UNIX/FORTRAN

Bourne Shell Script

Self-study Text

H08 Documentation 1

Naota Hanasaki

Tomoko Nitta

Introduction

This text is a computer self-study resource aimed at undergraduate and postgraduate students studying hydrology. By working through this study text, you can learn the basic skills with UNIX, Fortran, and Bourne Shell scripts that you will need for doing data analyses, computer simulations, and the like in hydrology.

This study text is based on transcripts of lectures in a course on UNIX for beginners that I had been delivering at the Oki Laboratory of the University of Tokyo Institute of Industrial Sciences. The Oki Laboratory conducts research into the Earth's water circulation systems, and has achieved great results with computer simulations and data analyses of the world's hydrological cycles. It is crucial for students writing theses for masters and doctoral degrees at the laboratory to have some familiarity with UNIX and programming.

Specifically, in the summer semester, almost all students enrolled at the laboratory attend lectures in advanced hydrology, which are overseen by Professor Oki. In around 2001, the following assignment was set as part of this course.

> Read the following article:
>
> *Xie, P., and P. A. Arkin (1997), Global Precipitation: A 17-Year Monthly Analysis Based on Gauge Observations, Satellite Estimates, and Numerical Model Outputs, Bull. Amer. Meteor. Soc., 78, 2539–2558.*
>
> - Summarize their methodology
> - List up advantage/disadvantage
> - Compare the data with other data
> (e.g. station observation, global dataset, etc.)
> - See the site below for original data
> ftp://ftp.cpc.ncep.noaa.gov/precip/cmap/monthly/

Briefly, the assignment was to read Xie and Arkin's (1997) report on the creation of the CMAP (Climate Prediction Center Merged Analysis of Precipitation) data of global precipitation by month, summarize their methodology and other details, obtain the actual data, and compare it with other precipitation data.

The students (who were mostly first year masters students and first year doctoral students and had just joined the laboratory) struggled with the latter part of this assignment. CMAP is gridded data covering the globe at a resolution of 2.5°×2.5°. Because Earth has 360° of longitude and 180° of latitude, the data for each month has values for 144×72=10,368 locations. The scale of this can be grasped by visualizing a Microsoft Excel spreadsheet with numbers arranged in 144 columns by 72 rows. The data can be downloaded from a website in the form of a text file, in which the 10,368 locations are arranged vertically in 12 monthly blocks, which makes 10,368×12=124,416 lines.

While most of the students could use Microsoft Excel, current versions of Excel could only open a text file of up to 65,536 lines, so the students started off at a loss as to how to open the file. Even when they could open the file, they then struggled to create maps. Given that CMAP data covers the globe, they wanted to draw precipitation maps. However, at the time there was no instruction on the workings of GIS software at the University of Tokyo Department of Civil Engineering, so most of the students did not know how to draw the maps. In conclusion, the students could not complete the assignment to Professor Oki's satisfaction unless they could master the UNIX computers that we had at the laboratory.

Putting it in other words, postgraduate students at the Oki Laboratory needed to be familiar with UNIX. Moreover, to complete the assignment for Professor Oki's summer semester lectures, they needed to quickly learn how to use UNIX to analyze the CMAP data. For the latter objective, the following skills are required.

- Controlling UNIX
- Programming with Fortran[1]
- Making maps with the Generic Mapping Tools (GMT) software[2]
- Techniques for using Bourne Shell scripts to efficiently deal with large volumes of data

I was a graduate student at the time, and I ran a UNIX course from 2002 to 2005, for which I prepared lectures to enable students to meet these requirements, and I created transcripts of the lectures. With the assistance of many people, I have now reworked those transcripts into this study text, organizing them into chapters and expanding the content. The skills described in this study text should be useful not just for completing the assignment but also in research for your masters and doctoral theses. They should, moreover, be useful for anyone dealing with the H08[3] global water resources model. The basics of all the techniques used in the H08 model can be learned from this study text.

I have prepared this study text with the emphasis on making it possible to complete without it getting tedious, even for beginners. My intention is that readers will feel they are making progress toward being able to complete the assignment as they read through each chapter. One consequence of this is that there are some places where the explanations may seem to be out of order or disorganized. Note that the study text is not intended to provide comprehensive explanations of UNIX, Fortran, and Bourne Shell scripts. It is basically limited to describing techniques that are not described in detail in

---

[1] The Fortran language is still very widely used in the earth sciences. Many of the most important computer programs in the earth sciences, such as the atmosphere–ocean coupled Model, are written in Fortran.

[2] http://gmt.soest.hawaii.edu/

[3] The global water resources model described in Hanasaki et al. 2008 a,b (http://direct.sref.org/1607-7938/hess/2008-12-1007, http://direct.sref.org/1607-7938/hess/2008-12-1027). A hydrological model capable of simultaneously calculating natural hydrological cycles and major human uses of water over the globe.

other study texts but are necessary for dealing with hydrological data. Other study texts are listed as references in Appendix D, and I recommend that you read them in parallel with this study text.

Speaking from my own experience, being able to understand UNIX and do programming makes a huge difference in the data and simulations that you will be able to handle. I sincerely hope that this study text will motivate you to get to grips with UNIX and programming.

Acknowledgements: The authors are grateful to Mr. Hiroshi Ueno for editing the manuscript. This is a translation of a study text of the same title from Japanese into English. Translation was funded by “Integrated Study Project on Hydro-Meteorological Prediction and Adaptation to Climate Change in Thailand” (IMPAC-T).

Contents

Chapter 1 8
1.1 The UNIX Environment 8
1.2 Login and Logout 8
1.3 UNIX Commands (1): Managing Files 8
1.4 UNIX Commands (2): Editing Files 11
1.5 UNIX Commands (3): Other Commands 13
1.6 UNIX Commands (4): Troubleshooting 13
1.7 Permissions 13
Exercises 15
Chapter 2 16
2.1 Redirects 16
2.2 Pipes 17
2.3 Wildcards 18
2.4 AWK 19
Chapter 3 22
3.1 Introduction to GMT 22
3.2 GMT Shell Scripts 23
3.3 Options in GMT Commands 26
3.4 CPT Files 28
Exercises 29
Chapter 4 30
4.1 ASCII and Binaries 30
4.2 Converting Data by Month to Data by Year 31
4.3 Data Deviation and Mapping 35
Exercises 36
Chapter 5 37
5.1 Basics of Fortran Source Code 37
5.2 Compiling Fortran Source Code 38
5.3 Input and Output of Fortran Files 39
Exercises 43
Chapter 6 44
6.1 Data Used in This Chapter 44
6.2 Commands for Controlling the CMAP Binary Files 45
Exercises 50
Chapter 7 51
7.1 Input and Output of Time Series Files in Fortran 51
7.2 Fortran Subroutines 56
7.3 make and makefiles 57

Exercise 59
Chapter 8 60
8.1 Control Statements and Conditional Expressions in Shell Scripts 60
8.2 Example 1: A shell script using a for loop 61
8.3 Example 2: A shell script using a while loop 62
8.4 Example 3: Performing Exercise 2 of Chapter 6 with a single Bourne Shell script 63
Exercise 64
Appendix A 65
A.1 Installing Xming 65
A.2 Using Xming 67
A.3 Troubleshooting 70
Appendix B 71
B.1 Obtaining the Resources for this Study Text 71
B.2 Using the Intel Fortran Compiler 72
B.3 Input/Output in Little-endian 72
Appendix C 74
C.1 Minimum UNIX Requirements for Working with this Study Text 74
C.2 An Example of Setting Up a UNIX Environment (Mac OS X 10.6) 74

Chapter 1
Basic Control of UNIX (1): Basic UNIX Commands

This chapter describes basic operations in UNIX, such as how to log in to UNIX and how you can control UNIX.

1.1 The UNIX Environment

The following kind of computer environment is required for this study text: a computer on which a UNIX-based operating system (OS) such as UNIX or Linux is installed (this will be called a UNIX computer in this study text), which is connected to a network and on which a personal account has been created for you. Alternatively, if you have a computer with the Windows operating system installed (called a Windows computer in this study text) and this is connected to a network, from your Windows computer you can log in to a UNIX computer over the network and use the UNIX environment. The Windows computer is a client and the UNIX computer is a server. You will need a kind of software known as a PC X Server program to log in to the UNIX computer and use the UNIX environment. In this study text, it will be assumed that you are using the free software called Xming.

See Appendix A for how to install and use the Xming software, and see Appendix B and Appendix C for configuration of a UNIX computer.

1.2 Login and Logout

Login:

1. Use Xming to connect to the server (see Appendix A).
2. Enter your username and password.
3. When a window appears on your screen, you have logged in. This window is called “the terminal”.

Logout:

Type “Exit” into the terminal or click the X button on the window.

1.3 UNIX Commands (1): Managing Files

In Windows, you use a mouse and click on icons to control the computer. This kind of interface is called a GUI (Graphic User Interface). In contrast, in UNIX you generally use the keyboard and type commands into the terminal to control the computer. This kind of interface is called a CUI (Command User Interface). Basic commands for controlling UNIX are shown in Table 1-1 to Table 1-4. Try typing them into the terminal one at a time to see the functions they perform.

Table 1-1 shows commands for managing files. These can be compared to operations in the Windows Explorer program. Where the word “directory” appears, you can understand it as meaning the same thing as a folder in Windows. The percent signs (%) in the table represent a prompt (which will be a character string such as pc090-041[35] that is displayed in the terminal). Do not type the percent signs.

Table 1-1 Managing Files (***directory*** represents the name of a particular directory, and ***file*** represents the name of a particular file)

Show the name of the current directory (pwd stands for “present working directory”)	% pwd
Show the names of files and directories within the current directory (ls is short for “list”)	% ls
The same as above but showing more detailed information about the files and directories (l stands for “long”)[4]The results show permissions, number of links, owner, group, size of file, month, day, and year/time of last modified, name of file/directory.	% ls –l
Move to a different directory (cd stands for “change directory”)[5]	% cd ***directory***
Move up one directory level (the two periods mean the directory one level up)	% cd ..
Move to the Home directory[6] (the tilde (~) means the Home directory)	% cd ~
Move to the current directory (the single period means the current directory)	% cd .
Display the contents of a file	% less ***file*** % more ***file*** (Press the Enter key to scroll down one line, press the space bar to scroll down one screen, and press the Q key to end this display.) % head ***file*** (This shows only the first 10 lines of the file.) % tail ***file*** (This shows only the last 10 lines of the file.)
Edit a file (Open a new window)	% emacs *file*
Edit a file	% emacs –nw *file*

[4] A character string starting with a hyphen (-) after a command as in this example is called an option. You can alter the functioning of commands by adding options to them.
[5] A character string after a command as in this example is called an argument. You can provide information to be used by commands by providing arguments.
[6] 5 The home directory is the first directory that a user starts with. In Windows, this would be C:¥Documents and Settings¥guest. In Mac OS X, the home directory for the login name "guest" is /~ and in most UNIX-based systems it is /home/guest.

