have the number 2 and
so on. At the end of this sample document the bibliography will be
listed. To refer to a section of your document, use the {\bf ref} command.
First give the section you wish to refer to a {\bf label}. The section
you are now reading has been given the label ‘‘refs’’ by including
a line just below the section header. Now whenever you wish to refer
to this section, just type

\ref{refs}

and LaTeX will automatically
print out the correct section number, taking into account any
renumbering which occurs due to the addition of new sections
and so on. Whenever you define or change references you have
to compile the file twice to get cross referencing correct.

\section{Tables}

Here is an example of how to build a table.

\mediumtext
\begin{table}[htbp]
\caption{First order Lanczos cluster expansion
approximation, $E_0(\lambda)$, for the ground state of the
anharmonic oscillator, $H = -{1\over 2}\,{d^2\over dx^2} + {1\over
2}\,x^2 + \lambda\,x^4$.}
\vspace*{10mm}
\label{AHO-table}
\begin{tabular}{cccccc}
& ${\rm log}_{10}\lambda$ & Variational & $E_0(\lambda)$ & Exact &\\
\tableline
& $0$ &  0.8125 & 0.8037 & 0.8038 &\\
& $1$ &  1.5313 & 1.5040 & 1.5050 &\\
& $2$ &  3.1924 & 3.1286 & 3.1314 &\\
& $3$ &  6.8280 & 6.6877 & 6.6942 &\\
& $4$ &  14.6871 & 14.3838 & 14.3980 &\\
& $5$ &  31.6317 & 30.9775 & 31.0103 &\\
& $6$ &  68.1434 & 66.7338 & 67.0993 &\\
\end{tabular}
\end{table}

Note the definition of the number of columns using {\bf cccccc} and the
alignment once again using {\bf \&}.

\section{figures}

Taking the postscript figure example {\bf planck.ps} from the
Fplot section this can be inserted into the text in the following way.

\newpage
\begin{figure}[htp]
\special{psfile=planck.ps hscale=40 vscale=40 hoffset=80
voffset=-300 angle=0}
\caption{This is a completely fictitious graph}
\end{figure}
\vspace{7cm}

Note the {\bf newpage} command which was needed to put the figure
on the next page and the vertical spacing of 7 cm to accommodate
it.

\begin{thebibliography}
\small
\bibitem{refname1} LaTeX manual
\bibitem{nextone}J. Bloggs, Private Communication.
\end{thebibliography}