Copy a file (cp is short for “copy”)	% cp ***file1 file2*** (***file1*** is the file being copied and ***file2*** is the copy)
Move a file/change the file’s name (mv is short for “move”)	% mv ***file1 file2*** (***file1*** is the file before the move and ***file2*** is the file after the move)
Delete a file (rm is short for “remove”)	% rm ***file***
Create a new directory (mkdir is short for “make directory”)	% mkdir ***directory***
Delete an empty directory (rmdir is short for “remove directory”)	% rmdir ***directory***
Delete a non-empty directory (-r stands for “recursive)	% rm –r ***directory***
Copy a directory	% cp –r ***directory1 directory2***
Change permissions (see Section 1.7) (chmod is short for “change mode”)	% chmod ***abc file*** a: for the user: +4 to this number to permit reading the file, +2 to permit writing, +1 to permit execution b: for the group: +4 to permit reading, +2 to permit writing, +1 to permit execution c: for others: +4 to permit reading, +2 to permit writing, +1 to permit execution
Change permissions, in another format (see Section 1.7)	% chmod ***abc file*** a: u stands for user, g for group, o for others b: + to add a permission, - to remove a permission c: r for reading, w for writing, x for execution
Compress a file (gzip is short for GNU zip)	% gzip ***file***
Decompress a file (gunzip is short for GNU unzip)	% gunzip ***file***
Finish	% exit

1.4 UNIX Commands (2): Editing Files

You can use a program called Emacs, which is similar to the Windows Notepad software, for editing text files. The Emacs program should be installed on nearly all UNIX computers. In Emacs you can use keyboard shortcuts to save files, copy sections, etc. Table 1-2 shows commands for editing in Emacs.

Table 1-2 Using Emacs to edit files.
(Ctrl-x means pressing the x key while pressing the Ctrl key.
Ctrl-x + Ctrl-s means pressing the x key while pressing the Ctrl key and then pressing the s key while pressing the Ctrl key.)

Start Emacs	% emacs ***file***
Save file	Ctrl-x + Ctrl-s
Finish	Ctrl-x + Ctrl-c
Cancel a command (If you typed in a command but it did not work satisfactorily, you can often get back to a previous state by repeating this operation two or three times.)	Ctrl-g
Delete a character	Ctrl-d
Delete a line (kill)	Ctrl-k
Cut[7]	Ctrl-Space (to set start point) Ctrl-w (to set end point)
Paste (yank)	Ctrl-y
Move the cursor right (forward)	Ctrl-f
Move the cursor left (backward)	Ctrl-b
Move the cursor up (previous)	Ctrl-p
Move the cursor down (next)	Ctrl-n
Scroll down a screen	Ctrl-v
Scroll to the last screen	Esc + >
Scroll up a screen	Esc + v
Scroll to the first screen	Esc + <
Search	Ctrl-s
Replace	Esc + %
Replace all	Esc-x replace string

[7] There is no copy command as there is in Windows. When copying text, you must first cut it and then paste it back in the same place.

1.5 UNIX Commands (3): Other Commands

Table 1-3 shows commands that are often used with image files for display, printing, etc.

Table 1-3 Other Commands

Start the Firefox browser[8]	% firefox
Display a JPEG or GIF file[9]	% display ***file***
Convert an image file[10]	% convert ***file1 file2***
Print[11]	% lpr -P***printer file***
Show print jobs	% lpq –P***printer***
Delete a print job	% lprm –P***printer jobID***
Show a list of users logged in to the computer	% who
Display the date	% date
Display a calendar for the year ***year***	% cal ***year***
Show the numbers of lines, words and characters in a file	% wc ***file***

1.6 UNIX Commands (4): Troubleshooting

When you have a problem, try one of the commands in Table 1-4.

Table 1-4 Troubleshooting commands

Display help (a manual) for a command	% man ***command***
Interrupt a command, before the prompt reappears.	Ctrl-c
Show the current state and process ID of a command	% top
Force a command to stop	% kill ***processID***
Force a command to stop (more powerful)	% kill -9 ***processID***

1.7 Permissions

Permissions are a very important concept in UNIX, so please make sure you understand them. First, there are three ways for a user to access a file or directory.

- Reading: Opening and reading a file.
- Writing: Creating a new file or changing the content of an old file.

[8] If Firefox is not installed at the computer, type the name of another browser.
[9] This can only be used if a program called ImageMagick is installed. See Appendix C for details.
[10] This can only be used if a program called ImageMagick is installed. See Appendix C for details.
[11] For example, if a file called test.txt is being printed at a printer called flood, % lpr -Pflood test.txt.

- Execution: Executing a file as a set of commands

In UNIX, every file or directory belongs to a particular individual user. The users are organized into groups. Thus, there are three kinds of relationship between a file or directory and users.

- User: The user who is the owner of the file or directory
- Group: Users who are not the owner but who belong to the same group as the owner
- Others: Users who are not the owner and do not belong to the same group as the owner

Executing files may be difficult to understand, so try a practical example. Use Emacs to create the following text file.

```
pwd
ls
who
```

Save this file with the name “example1.txt”. This is the same as creating an ordinary text file in Windows.

Now change the permissions and give yourself permission to execute it. You will want to read, edit, and execute this file yourself. Users in your group and other users can be allowed to read it, but you do not want them to change or execute it. Therefore, use the following command.[12]

```
% chmod 744 example1.txt
```

Alternatively, the following command has the same effect in this case.[13]

```
% chmod u+x example1.txt
```

Now you can execute the file. Try executing example1.txt.

```
% example1.txt
```

You have seen that in UNIX, unlike Windows, text files can be executed (provided execution is permitted). When a text file is executed, the content written in the file is interpreted as commands.

[12] Referring to Table 1-1, for the user to read, write and execute, their number is 4+2+1=7. For the group and others to only be able to read, their numbers are 4+0+0=4. Putting the numbers together makes 744.

[13] The argument u+x gives execution permission to the user. There are no other changes in this example.

Exercises

1. Find the name of the current directory?
2. Move to your home directory (~).
3. Create a directory with the name “CMAP”.
4. Move to the “CMAP” directory you created.
5. Copy the following from ~/unix_semi/lesson1
   - global.sh
   - grad.cpt
   - data.xyz
6. Change the permissions of global.sh to give yourself execution permission.
7. Execute global.sh.
8. Display the output image file “image.eps”. As will be discussed in the next chapter, this represents global precipitation data for January 1979.
9. Convert the image file image.eps from the EPS format to the GIF format.

(Solutions to the exercises are in ~/unix_semi/lesson1/exercise1.txt.)

Chapter 2

Basic Control of UNIX (2): Redirects/Pipes/Wildcards/AWK

Open the file global.sh that you used in Chapter 1. The file is made up of a large number of short commands. If you look at it closely, you can see many special characters such as >, >>, <, <<, and |. These symbols are very important in controlling UNIX. In this chapter you will learn about these symbols.

2.1 Redirects

The symbols < and > are called redirects. Redirects are used when you want to write output from a command to a file or read input for a command from a file.

Table 2-1 Types of redirect

To write the output of a command into a new file	% ***command*** > ***file***
To add the output of a command to the end of a file	% ***command*** >> ***file***
To write the output of a command to a file, overwriting the original file.[14]	% ***command*** > ***file***
To execute a command using a file as input	% ***command*** < ***file***
To execute a command using text typed in at the keyboard as inputs, until "EOF" is typed in.	% ***command*** << EOF *strings* EOF

Example 1

Type the following in at the terminal. The date command outputs the current date and time. The wc command counts characters and lines, giving three outputs—the number of lines, the number of words, and the number of characters.

```
% date > redirect.txt
% more redirect.txt
% date >> redirect.txt
% more redirect.txt
% date > redirect.txt
% more redirect.txt
% wc < redirect.txt
```

If it doesn't work, try

```
% date >! redirect.txt
```

[14] If you type in `% echo $SHELL` at the terminal and tcsh or csh is displayed, this means that a function that prevents overwriting by redirection has been turned on. If there is already a file called redirect.txt, use ***`% command >! file`*** to overwrite it.

```
% wc << EOF
Hello.
My name is Naota.
EOF
```

```
% wc << EOF >> redirect.txt
Hello.
My name is Naota.
EOF
% more redirect.txt
```

Exercise 1

1. Create a file storing a list of the people currently logged in at the server. (Make use of the who command in Table 1-1.)
2. Create a file storing a calendar from 2004 to 2005. (Make use of the cal command in Table 1-1.)

(Solutions to Exercise 1 are in ~/unix_semi/lesson2/exercise1.sh.)

2.2 Pipes

The symbol | is called a pipe. A pipe passes the output of a command to the input of a following command.

Table 2-2 Piping

First command1 is executed, and then command2 is executed, using the output of command1 as the input of command2.	% ***command1*** **|** ***command2***

Example 2

```
% cd ~/unix_semi/dat_bin
% ls -l
% ls -l | more
% ls -l | tail
% ls -l | head
```

Exercise 2

1. Count the number of files and directories in your home directory. (Make use of the wc command.)
   - Try to use redirection
   - Try to use piping

(A solution to Exercise 2 is in ~/unix_semi/lesson2/exercise2.sh.)

2.3 Wildcards

The symbols ? and * are called wildcards. The question mark ? represents any single character, and the asterisk * represents any string of characters.

Table 2-3 Wildcards

Any string of characters, including a null string[15]	*
Any single character	?

Example 3

```
% cp -r ~/unix_semi/lesson2/wildcard .
% cd wildcard
% ls -l
% ls -l *txt
% ls -l test*
% ls -l te?t1.txt
```

Don't miss "." (period)

Exercise 3

1. Move to the "wildcard" directory that you copied in Example 3.
2. Delete files that have the numeral "1(one)" in the filename.
3. Delete all files in the "wildcard" directory.
4. Copy all files from ~/unix_semi/lesson2/wildcard without using -r as in Example 3. (Make use of the * wildcard.)

(A solution to Exercise 3 is in ~/unix_semi/lesson2/exercise3.sh.)

[15] This means * can match an absence of characters.

2.4 AWK

AWK is a kind of programming language, which is often used for dealing with text-based data. It is an interpreted language, which does not need compiling, so it can be used by typing straight into the terminal. Its grammar is very similar to the C language. AWK can be used for quite sophisticated programming, but in this section you will only learn the two uses shown in Table 2-4. In Table 2-4, the parts between the quote marks ‘ and ’ are source codes that are interpreted in AWK.

Table 2-4 AWK commands

Output the data in column N_1 and N_2 of a file	% awk ‘{print $$N_1$, N_2}’ ***file***
When column M in a file is m, output the data in column N[16]	% awk ‘($***M***==***m***){print $***N***}’ ***file***
E.g., when column 1 is the character string “a”, output the data in column 2	% awk ‘($1==“a”){print $2}’ ***file***
E.g., when column 1 is the number 12, output the data in column 2	% awk ‘($1==12){print $2}’ ***file***

Example 4

```
% cd ~
% ls –l > home.txt
% awk ‘{print $9}’ home.txt
% awk ‘($5==512){print $9}’ home.txt
```

Open the file data.xyz that you used in Chapter 1 and look at it again. The data is CMAP precipitation data for January 1979, which is sequentially recorded with longitudes in the first column, latitudes in the second column and precipitation levels (in mm/day) in the third column. Because CMAP covers the world in 2.5°×2.5° units, there are 144×72=10,368 lines. As will be described in Chapter 3, global.sh reads data.xyz and plots an image. In other words, the map created in Chapter 1 visualized data.xyz.

Now look at the files distributed by the CMAP website (ftp://ftp.cpc.ncep.noaa.gov/precip/cmap/monthly/). Download the file with monthly data for 1979(see Appendix B).[17] This file is compressed, so decompress it. (Use the gunzip command to decompress a gz compressed file.)

[16] If m is a character string, it is enclosed between the quote marks “ and ”. If m is a real number or an integer, it is put just as it is.

[17] The easiest way to download a file is to use a Web browser such as Firefox. However, you can use commands to download a file, as follows.

```
% ftp -i ftp.cpc.ncep.noaa.gov ←Log in to an FTP server (Note 1)
% ls                            ←Display a list
```

```
% gunzip cmap_mon_v0705_79.txt.gz
```

Now display the contents of the file. Figure 2-1 shows a legend for the file. You can see that it has a quite different format from data.xyz.

```
% more cmap_mon_v0705_79.txt
```

```
% cd precip/cmap/monthly          ←Move directory
% binary                          ←Set the file transfer mode to binary (Note 2)
% get cmap_mon_v0705_79.txt.gz    ←Download the file (Note 3)
% quit                            ←Finish FTP
```

Note 1: If a "Name" box appears, type in "anonymous". If a "Password" box then appears, carefully type in your e-mail address. You should then be able to log in.

Note 2: There are two file transfer modes—binary and ASCII. With binary, a file is transferred just as it is. With ASCII, a file is transferred with linefeed codes that differ between different operating systems being automatically converted. Except when transferring text files, you should always select binary mode.

Note 3: The command to download a file is `% get` ***file***. If you are using a wildcard such as `*` to select files, as you did earlier in this chapter, use `% mget` ***file***. For example, use `% mget *` to download all the files in a directory.

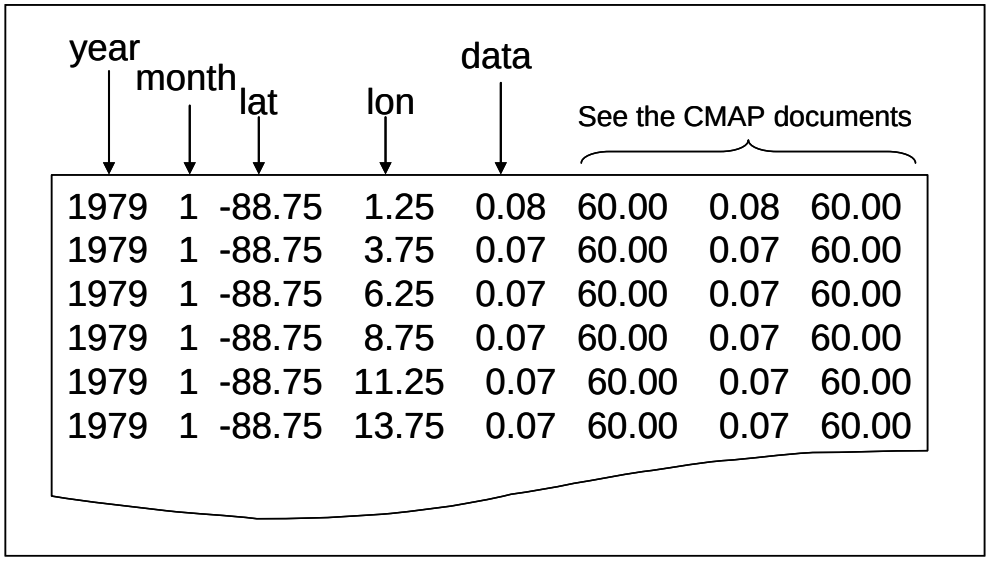


Figure 2-1 Legend for CMAP files

Exercise 4

1. Figure 2-1 shows the format of the file cmap_mon_v0705_79.txt that you downloaded. Study this format and use AWK to create a file in exactly the same format as data.xyz. Note that data.xyx is a file of data for January 1979 written in the format: longitude, latitude, precipitation.

(A solution to Exercise 4 is in ~/unix_semi/lesson2/exercise4.sh.)

If you wish to adjust the beginnings of each number, just search the internet for technical reference.

Chapter 3

Bourne Shell Scripts (1): Combining GMT Command

In Chapter 1, you executed the file called global.sh and drew a map. In Chapter 3 you will learn how to draw this map, and how you can change it.

3.1 Introduction to GMT

Looking back to Chapter 1, you typed the following commands and outputted the map shown in Figure 3-1.

```
% global.sh
% display image.eps
```

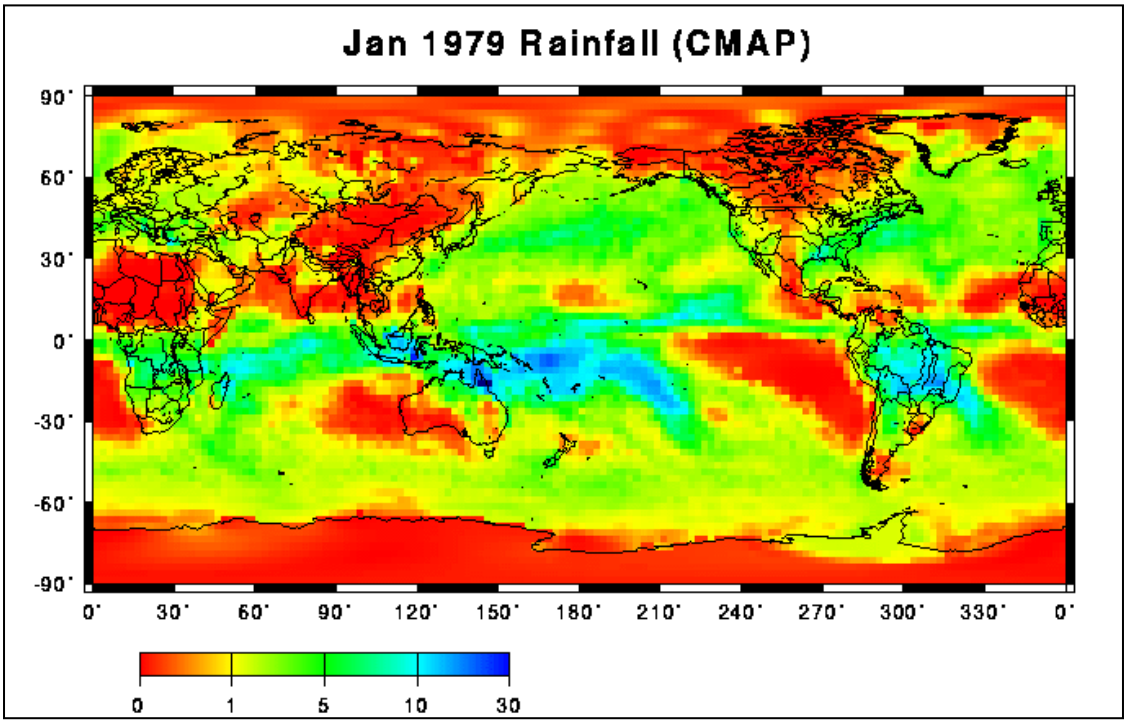


Figure 3-1 Output figure of global.sh

A free software program called the Generic Mapping Tools (GMT)[18] was used for making this map. GMT has a large number of commands with forms similar to ls, cd and the like. The global precipitation map was drawn by these GMT commands being combined in the file global.sh.

The following three kinds of file are required when using GMT for mapmaking.

- A source file (global.sh)
- A data file (data.xyz)

[18] http://gmt.soest.hawaii.edu/

- A CPT file (grad.cpt)

The roles of these files are described below.

3.2 GMT Shell Scripts

Open global.sh in a text editor. Recall Section 1.7, in which you permitted execution and then executed a text file containing commands such as pwd, ls, and who. A large number of commands are similarly written in global.sh. However, global.sh is written in accordance with Bourne Shell grammar and controls execution of the commands accordingly. This kind of file is called a Bourne Shell script. That is, global.sh is a Bourne Shell script containing GMT commands. In this section, the file is divided into three blocks for description, respectively called the first block, the second block, and the third block.

```
 1:#!/bin/sh
 2:##############################################################
 3:#to   draw a global precipitation map
 4:#on   31/May/2002
 5:#by   nhanasaki
 6:#at   IIS,UT
 7:##############################################################
 8:# Define File Names
 9:##############################################################
10:DATFILE=./data.xyz      #data file (input)
11:#DATAFILE=./cmap_mon_v0203_79.txt
12:EPSFILE=./image.eps      #image file (output)
13:GRDFILE=./grd            #temporary file
14:CPTFILE=./grad.cpt       #color palette table file
```

Definition of shell

Assign values to variables

After a # are not interpreted

This is the first block. In the first line, #!/bin/sh is a character string indicating that the file is a Bourne Shell script. Lines 2 to 9 start with the # symbol. Any characters coming after a # are not interpreted by the computer, so are used for making comments, marking out sections of the file, and suchlike, as in this example. This kind of statement is called a comment.