\end{document}
```

The LaTeX'ed template report appears as follows.

THE UNIVERSITY OF MELBOURNE
Australia

Unification of General Relativity and Quantum Mechanics

S.O. Clever

Honours Report, 1998

Abstract

A new quantization of space-time geometry is developed which, for the first time, unifies general relativity and quantum mechanics. An amusing corollary on renormalization of the manifold summation provides a detailed temporal evaluation of the Hubble constant and the ultimate fate of the universe.

Supervisor: Dr. A.R.C. Forms

Signature....................................

1. INTRODUCTION

I will try to cram in most of the bits and pieces needed for a typical report, namely

- sections
- equations
- references
- tables
- figures

Well, that covered itemization already (see enumerate).

2. SECTIONS

As you can see sections are pretty easy and self explanatory. The new section is automatically numbered correctly. Subsections are just as easy. A table of contents is constructed by the tableofcontents command.

3. EQUATIONS

This is one of the most important (sometimes fun, quite often extremely frustrating) aspects of writing a report in physics.

Using the **equation** environment, one line equations will be numbered

$$Z = \sum_{n=0}^{\infty} e^{-\beta E_n}. \tag{2}$$

Bra and kets are easy. When you are including equations in a sentence, such as

$$H|\psi\rangle = E|\psi\rangle, \tag{3}$$

don't forget to use punctuation.

The most common problem to overcome is when equations get large and run over more than one line. How to break the equation is an art form and best left to the author. Use the **eqnarray** environment and **nonumber** to turn off the numbering of every line. Here are some examples.

$$\begin{aligned}\beta_n^2 &= nc_2N_p + \frac{1}{2}n\,(n-1)\left[\frac{c_2c_4 - c_3^2}{c_2^2}\right] \\ &+ \frac{1}{6}n\,(n-1)\,(n-2)\left[\frac{-12c_3^4 + 21c_2c_3^2c_4 - 4c_2^2c_4^2 - 6c_2^2c_3c_5 + c_2^3c_6}{2c_2^5}\right]\frac{1}{N_p} + \ldots \qquad (4)\end{aligned}$$

The equation is split by aligning the = sign and the + sign with the & characters together with nonumbered carriage returns. Two returns were put in to space the lines out a bit. Note how the left and right brackets automatically size themselves for the content (braces "**{**" or parentheses "**(**" are used in the same way). If you need to break a line with matching **left**[and **right**] brackets (or parentheses or braces) you can instead use independent brackets tailored to the right size using commands such as **bigl**[for a big left bracket **Bigr****}** for an even bigger right brace, etc.

Normal spaces are ignored in math mode, but small spaces can be introduced, e.g. by using backslash-comma and larger spaces can be put in using **quad**.

Sometimes, no matter how hard you try it just can't be done elegantly and you have to cut your losses (or make some drastic notational changes):

$$\begin{aligned}M_V(y) &= \frac{4}{3\,y} - \frac{(2y^2+3)^2(336y\sqrt{\Delta(y)} - 1129y^4 - 1998y^2 - 432)}{36y^2\sqrt{\Delta(y)}(2y^6+114y^2-93)} \\ &+ \frac{2(2y^2+3)(90y^8 - 2426y^6 - 713y^4 - 7683y^2 - 288)(12y\sqrt{\Delta(y)} + 3y^4 - 32y^2 - 12)}{27y^3(2y^6+114y^2-93)^2} \qquad (5)\end{aligned}$$

4. REFERENCES

References are handled using the **cite** command. Let's add a reference now[1] to the LaTeX manual. At the end of this document the bibliography will be defined and next to **refname1** will appear the appropriate reference. Wherever **refname1** is cited in the text from now on it will be given the correct numeral, in this case number 1 because it is the first reference of this document. The next reference[2] will have the number 2 and so on. At the end of this sample document the bibliography will be listed.
To refer to a particular section of your document by number, use the **ref** command. First give the section you wish to refer to a **label**. The section you are now reading has been given the label "refs" by including a line just below the section header. Now whenever you wish to refer to this section, just type