In each of line 10 to line 14, two character strings are joined by an equals sign. The character string to the left of the equals sign is a “variable”, which can be visualized as an empty box. The character string to the right of the equals sign is a “value”, which can be visualized as some kind of object. When a variable and value are joined by an equals sign, the value is assigned to the variable, which can be visualized as the “value” object being put into the empty “variable” box. For example, in DATFILE=./data.xyz in line 10, ./data.xyz is the value (object) being put into the variable (box) DATFILE. In line 10, the name of a file is being defined.[19]

[19] If you wish to use the original CMAP file (cmap_mon_v0203_79.txt), remove # of the line 11, and put # at the beginning of line 10.

```
 1:##############################################################
 2:# Define Mapping Area
 3:##############################################################
 4:XMIN=0.00              #Horizontal minimum [degree]
 5:XMAX=360.00            #Horizontal maximum [degree]
 6:YMIN=-90.00            #Vertical minimum [degree]
 7:YMAX=90.00             #Vertical maximum [degree]
 8:XWID=21.0              #Width of image [cm]
 9:YWID=10.5              #Height of image [cm]
10:DXa=30.0               #a:Horizontal Annotation Interval [degree]
11:DXf=30.0               #f:Horizontal Frame Interval [degree]
12:DXg=10.0               #g:Horizontal Grid Interval [degree]
13:DYa=30.0               #a:Vertical Annotation Interval [degree]
14:DYf=30.0               #f:Vertical Frame Interval [degree]
15:DYg=10.0               #g:Vertical Grid Interval [degree]
16:D=2.5                 #grid size
17:##############################################################
18:# GMT Options
19:##############################################################
20:RFLAG=-R${XMIN}/${XMAX}/${YMIN}/${YMAX}
21:JFLAG=-JX${XWID}d/${YWID}d
22:BFLAG=-Ba${DXa}f${DXf}g${DXg}:Longitude:/a${DYa}f${DYf}g${DYg}:Latitude:neWS
```

Variable DYa is substituted by its value 30.0

Moving onto the second block, a succession of values are assigned to variables from line 1 to line 16. These define the plotting region. See Figure 3-2 for the meanings of these variables and values.

Line 17 to line 22 defines -R, -J and -B options which are used by GMT. In these lines, the expression to the right of the equals sign has the symbols ${ }. This means that the expression is to be substituted with the value assigned to a variable. For example, where ${XMIN} is written, it will be substituted with the value assigned to the variable XMIN. The variable XMIN was defined with the value 0.00 in line 4, so -R${XMIN}/ is substituted with -R0.00/.

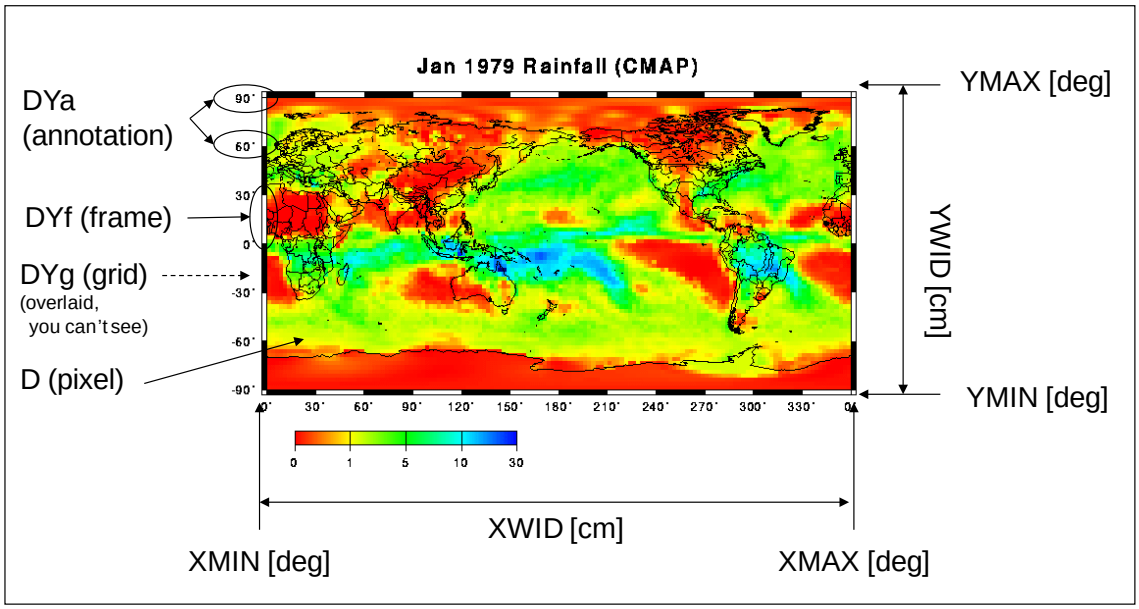


Figure 3-2 Plotting region of GMT

```
 1:##########################################################
 2:#---Jobs
 3:##########################################################
 4:#awk '($2==1){print $4, $3, $5}' $DATAFILE | ¥
 5:#xyz2grd    $RFLAG -G$GRDFILE -I${D}/${D} -F
 6:xyz2grd    $RFLAG -G$GRDFILE -I${D}/${D} -F $DATFILE
 7:#
 8:psbasemap  $RFLAG $JFLAG $BFLAG -X5.0 -Y5.0          -K > ./$EPSFILE
 9:grdimage -O $RFLAG $JFLAG $GRDFILE -C$CPTFILE        -K >> ./$EPSFILE
10:pscoast  -O $RFLAG $JFLAG -Dc -W5 -N1 -I1            -K >> ./$EPSFILE
11:psscale  -O -C$CPTFILE -D5.0/-1.5/8/0.5h -L          -K >> ./$EPSFILE
12:pstext   -O $RFLAG $JFLAG -N                      << EOF >> ./$EPSFILE
13:180 110 24 0 0 1 6 Jan 1979 Rainfall (CMAP)
14:EOF
```

Redirect

(See Chapter 2)

Overlay plot mode

Code appended later

Read until EOF appears

Commands

In the third block, the commands are written. This is the main body of the shell script.[20] Line 6 shows the first command, in which $RFLAG[21] is substituted with the value assigned to the variable RFLAG. Thus, line 6 is exactly the same as writing the following.

xyz2grd -R0.00/360.00/-90.00/90.00 -G./grd -I2.5/2.5 -F

In this, xyz2grd is the command, and the character strings starting with hyphens are options. In Chapter 1, you studied the command ls -l, in which the addition of the option -l to the command ls which displays files extends the functionality to display a more detailed list. In line 8, note the redirect (>) after the command psbasemap. If you do not recall how redirection works, have another look at Chapter 2.

The operations that the third block actually does can be conceptually represented as in Figure 3-3.

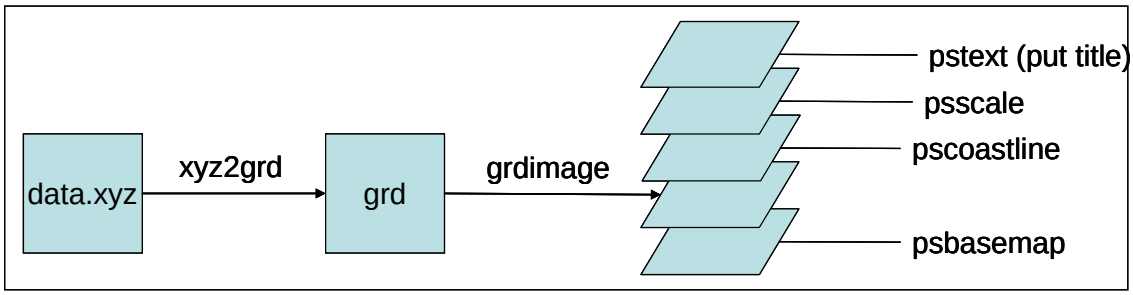


Figure 3-3 Procedure of global.sh

First, xyz2grd is a GMT command that converts a text file (the variable DATFILE, with the value data.xyz) in which each location is represented in the form “longitude - latitude - data”, to a two-dimensional form of gridded data, such as a map. The result of xyz2grd is that an intermediate file is created (the variable GRDFILE, with the value ./grd). The commands psbasemap, grdimage, pscoast, psscale, and pstext are mapmaking commands. GMT makes a map as if overlaying overhead projector

[20] If you wish to use the original CMAP file (cmap_mon_v0203_79.txt), remove # of the line 4 and 5, and add # at the beginning of line 6.

[21] ${RFLAG} and $RFLAG are the same; the { } may be omitted.

transparencies or sheets of tracing paper. As the name indicates, psbasemap is a command that creates the base map frame. The background image that is created is written to an output image file (the variable EPSFILE, with the value image.eps). The command grdimage draws the intermediate file created by xyz2grd. It specifies colors corresponding to the precipitation levels and writes them into the output image file. The command pscoast draws coastlines, national boundaries, rivers, etc.; the command psscale draws a color bar; and the command pstext draws text strings (a title and so forth). Arguments are: x coordinate, y coordinate, size of strings, angle of strings (zero for horizontal), ID of font (zero for default), and ID of justification (six for vertical & horizontal center).One after another, these are written to the output image file.

3.3 Options in GMT Commands

Options that are often used with GMT commands are summarized in Table 3-1. Refer to the GMT manual for more details on how to use them. There are also some excellent websites, which you can find with a search engine such as Google.

Two particularly important options are -O and -K. Because a map is drawn in GMT by layers being overlaid in the manner of OHP transparencies, indicating which of these layers is at the bottom and which is at the top is important. If the output of a GMT command is not the first (bottom) layer, the option -O must be appended. “-O” stands for “Overlay plot mode”, and means that header information signifying the first layer should be omitted. Similarly, if the output is not the last (top) layer, -K must be appended. “-K” represents “More PostScript code will be appended later”, and means that footer information signifying the last layer should be omitted.

Table 3-1 GMT command options

Options usable with various GMT commands	
-R	Sets a plotting area
-J	Sets the map projection
-B	Sets how the axes are drawn
-O	Omits a header marking the first layer
-K	Omits a trailer marking the last layer
Options with psbasemap	
-X	Moves the origin in the x direction (unit: cm)
-Y	Moves the origin in the y direction (unit: cm)
Options with pscoast	
-D	Resolution of the map
-W	Line thickness
-N	Plotting of national boundaries
-I	Plotting of rivers
Options with psscale	
-D	Sets the position and size of the color bar
-L	Makes intervals in the color bar uniform
Options with pstext	
-N	Allows text strings to be displayed outside the basemap

3.4 CPT Files

In Figure 3-1, low precipitation levels are shown in red and high precipitation levels are shown in blue. The relationship between colors and precipitation levels is shown in the color bar at the bottom left. You can see that the five colors red, yellow, green, cyan, and blue are assigned to precipitation levels from 0 to 30. This specification of the colors is defined in a CPT file (Color Palette File). The CPT file used for Figure 3-1 is called grad.cpt.

Open grad.cpt, which you used in Chapter 1. It should appear as in Figure 3-4. A CPT file is basically made up with each line having eight columns of numbers. These represent, in order, a start value, R, G, and B values, an end value, and R, G, and B values. For example, when the color is to be altered in steps between red (RGB 255/0/0) at a start value of 0 and yellow (RGB 255/255/0) at an end value of 1, this is written as follows.

```
0	255	0	0	1	255	255	0
```

<u>You must use the tab key to separate these numbers; take care not to use spaces, as this will cause an error.</u> For colors to be assigned to values that are smaller than the minimum value (0), values that are larger than the maximum value (30), and values that are not numbers, RGB values are entered after, respectively, B (background), F (foreground), and N (not a number). In grad.cpt, dark red is set for values less than 0 and dark blue is set for values greater than 30.

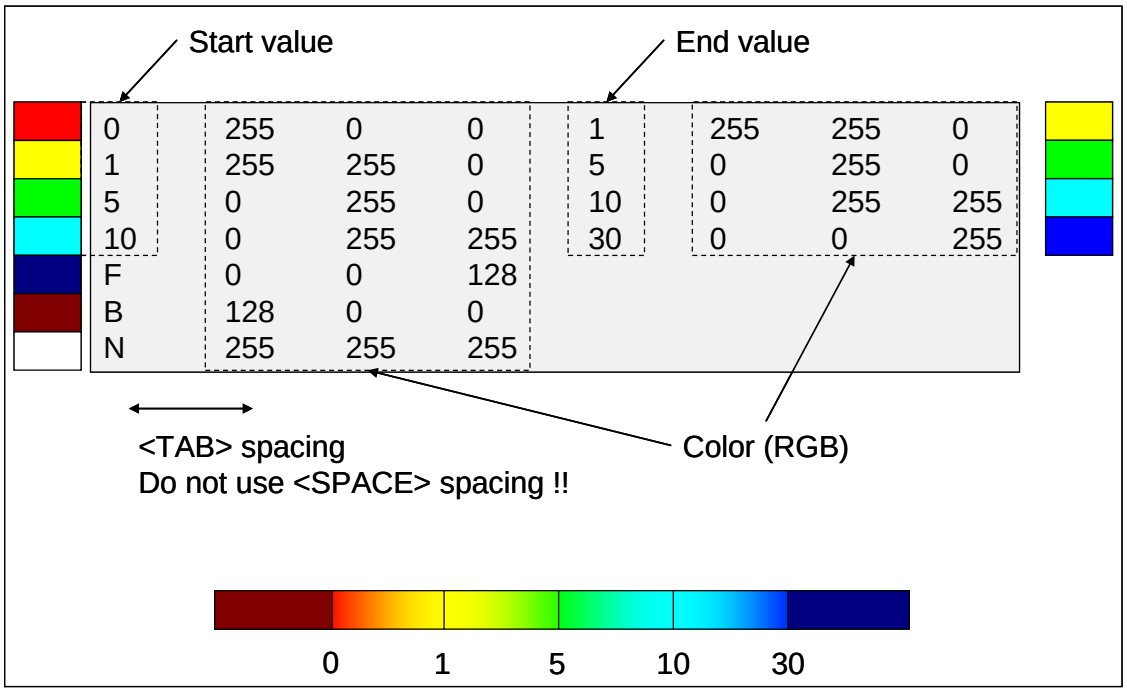


Figure 3-4 Legend for grad.cpt

Exercises

Change the script global.sh and draw maps as follows.

1. Change the title of the map (to anything you like).
2. No unit is given for the color bar. Put “Unit: mm/day” at a suitable location on the color bar.
3. Magnify the map to display a magnified view of Iran. Set 60° in the horizontal direction and 30° in the vertical direction. Then, because Iran has little rain, change the current CPT file (grad.cpt) to make the distribution of rain in Iran clearer. Finally, change the title of the map and add “Unit: mm/day”.

(Solutions to the exercises are in ~/unix_semi/lesson3.)

Chapter 4
Manipulating Grid Data (1): Calculations Using Fortran Programs

In this chapter, you will use the CMAP data to analyze global precipitation levels. The CMAP data is gridded data with a resolution of 2.5°×2.5° over the whole world, and data by month from 1979 to 2002 is available. You will create a map comparing precipitation amounts by year between 1987 and 1988. Incidentally, you may know that 1987 was an El Niño year and 1988 was an El Niña year.

4.1 ASCII and Binaries

As you saw in Chapter 2, the CMAP data is distributed in the form of text data. The whole of the distributed data has been downloaded and converted to the binary format, and this is in ~/unix_semi/dat_bin. You will use this binary data in this chapter.

There are basically only two ways of storing information in a computer—ASCII and binaries. ASCII is also referred to as text. You can read ASCII data with the Windows Notepad software or suchlike. In contrast, binaries are in machine code and cannot be read with Notepad. The characteristics of ASCII and binaries are summarized in Table 4-1.

Table 4-1. Comparison of ASCII and Binaries

	ASCII	Binaries
Readable with Notepad?	Yes	No
Differs between machines?[22]	No	Yes
Data units	1 byte/character	4 bytes/data unit

In ASCII data, one alphanumeric character is recorded in one byte.[23] In binary data, one real number can be recorded in four bytes. For example, the amount of memory needed to store the real number 1.0 is three bytes in ASCII and four bytes in binary, while the number of bytes needed to store the real number 1.000000 is eight bytes in ASCII but still four bytes in binary.

Because the CMAP data has the resolution 2.5°×2.5° (longitude x latitude), the data for one month is made up of 144×72 real numbers. If this data is stored in 144 columns horizontally and 72 rows vertically, as in Figure 4-1, the file volume is 144 × 72 × 4 =

[22] For example, a binary created on a machine with the Sun Microsystems SPARC architecture is different from a binary created on a machine with an Intel CPU. To learn about this in more detail, search the Web for the terms "big endian" and "little endian".
[23] Japanese characters are recorded in two bytes each.

41,472 bytes. Being real numbers, the values are all represented by four bytes, whether they be 1.0, 1.0000, or 9.87E+9. Therefore, the data for any month of any year in ~/unix_semi/dat_bin is always 41,472 bytes. The files have names in the form CMAP203_19870100.cmap. That particular name indicates that the data is from CMAP version 2.03 and is for January 1987.[24]

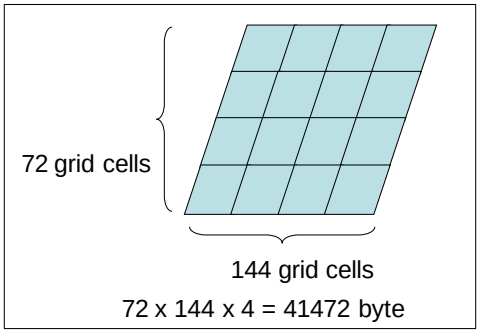


Figure 4-1 Legend for binary files of CMAP data

4.2 Converting Data by Month to Data by Year

Moving on to the main point of this chapter, your objective is to compare precipitation levels by year between 1987 and 1988. At the moment, you only have monthly data, so you will convert the monthly data from 1987 and 1988 to yearly data, and then calculate differences between the two years.

First, create yearly precipitation data. You can do this by adding the data from January to December and then dividing by 12 (for annual average daily figures), as illustrated in Figure 4-2.

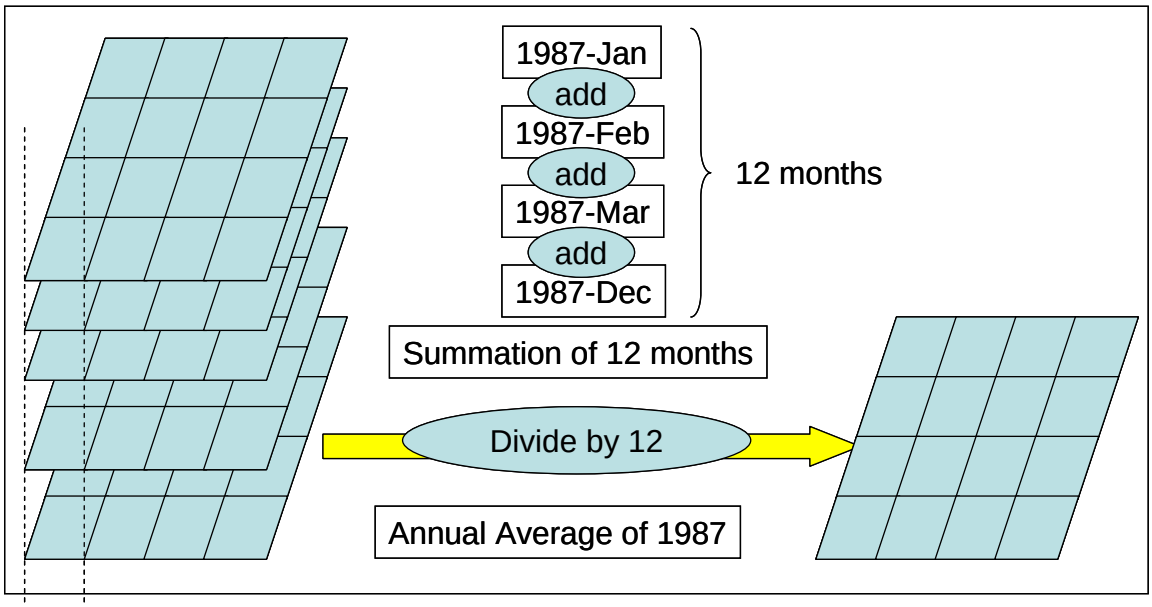


Figure 4-2 Conversion of monthly data to yearly data

[24] Standard input and output data files for the global water resources model H08 also use such names.

To perform this operation, you will need a special command that adds together the binary files and performs the division. You can use the commands shown in Table 4-2.[25] When using these commands, you must specify which files to read from and which files to write to. For example, use the createcmap command to create a single new binary file called temp.cmap, in which 144 columns and 72 rows are filled with real numbers with the value 0.

```
% createcmap temp.cmap 0
```

In this case, the information “temp.cmap” and “0” is passed to the createcmap command. These are arguments. The orders of arguments in the commands in Table 4-2 must not be altered.