```
\ref{refs}
```

and LaTeX will automatically print out the correct section number, taking into account any renumbering which occurs due to the addition of new sections and so on. Whenever you define or change references you have to compile the file twice to get cross referencing correct.

5. TABLES

Here is an example of how to build a table.

TABLE I. First order Lanczos cluster expansion approximation, $E_0(\lambda)$, for the ground state of the anharmonic oscillator, $H = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}x^2 + \lambda x^4$.

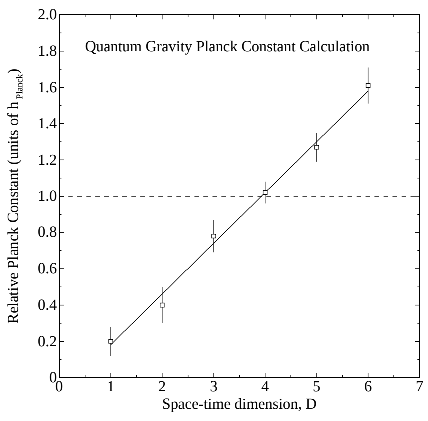
$\log_{10}\lambda$	Variational	$E_0(\lambda)$	Exact
0	0.8125	0.8037	0.8038
1	1.5313	1.5040	1.5050
2	3.1924	3.1286	3.1314
3	6.8280	6.6877	6.6942
4	14.6871	14.3838	14.3980
5	31.6317	30.9775	31.0103
6	68.1434	66.7338	67.0993

Note the definition of the number of columns using **cccccc** and the alignment once again using **&**.

6. FIGURES

Taking the postscript figure example **planck.ps** from the Fplot section this can be inserted into the text in the following way.

FIG. 1. This is a completely fictitious graph



Note the **newpage** command which was needed to put the figure on the next page and the vertical spacing of 7 cm to accommodate it.

REFERENCES

[1] LaTeX manual

[2] J. Bloggs, Private communication

10.3. More on including figures

Inserting figures and graphs can be one of the more frustrating aspects of using LaTeX . Don't expect to always get the results you want right away. If you persevere, and check the manual or ask for help when necessary, you should always be able to achieve decent results.

Note that xdvi will not show some figures - you have to use dvips and then ghostview the document to see them. In particular, the Melbourne University crest on the title page of this template file will give error messages when you use xdvi to view the document - don't worry about this, as the logo will come up fine when once you have converted the file to postscript using the dvips command.

Since it is not always possible to place an inserted figure directly next to the appropriate text, LaTeX needs to make decisions regarding where the most reasonable position for each figure is. This is called "floating" the figures. LaTeX has a hierarchy of standard rules for determining where to insert floating objects such as figures.

Varying the settings in brackets directly after the **beginfigure** command (in the example, [htb]) is one way of altering the placement of figures. In this case we have told LaTeX that we would like the figure to be printed (h)ere if possible, or at the (t)op or the (b)ottom of a page, if this proves to be impractical.

These rules should usually ensure that the figure is placed somewhere reasonable, however if LaTeX is not doing what you would like with your figures, check these rules in the manual and see if you can work out why it's doing what it's doing - once you know this it should be possible to rectify the problem.

The **special** command should work for most figures if they are in post-script format. If this command appears to be behaving strangely, or fails to work at all, you can try the **psfig** command. This command requires the inclusion of the psfig revtex package in your **documentstyle** command at the start, ie

```
\documentstyle[psfig,preprint,aps,epsf,tighten,floats]{revtex}
```

Psfig is a revtex package or "macro" designed to help in the importation of figures and other postscript files. If it's not already there, you simply need to type "psfig" somewhere between the square brackets in your **documentstyle** command. Now you can use the **psfig** command to insert your figures. This command requires the postscript file to be in a form known as "encapsulated postscript". A normal postscript file assumes the figure is meant to take up a whole page. When this file is then imported into your LaTeX document, there can be difficulties related to the fact that the boundary of the figure is in a sense undefined. An encapsulated file has some information somewhere in it which in effect defines a *bounding box* around the figure. This bounding box then makes it easier for applications such as LaTeX to deal with the figure. To convert your postscript files to encapsulated postscript is easy - at the UNIX prompt just type

```
ps2epsi infile.ps [outfile.epsi]
```

To insert an encapsulated figure, **converg4.epsi** (the four-to-a-page example from the Fplot section), into your document using the **psfig** command include the following (output shown overleaf):

```
\begin{figure}[htb]
\psfig{file=converg4.epsi,width=150mm,clip=0}
\caption{Figures Comparing Convergence for the Two Methods}
\label{converg4}
\end{figure}
```

This should work in most cases. If neither **psfig** nor **special** will work, check the LaTeX manual or ask someone for help.

Another feature of the **psfig** command which is sometimes useful is that the width setting is measured directly in millimeters, so you don't have to worry about the vscale and hscale settings. LaTeX uses the bounding box proportions and the width in millimeters to work out how to appropriately scale the figure.

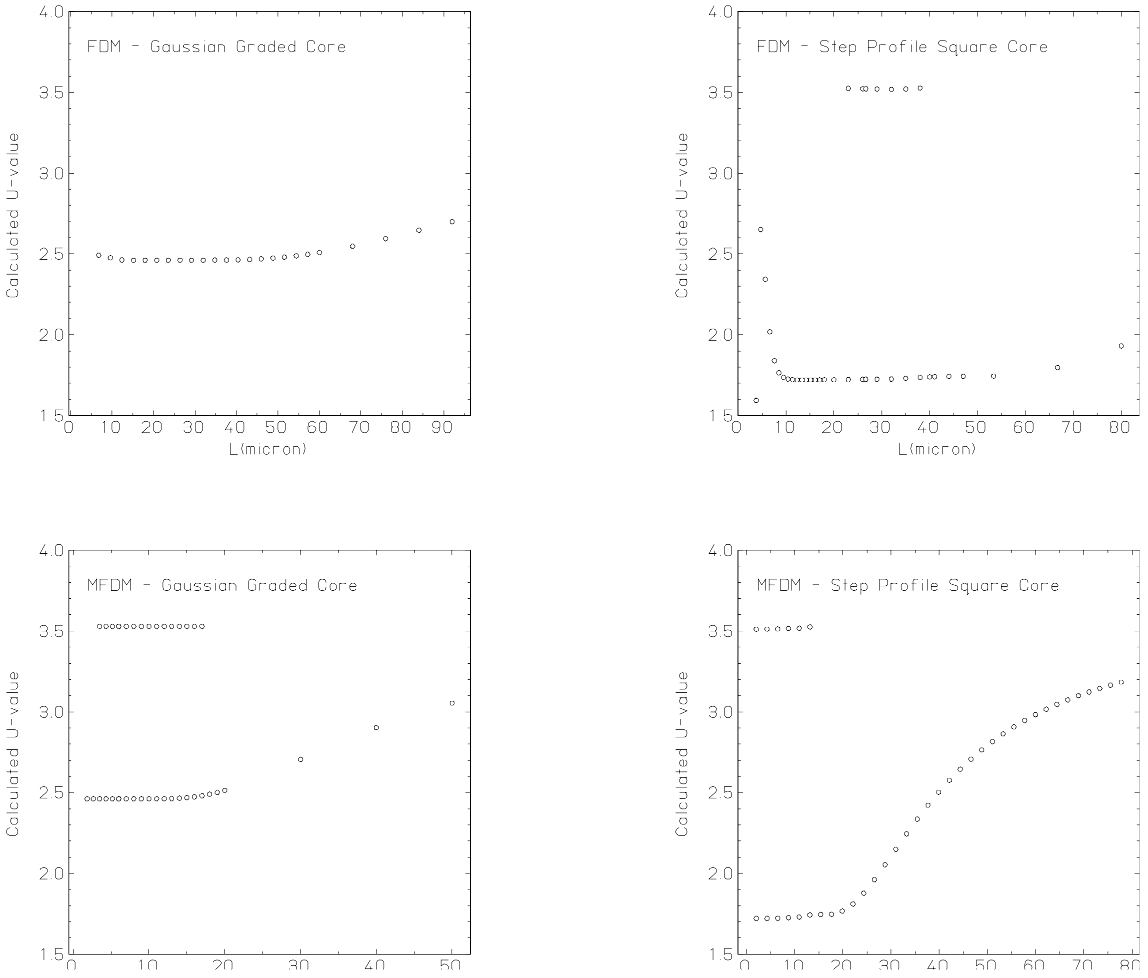


FIG. 2. Figures Comparing Convergence for the Two Methods

11. ALGEBRAIC PACKAGES: MATHEMATICA, MAPLE AND REDUCE

Somewhat of a revolution in physics has occurred with the advent of reliable, easy to use algebraic packages which can perform large mind numbing algebraic tasks nearly at the speed of light. For many it has become the norm to do analytic and numerical work using such packages. Extreme caution is urged in using the numerical aspects of these packages – for the simple reason that you are using a black box with no understanding of the method and its limitations. However, the analytic capabilities of these packages has a great deal of potential (watch out for bugs by cross-checking results with another package!) and it is therefore important to learn how to wield these new tools.

Mathematica and *Maple* do essentially the same analytical and numerical tasks. The main difference is that *Maple* has a user friendly X-window interface and an extensive online help facility. *Mathematica* has no online help, so to learn how to use it you need to refer to the reference book kept (and **never** removed) in the Baker Lab.

Reduce is much less general in its analytic and numerical capabilities. In fact the main reason for using *Reduce* is that it is a low level algebraic language that allows one to program virtually *any* algebraic task. This is particularly useful for non-commutating algebra, which the other more general packages are less useful for. Also, *Reduce* has an inbuilt capability for covariant vector algebra and Dirac-ology. *Reduce* runs in an X-window interface.

Due to license restrictions these programs are only available on mozart.

To start a session on *Mathematica*, *Maple* or *Reduce* the commands are **math**, **xmaple** and **xr** respectively.

12. ELECTRONIC MAIL AND THE INTERNET

Probably one of the most common uses of the computer system is for fast communication, both locally and worldwide. Communication is possible by electronic mail (almost instantaneous delivery), logins to remote computer systems, file transfers between computers and real time conversations between users on different computers.

12.1. Internet addresses

Each user of the computer system has an address for the worldwide internet enabling communication in all the above modes. This internet address is made up of the username of the person and the computer's node address as **username@node**. For example the internet address for the user **janet** on the computer **tauon** is **janet@tauon.ph.unimelb.edu.au**. More recently internet addresses on the computer system in the School of Physics can be specified without the computer name by using the person's first initial and surname, e.g. **l.hollenberg@physics.unimelb.edu.au**. An address on a computer in another country follows the same pattern, e.g. an address in Japan might be **john@sushi.raw.jp**. Once the internet address of a particular person is known it is easy to use the various communication modes.

12.2. Electronic mail

Electronic mail, or email as it is usually referred to, has become the standard method of communication between institutions. A letter sent by email reaches the recipient's account in a matter of seconds – regardless of where that computer is actually situated on the globe.

You can compose, send, and read mail using utilities such as **mail**, **pine**, and **elm**.

The use of **mail** is described below. However personally I find the **pine** mail-reading utility more convenient to use, as it is menu-driven. To use pine to read your mail simply type