Table 4-2 Commands for controlling binary files (1)

Program	Arguments	Function
createcmap	OUT, num	Creates the binary file OUT, filled with real numbers with the value num
addcmap	IN1, IN2, OUT	Calculates sums between two binary files (IN1 and IN2) and writes to a binary OUT
subcmap	IN1, IN2, OUT	Calculates differences between two binaries (IN1 minus IN2) and writes to a binary file OUT
mulcmap	IN, num, OUT	Multiplies a binary file (IN) by a real number value (num) and writes to a binary file OUT
divcmap	IN, num, OUT	Divides a binary file (IN) by a real number value (num) and writes to a binary file OUT
procmap	IN1, IN2, OUT	Calculates products between two binary files (IN1 and IN2) and writes to a binary file OUT
ratcmap	IN1, IN2, OUT	Calculates quotients between two binary files (IN1 / IN2) and writes to a binary file OUT
asc2cmap	IN, OUT	Converts text (IN) to a binary (OUT)
cmap2asc	IN, OUT	Converts a binary (IN) to text (OUT)
xyz2cmap	IN, OUT	Converts text in a “long lat data” format (IN) to a binary (OUT)
cmap2xyz	IN, OUT	Converts a binary (IN) to (long lat data) text (OUT)
shiftcmap	IN, OUT	Shifts a binary (IN) 180º in the east–west direction and writes to a binary OUT
upsidedowncmap	IN, OUT	Flips a binary (IN) between north and south and writes to a binary OUT

[25] This set of commands is called Htool. It is used in processing of standard input and output files for the global water resources model H08. See H08 manual for detail.

cmap2eps	IN, CPT, OUT, [title1], [title2]	Converts a binary (IN) to an EPS image (OUT)
eps2gif	IN, OUT [rot]	Converts an EPS image (IN) to a different image format (OUT). Add rot to rotate 90 degrees.
mon2yearcmap	DIR, PRJ, RUN, yearmin, yearmax, suf	Converts data by month to data by year
nofday	year mon	Returns the number of days in month (mon) of year (year)

In these commands:
IN is an input file; OUT is an output file; CPT is a CPT file;
num is a real number; year and mon are integers representing a year and a month;
[title1] and [title2] are title strings (the arguments in brackets [ ] may be omitted);
yearmin and yearmax are 4-digit integers representing a start year and an end year (in the Western calendar);
DIR is the directory a file is in (./ if the file is in the current directory);
PRJ is a 4-character text string representing a project title (CMAP in this study text);
RUN is a 4-character text string representing a run (or experiment, version) title (203_ in this study text); and
suf (suffix) is a string representing a filename extension (.cmap in this study text).

Now you can use the commands in Table 4-2 to proceed with the task. First create a directory called BINARY in your home directory, and then copy the binary data for 1987 and 1988 into this directory.

```
% mkdir ~/BINARY
% cd ~/BINARY
% cp ~/unix_semi/dat_bin/*1987* .
% cp ~/unix_semi/dat_bin/*1988* .
```

As it happens, files for 1987 and 1988 annual averages are also present in ~/unix_semi/dat_bin/. You have just made copies of these along with the monthly files, so delete those two copies.

```
% rm CMAP203_19870000.cmap
% rm CMAP203_19880000.cmap
```

Now continue the operations shown in Figure 4-2. First, make a new file for average values over the year 1987 called CMAP203_19870000.cmap.

```
% createcmap CMAP203_19870000.cmap 0
```

CMAP203_19870000.cmap is a binary file with 144 columns and 72 rows of real values, all of which are currently filled with the value 0. Add the data for January 1987 to this file.

```
% addcmap CMAP203_19870000.cmap CMAP203_19870100.cmap CMAP203_19870000.cmap
```

Then add the data for February to the file.

```
% addcmap CMAP203_19870000.cmap CMAP203_19870200.cmap CMAP203_19870000.cmap
```

Now CMAP203_19870000.cmap is filled with sums for January and February. Continue the process up to December.

```
% addcmap CMAP203_19870000.cmap CMAP203_19871200.cmap CMAP203_19870000.cmap
```

Finally, divide by 12.

```
% divcmap CMAP203_19870000.cmap 12 CMAP203_19870000.cmap
```

Thus, you have obtained average values for 1987. Try to create average values for 1988 in the same manner.

When you have done this, you have calculated year averages by adding together the files for the 12 months and dividing by 12. However, this method is not strictly accurate because the numbers of days in the months are not all the same. To perform the calculation more accurately, you could multiply the precipitation values by the number of days in each month, like this.

```
% mulcmap CMAP203_19870100.cmap 31 temp.cmap
```

Then you could add up the months.

```
% addcmap CMAP203_19870000.cmap temp.cmap CMAP203_19870000.cmap
```

And finally you could divide by the number of days in the year.

```
% divcmap CMAP203_19870000.cmap 365 CMAP203_19870000.cmap
```

Taking account of the number of days in each month and of leap years is troublesome. However, there is a command called mon2yearcmap that accurately converts data by month to data by year. You can calculate the year averages accurately for 1987 and 1988 with the following command.

```
% mon2yearcmap ./ CMAP  203_ 1987  1988  .cmap
```

4.3 Data Deviation and Mapping

Now you will calculate the differences between 1987 and 1988. Subtract the 1988 data from the 1987 data, as shown in Figure 4-3, and write the results to a file called dif.cmap.

```
% subcmap CMAP0203_19870000.cmap CMAP0203_19880000.cmap dif.cmap
```

Make a map of these results. Make a CPT file on your own called dif.cpt to express the differences. Then use the cmap2eps command to add the title “Difference” and the subtitle “1987-1988”.

```
% cmap2eps dif.cmap dif.cpt dif.eps Difference 1987-1988
```

Does your map of the differences look like Figure 4-3?

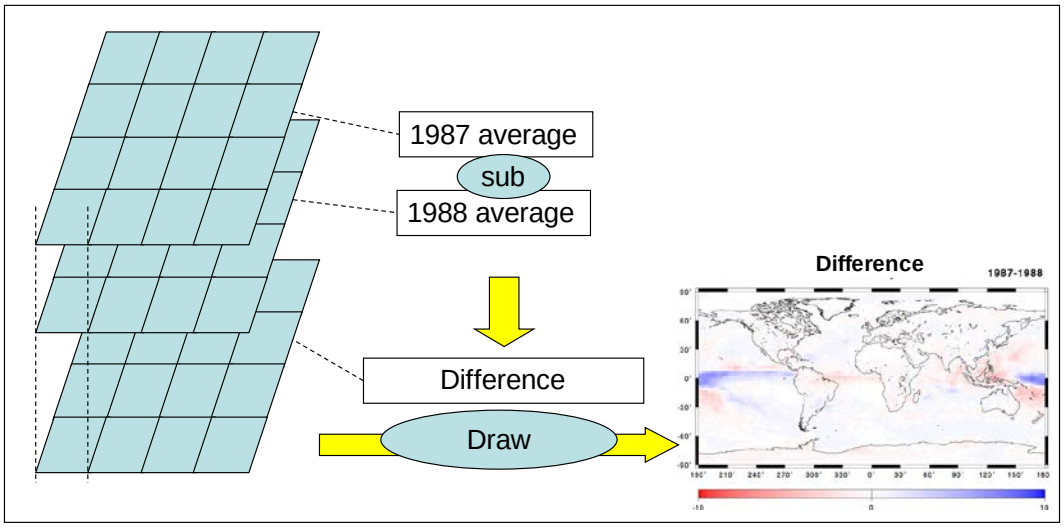


Figure 4-3 Calculating difference between 1987 and 1988

Exercises

1. Create precipitation data for JJA (June-July-August) and DJF (December-January-February) of 1987 and 1988, and create maps showing the differences (make your own original CPT files). Note that DJF 1987 means from Dec 1986 to Feb 1987, and DJF1988 from Dec 1987-Feb 1988.
2. The method in the example above produces maps in which the differences are more obvious in regions with high rainfall (such as tropical oceans). No differences can be seen in regions with low rainfall, such as the Sahara desert, while the differences appear large in high-rainfall tropical regions, because small values are being subtracted from relatively small values in the dry regions and large values are being subtracted from relatively large values in the wet regions. Draw a map showing the differences between precipitation levels in 1987 and 1988 as ratios based on the 1987 precipitation levels. For example, if the year average is 5 mm/day in 1987 and 6 mm/day in 1988, the ratio is (5 – 6) / 5 = –0.2. Create an original CPT file and express these differences on a map.

(Solutions to the exercises are in ~/unix_semi/lesson4.)

If the results show strange numbers (e.g. extremely large or small), see Appendix B, and try changing the Endian of binary.

Chapter 5

Fortran (1): Fortran Input and Output

In Chapter 4, you used commands listed in Table 4-2 to perform calculations with binary files. I made those commands myself, using the programming language called Fortran. This chapter will explain how to use Fortran to read and write files.

Fortran was originally developed in the 1950s but its formats and functions have changed over the decades since. It is often thought of as an old-fashioned programming language, but it is still widely used in scientific fields. In particular, many important programs that are used around the world in the field of hydrometeorology, such as climate models, land surface process models, and river models, are written in Fortran.

Although this study text uses Fortran, you can use whichever programming language you prefer when you do your own programming, be it C, C++, C#, Java, Ruby, or some other. Note that only techniques that are specific to or characteristic of Fortran will be introduced in this chapter. If you wish to learn more, consult a Fortran study text.

5.1 Basics of Fortran Source Code

Here is some Fortran source code. This is a program that outputs a text, such as “Hello, World”, to the terminal.

```
1:     program hello
2:c
3:c this is comment.
4:c
5:     write(6,*) ‘Hello, World’
6:     write(6,*) ‘A very very long sentence such as this
7:    $     must be divided in two lines’
8:c
9:     end
```

Line 1 declares that a program called “hello” is starting. The term program is a start declaration instruction, and must be written in the first line of any Fortran source code. In Fortran source code, the actual program is written from the 7th column to the 72nd column. In this example, six spaces are left at the start of line 1 before program hello is written. Integers, specifically meaning line numbers, can be put in the 2nd to 5th columns, but these are almost never used nowadays. If you wish to know how to use them, consult a Fortran study text.

Lines 2 to 4 are comments. When you want to write a comment, put a c or * in the first column and then the comment.

Line 5 is an instruction to write “hello world” to the terminal. The term write is an instruction to write something out. In the parentheses, the first parameter is a “device number” and the other parameter is a “format”. The device number is a positive integer indicating where the message should be written. Device numbers are defined by users, except 5 and 6, which are set to a standard input (keyboard) and a standard output (terminal), respectively. The format * means automatic formatting. After that, the text string between the quote marks ‘ and ’ is the text string to be outputted.

Line 6 and line 7 are an instruction to write “A very very long sentence such as this must be divided in two lines” to the terminal. Because instructions can only be written up to column 72, this instruction must be divided between two lines. When inputting an instruction over two lines, type a character, other than “0”, into column 6 of the second line. In this example, the dollar sign is used.