```
pine
```

You can now select what you want to do from the main menu - the rest is up to you!

Instead of using the pine mail-reader for your email, you can use the standard unix mail feature if you wish. To mail a letter to John in Japan from Janet's account is as easy as the following (N.B. On Digital Alpha Stations (i.e. **tauon**) the same command is **Mail**):

```
% mail john@sushi.raw.jp
Subject: Your latest research...

Dear John,

I think your latest paper has some serious flaws.

Yours sincerely,

Janet.
^D
%
```

The mailer prompts for a subject line (not compulsory) and then the message is entered with returns at the end of lines. Finally, to send the message type **control d**. To abort the message type **control c** (before using **control d** of course).

You can send a file containing the message – this useful for sending messages which may be difficult to write interactively or data files etc. The format to send the file **letter.tex** to John is (assuming **letter.tex** is in the working directory):

```
% mail john@sushi.raw.jp < letter.tex
```

There are various useful options with the mail command. For example, to check that the mail actually was sent to the recipient's account the **-v** option will printout the progress of the message across the internet.

For Janet to check her mailbox for new mail the command is (see the command **elm** for a more sophisticated mailbox program)

```
% mail
Mail [5.2 UCB] [IBM AIX 3.2]  Type ? for help.
"/usr/spool/mail/janet": 1 message 1 new
>N  1 john@sushi.raw.jp    Fri Dec 23 19:24  11/309 "Who cares?"
 N  2 prize@nobel.vax.swe   Fri Dec 23 20:04  12/319 "rejection of prize??"
&
```

To view a message type the number of the message (digit to the right of **N** for new mail). A message can be written to a file, for example