5.2 Compiling Fortran Source Code

As practice, create a text file exactly the same as the example above with Emacs and save it with the name “example1.f”. This file cannot be executed as it is. You have to use a special command known as a Fortran compiler to convert it to executable machine code. This operation is referred to as compiling Fortran source code. If your UNIX computer is configured as in Appendix C, you should be able to use the Fortran compiler called “gfortran”. Use the following command to compile the Fortran source code in example1.f.

```
% gfortran example1.f
```

This creates an executable file called “a.out”. Execute a.out.

```
% a.out
```

The following should be displayed in the terminal.

```
Hello, World
A very very long sentence such as this must be divided in two lines
```

All files produced by compiling source code have the name a.out. Obviously, this could make compiled source codes difficult to identify. However, you can change the name of the executable file that is produced by adding the option -o and following it with a filename. For example, the following command produces an executable file called example1.

```
% gfortran example1.f –o example1
```

This works in most other Fortran compilers as well as gfortran.

5.3 Input and Output of Fortran Files

One of the commands presented in Table 4-2 in Chapter 4 was cmap2asc. This command is a program that converts binary data to ASCII data. For example, to convert the binary file for January 1987 that you used in Chapter 4 to a text file with the name "temp.txt", you would do the following.

```
% cd ~/BINARY
% cmap2asc CMAP203_19870100.cmap temp.txt
```

The source code of cmap2asc is in ~/unix_semi/src/cmap2asc.f.

In this section, the source code of this program is divided into four blocks and described.

```
 1:     program cmap2asc
 2:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 3:cto   Convert binary (144x72) to ascii
 4:con   30th/May/2002
 5:cby   nhanasaki
 6:cat   IIS, UT
 7:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 8:     implicit none
 9:c
10:     integer         nx
11:     integer         ny
12:     parameter       (nx=144)
13:     parameter       (ny=72)
14:c
15:     integer         i,j
16:     real            rdat(nx,ny)
17:     character*128    cifname
18:     character*128    cofname
19:c
20:     integer         iargc
```

In the first block, first of all, line 1 declares the start of the program. Where program XXXXXX is written, this declares that the following code is a program called XXXXXX.

Lines 2 to 7 are comments, giving the programmer's name, the creation date, and so forth.

Line 8 says implicit none, which means that no "implicit variables"[26] are used. You should always add this line.

The descriptions in line 10 to line 20 are declarations of types of variables. You studied variables and assignment of values in Bourne Shell scripts in Chapter 3. Values can be assigned to variables in the same way in Fortran, but are more tightly specified. The variables (the empty boxes of Chapter 3) are declared here. The term integer is used for declaring an integer variable, real for a real number variable, and character for a

[26] An implicit variable is a variable for which there is no declaration of the variable type. Fortran allows implicit variables whose names start with characters in the range i–n are treated as integers and others are treated as real numbers. At first glance, leaving out variable declarations may seem convenient, but it tends to produce unforeseen effects and bugs, so it is better not to use implicit variables.

string of characters. The declarations of the variables can be understood as setting types and sizes of the boxes into which numbers and characters are to be put, as in Figure 5-1. For example, integer nx prepares a box called nx into which just one integer can be put, and character*128 cifname prepares a box called cifname into which a text string of up to 128 characters (character*128) can be put. The variable rdat(nx,ny) can be understood as a box that holds real numbers arranged into nx columns and ny rows. This kind of variable, a variable to which more than one value can be assigned, is called an array.

In line 12 and line 13, parameter assigns a value to a variable that will be a constant. Being constants, these values cannot be overwritten during execution of the program. In this case, nx and ny must be assigned as constants because they are used in the declaration of the array rdat.

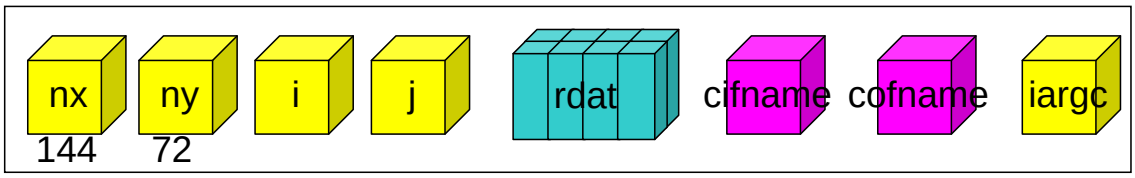


Figure 5-1 Types of variable

```
 1:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 2:c Get file name
 3:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 4:      if (iargc().ne.2) then
 5:        write(6,*) 'Usage: cmap2asc Infile Outfile'
 6:        stop
 7:      end if
 8:c
 9:      call getarg(1,cifname)
10:      call getarg(2,cofname)
```

In the second block, file names are read in.

In line 4, the condition in parentheses after the if is tested. If it is “true”, the instructions in the following lines are executed. Table 5-1 shows how to write conditions. Here, iargc() is a function that returns the number of arguments provided with a command (the command cmap2asc in this example). Thus, this condition means “true if the number of arguments is not equal to 2”. If it is true, the instructions in the following lines are executed, and if it is false, those instructions are not executed but the instructions following end if are executed.

Line 5 is a write statement, which you have already studied. Recall that the contents of the parentheses specify the device number and format, with the device number 6 representing the terminal and the format * representing an automatic setting. The writing format may be more specifically specified; consult a Fortran study text if you wish to learn more. The character string that is to be written is put between the quote marks. In line 6, stop means that the program ends.

Thus, if lines 4 to 7 were to be written in ordinary English, they would be: “If the number of arguments is not 2, output ‘Usage: cmap2asc Infile Outfile’ to the terminal in the default format and end the command. Otherwise (i.e., if the number of arguments is 2), carry on.”

Line 9 and line 10 read in the first argument and the second argument, respectively. The instruction call getarg(1,cifname) reads the first argument and assigns it to the variable cifname. You can see in Table 4-2 that the command cmap2asc requires two arguments—an input filename and an output filename. If line 9 and line 10 were written in ordinary English, they would be: “Assign the first argument to the variable cifname and assign the second argument to the variable cofname.”

Table 5-1 Types of conditional statement

a.gt.b	a is greater than b (true if a is greater than b)
a.ge.b	a is greater than or equal to b
a.lt.b	a is less than b
a.le.b	a is less than or equal to b
a.eq.b	a is equal to b
a.ne.b	a is not equal to b

```
1:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
2c Read Data
3:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
4:      open(15, file=cifname, access='direct', recl=nx*4)
5:      do j=1, ny
6:        read(15,rec=j) (rdat(i,j),i=1,nx)
7:      end do
8:      close(15)
```

In the third block, values are read from the binary file. In line 4, the file is opened. The open statement uses the form open(device number, file=filename, access=‘access method’, recl=record length).

As has already been mentioned, a device number is a positive integer representing a subject of operation. In this case, it is an integer to be assigned to the file that is opened. The access method can only be ‘direct’ or ‘sequential’. The former is for reading binary files and the latter is for reading ASCII files. The record length is a positive integer, which only needs to be specified when reading a binary file with access=‘direct’. The record length sets how many bytes of data are to be read (or written) in one read (or write) operation. As shown in Figure 5-2, CMAP is two-dimensional data with nx columns and ny rows, and nx columns of real numbers are stored in an arbitrary row j. Because the data for one real number is four bytes, the record length is nx×4 bytes.

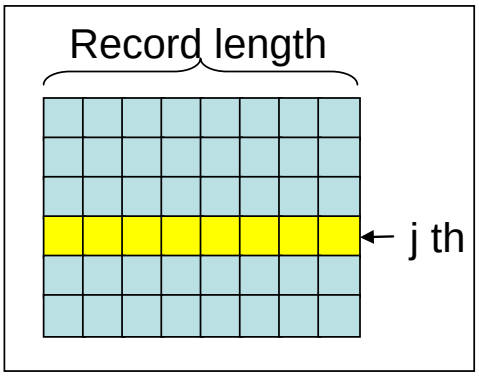


Figure 5-2 Record length of binary files of CMAP data

Line 5 represents repetitive processing, called a “do loop”. In this case, j is increased from 1 to ny in increments of 1, and the instruction between do and end do is repeated ny times.

Line 6 reads variables from the file. This read statement has the form read (device number, record to be read) variables to which the read values are assigned. The expression (rdat(i,j),i=1,nx) means increasing i from 1 to nx in increments of 1 and, nx times, reading a value into rdat.

Line 8 closes the file. A close statement uses the form close(device number). You must always close the file after finishing this kind of operation.

If the contents of the third block were expressed in ordinary English, they would be as follows:

[Lines 1 to 3] Comments

[Line 4] Open the file cifname and assign to it the device number 15. This file is written as a binary (direct access). The record length is nx*4 (=144*4=576) bytes.

[Line 5] Change j from 1 up to ny. (In other words, do the processing up to end do ny times.)

[Line 6] Read record number j from the file assigned to device number 15. Put the data into the array rdat.

[Line 7] Upon reaching end do, return to do and repeat.

[Line 8] Close the file assigned to device number 15.

```
 1:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 2:c Write Data
 3:cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 4:      open(16, file=cofname, access='sequential')
 5:      do j=1, ny
 6:        write(16,*) (rdat(i,j), i=1,nx)
 7:      end do
 8:      close(16)
 9:c
10:      end
```

The fourth block performs operations to write the values assigned to the variable rdat in ASCII. The open statement in line 4 opens the file cofname. The device number that

is assigned is 16, and the access method is set to sequential. Because the access method is not direct, there is no need to set a record length.

If the fourth block was expressed in ordinary English, it would be as follows:

[Lines 1 to 3] Comments

[Line 4] Assigning 16 as the device number, open the file cofname. This file is written as ASCII.

[Line 5] Change j from 1 up to ny.

[Line 6] Write from the array rdat to the file assigned to device number 16, changing i from 1 up to nx. The writing format is the default.

[Line 7] Upon reaching end do, return to do and repeat.

[Line 8] Close the file assigned to device number 16

[Line 10] End the program.

Exercises

1. Read the Fortran source code for addcmap in Table 4-2, and rewrite it in ordinary language. The source code is in ~/unix_semi/src/addcmap.f.
2. Create a program called asc2cmap.f with the opposite function to cmap2asc.f. (When you have finished writing it, try to compile and execute it.)

(Solutions to the exercises are in ~/unix_semi/lesson5.)