```
& w 1 john_reply
```

Caution: if the message was being read for the first time it will be deleted from the mailbox when using the write command.

Type **q** to quit from the mail viewer and save the read messages to the mailbox. To view the contents of the mailbox the command is

```
% mail -f
Mail [5.2 UCB] [IBM AIX 3.2]  Type ? for help.
"/home/janet/mbox": 3 messages
>   1 john@sushi.raw.jp    Fri Dec 23 19:24  11/309 "Who cares?"
    2 prize@nobel.vax.swe   Fri Dec 23 20:04  12/319 "rejection of prize??"
    3 root@tauon   Tue Dec 20 09:02  12/537 "Closure of account"
&
```

12.3. Telnet and File Transfer Protocol (FTP)

If you are logged on to an account on a computer on the internet, you are able to login to any other internet connected computer (providing you have an account). For example, if Janet had an account on **sushi** she is able to use **telnet** to log into that account from **tauon**:

```
% telnet sushi.raw.jp
Trying...
Connected to sushi.raw.jp
Escape character is '^]'.

AIX telnet (sushi.raw.jp)
IBM AIX Version 3 for RISC System/6000
(C) Copyrights by IBM and by others 1982, 1990.

login: janet
janet's Password:
```

Files can be transferred to other computers using email, but the most convenient way to transfer files, especially in bulk, is to use **ftp**. Janet can **put** the file **letter.tex** into her **sushi** account using **ftp** as follows.

```
% cd tex
% ftp sushi.raw.jp
Connected to sushi.raw.jp
Miscellaneous network information...
Name (sushi:janet): janet
331 Password required for janet.
Password:
230 User janet logged in.
ftp> put letter.tex
200 PORT command successful.
150 Opening data connection for todo.
226 Transfer complete.
124 bytes sent in 0.03409 seconds (3.552 Kbytes/s)
ftp> quit
221 Goodbye.
%
```

Files can be *retrieved* from the remote site (i.e. **sushi** in this case) using the **get** command in the ftp environment. Use the online help for other commands.

12.4. Talking over the Internet

A facility exits whereby one user can talk to another user on another internet computer (compatibility usually requires both systems to be operating under UNIX, but exceptions do exist). For example if Janet wants to talk to John in more detail about his paper, she can initiate contact by typing

```
% talk john@sushi.raw.jp
```

If all goes well, and John is logged on and choses to answer the call (instructions appear on the screen for the callee) then the connection will be set up and each will be able to see what the other types in real time (modulo a small time lag of a second or so).

It is possible to find out login information about a user (including if they are currently logged in) by using the **finger** command. For example, fingering John might result in the following:

```
% finger john@sushi.raw.jp
Login name: john                        In real life: John Hancock
Directory: /home/john                   Shell: /usr/bin/csh
On since Dec 23 18:21:10 on pts/0       9 minutes 16 seconds Idle Time
No Plan.
```

13. WORLD WIDE WEB

The World Wide Web is an interactive tool for searching data bases and bulletin boards on the international internet. It has fast become a standard in information exchange on a global scale. Users of the web should be aware of the fact that it costs money to download information from any remote site (e.g. international or interstate). This includes actions as simple as looking at a particular site to the transfer of picture or data files. This cost will be taken up by research groups and the amount of raw data downloaded (i.e. usage) is monitored.

One gains access to the World Wide Web through a program called **netscape**. Typing the command **netscape** brings up a X-window interface from which it is simply a matter of pointing to what you want to see with the mouse and clicking. There is a great deal of useful (and useless) information on the Web; due to costs users should try to control their forays into Web-land.

Honours homepage

A part IV homepage with useful links now exists end can be found at:

http://www.ph.unimelb.edu.au/part4.html

This page is being developed into a resource base for certain lecture courses.

Preprints

Preprint archive: to find/retrieve preprints in a wide range of areas go to http://xxx.adelaide.edu.au/

14. REFERENCES AND SUGGESTED READING (A FEW STARTERS)

UNIX, *vi* and all that:

“A Practical Guide to the UNIX System”, by M.G. Sobell, Benjamin Cummings, USA, 1984.

“Introduction to the UNIX Operating System”, by V.Y. Hansper (copies available in the Baker Lab. There is also a copy available in the form of a **.dvi** file which can be accessed from the Part IV “current students” home page. Save this file to one of your directories by clicking on the link, and then type “xdvi unix.dvi” to view it.)

Fortran C and C++ programing:

“The Essentials of FORTRAN”, by Rev. Dennis C. Smolarski, S.J., PhD Research and Education Association, 1994.

“C How to Program”, by H.M. Deitel and P.J. Deitel, Prentice-Hall, 1992

“The C Programming Language”, 2e, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1988

Numerical methods and software:

“Computational Methods in Physics and Engineering”, S. Wong, Prentice-Hall, 1992.

“Numerical Recipes Online” (Fortran and C): retrieve ps files from http://cfatab.harvard.edu/nr/nronline.html (also, see reference in the Baker lab (not to be removed!))

Netlib library of subroutines: see http://netlib2.cs.utk.edu

LaTeX:

“LaTeX user’s guide and reference manual”, L. Lamport, Addison-Wesley, 1997.

Or there are plenty of references online, for example:
http://newton.ex.ac.uk/people/resende/tex/node1.html

Acknowlegements

Many thanks to Dr. Mark Munro for a careful reading of the original manual and many useful suggestions and to Steven Karataglidis for help on some of the finer points of RevTeX.

15. APPENDIX

15.1. UNIX glossary

Below is a list of UNIX commands, with a brief description of each one. This list should give an indication of what is available. To find out more about any of these commands use the **man** utility.

15.2. Files and directories

pwd → Print Working Directory - tells you which directory you are currently in.
ls dirname → lists the contents of the named directory(defaults to the current directory) .
cd dirname → moves to the named subdirectory.
cd .. → moves up to the directory above the current one.
mkdir dirname → creates a new directory.
rmdir dirname → deletes the named directory.
touch filename → creates an empty file named *filename.*
vi filename → Starts the **vi** editor and opens the named file.
more filename → displays the named file using the **more** utility.
tail filename → displays the last part of a file.
rm filename → deletes the named file.
cp file1 file2 → makes a copy of file1 and names it file2.
mv file1 file2 → renames file1 to file2.
ln → makes a link to a file.
which programname → locates the named program file or utility.
whereis filename → searches for the named file.
find → finds files.

grep → searches for a pattern in a file.
awk → searches for and processes a pattern in a file.
comm → compares two files.
diff → displays differences between two files.

sort → sorts and/or merges files.
spell → checks a file for spelling errors.
ispell -t → checks a LaTeX file for spelling errors.
wc → gives a line, word, or character count.
uniq → deletes repeated lines in a file.

15.3. Printing

psf file.txt > file.ps → converts a file from text format to post-script format.
lpr -P bakerps file.ps → prints a post-script file.
lpr -h -P bakerps-duplex file.ps → prints with no header, and double-sided. **lpq** → displays information about the print-queue and printer-status.

15.4. Monitoring and controlling system processes

monitor → displays information about system events and processes.
ps → lists currently active processes, and their process ID numbers(PID).
who → lists currently logged in users.
finger name → tells you whether the named person is logged on, and displays their .plan file.

last name **-n** → gives the last n times the named person has logged on.

lpq → displays information about print-queue and printer-status.
chmod → changes access mode to a file.
file → manipulates a file name and/or attributes.
kill x → terminates process x, where x is the process ID number(PID). See also the "ps" command.

sleep x → puts the process with ID number x to sleep.
stty → sets, resets, or reports on workstation parameters.

15.5. Communication

talk username → allows you to correspond with another user in real-time. Especially useful when they're sitting right next to you.

write username → similar to talk.
mesg n → your account will refuse any talk or write messages, until you type **mesg y**.