Chapter 6

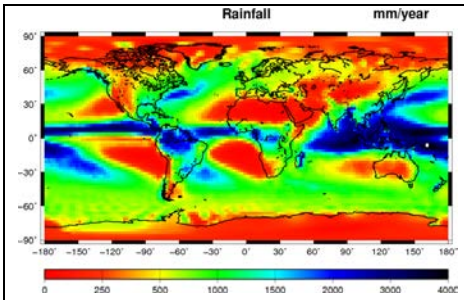
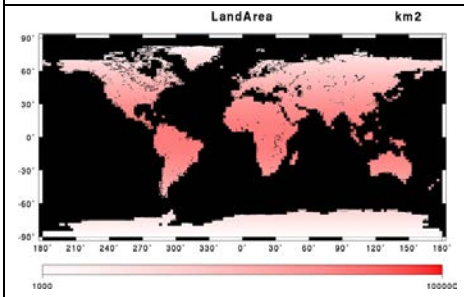
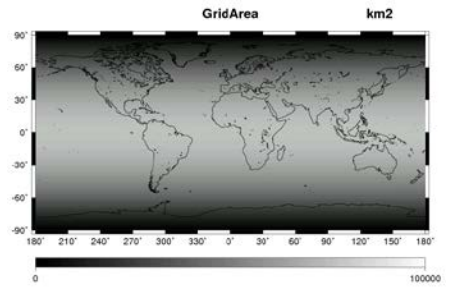
Manipulating Grid Data (2): Masking With a Fortran Program

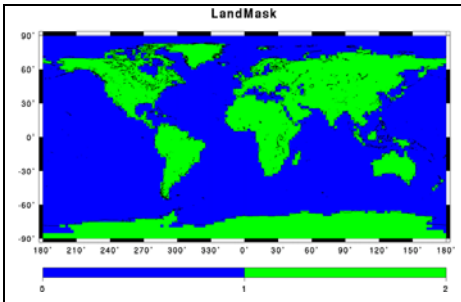
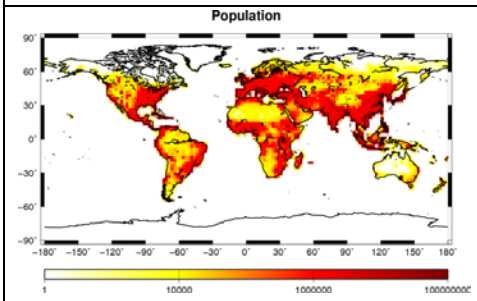
In this chapter, you will use the CMAP data to investigate precipitation levels by year for Tokyo and average precipitation levels by year for the world. You will also try to find out how much of the world has higher annual precipitation levels than Tokyo, and how many people live in areas with less than 500 mm/year of precipitation.

6.1 Data Used in This Chapter

The data used in this chapter is shown in Table 6-1, and is in ~/unix_semi/lesson6.

Table 6-1 Data used in this chapter (binary files)

	**Yearly average precipitation levels (1979–2001)** Filename: prc.cmap Unit: mm/year Resolution: global, 2.5º×2.5º Notes: Created from CMAP data
	**Surface areas (area/grid cell)** (land only) Filename: lndara.cmap Unit: km^2 Resolution: global, 2.5º×2.5º
	**Surface areas (area/grid cell)** (whole world) Filename: grdara.cmap Unit: km^2 Resolution: global, 2.5º×2.5º

	**Land/sea map** Filename: lndmsk.cmap Unit: none Resolution: global, 2.5º×2.5º Notes: 1 for land, 0 for sea
	**Population** Filename: pop.cmap Unit: persons Resolution: global, 2.5º×2.5º Source: CIESIN GPW2[27]

6.2 Commands for Controlling the CMAP Binary Files

You studied some commands for controlling the CMAP binary files in Chapter 4. This section will describe more commands (see Table 6-2).

Table 6-2 Commands for controlling binary files (2)

Program	Arguments	Function
avecmap	IN, miss	Calculate the average value of data in the binary IN, excluding miss.
maxcmap	IN, miss	Find the maximum value of data in the binary IN, excluding miss.
mincmap	IN, miss	Find the minimum value of data in the binary IN, excluding miss.
sumcmap	IN, miss	Calculate the total sum of data in the binary IN, excluding miss.
pointcmap	IN, lon, lat	Display data for the position with longitude lon and latitude lat
findcmap	IN, sign, num, OUT	Display data in the binary IN that has the relationship sign to the value num. Write this data to the binary OUT, replacing other data with 0.
rplccmap	IN, sign, num1, num2, OUT	Write data in the binary IN to the binary OUT, changing to num2 those data with the relationship sign to the value num1.
maskcmap	IN, MASK, sign, num, OUT	Where data in the binary MASK has the

[27] http://sedac.ciesin.columbia.edu/gpw/

		relationship sign to the value num, write data in the binary IN to the binary OUT, replacing other data with 0.
maskrplccmap	IN, MASK, sign, num1, num2, OUT	Write data in the binary IN to the binary OUT, changing the data to num2 where data in the binary MASK has the relationship sign to the value num1.
IN, MASK, and OUT are binary files. sign is a conditional relationship selected from eq (equal to), ne (not equal to), gt (greater than), ge (greater than or equal to), lt (less than), and le (less than or equal to). num, num1, and num2 are real numbers; and lon and lat are real numbers representing longitude and latitude. miss is a real number ("missing value").[28]		

In Table 6-2, findcmap, rplccmap, maskcmap, and maskrplccmap are a little difficult to understand, so they will be illustrated with examples. First, findcmap is a command that extracts data meeting a certain condition. Look at the example in Figure 6-1a From a binary file IN filled with data from 1 to 16, as shown on the left, the following command extracts numbers greater than (gt) 8 and writes them to the binary file OUT.

```
% findcmap IN gt 8 OUT
```

Next, rplccmap is a command that changes data meeting a certain condition to another real number. Real values of data that does not meet the condition are outputted as is. In the example in Figure 6-1b, from the binary file IN filled with data from 1 to 16, the following command replaces numbers greater than (gt) 8 with 1 and writes the results to the binary file OUT.

```
% rplccmap IN gt 8 1 OUT
```

The command maskcmap judges where a separately prepared binary file MASK (illustrated in Figure 6-1c; this is called a mask file) meets a condition and, where the condition is met, extracts data from the binary file IN and writes the data to the binary file OUT. In the example in Figure 6-1d, from the binary file IN filled with data from 1 to 16, the following command extracts data at positions where data in the binary file MASK equals (eq) 1 and writes the data to the binary file OUT.

```
% maskcmap IN MASK eq 1 OUT
```

[28]The missing value is a real number representing a particular number to be excluded from calculations. For example, in data that should only have zeros and positive values (for example, population or rainfall), a value such as –999 may be used to clearly show where data is missing. In meteorology, 1.0E20 is often used. In the case of CMAP data, however, there is no missing data at any period or in any grid cell, so a missing value need not be specified.

Lastly, the command maskrplccmap judges where the separately prepared binary file MASK meets a condition and writes data in the binary file IN to the binary file OUT, changing the data to another real value where the condition is met. In the example in Figure 6-1e, from the binary file IN filled with data from 1 to 16, the following command replaces data at positions where data in the binary file MASK equals (eq) 1 with 1, and writes the results to the binary file OUT.

```
% maskrplccmap IN MASK eq 1 1 OUT
```

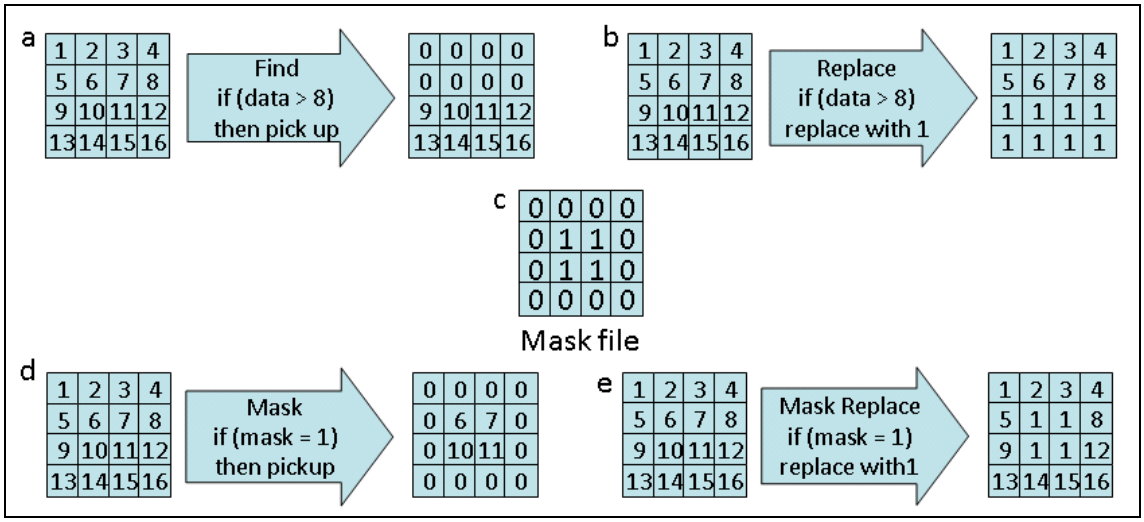


Figure 6-1 Concept of findcmap, rplccmap, maskcmap, maskrplccmap

Example 1

Find annual precipitation for Tokyo (138.75° East, 36.25° North[29]).

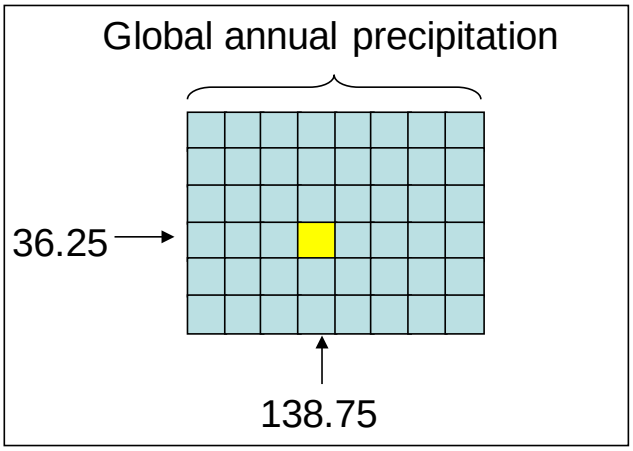


Figure 6-2 Concept of pointcmap

[29] The location represented by this longitude and latitude is actually at Takasaki city in Gunma prefecture. The individual grid cells in CMAP are quite large (about 280km x 280km at equatorial regions) and this is the grid node closest to Tokyo.

First of all, consider which data from Table 6-1 you could use and which commands from Table 6-2 you could use. How about applying pointcmap to the yearly average precipitation data (prcp.cmap)? Try typing in the following command.

```
% pointcmap prcp.cmap 138.75 36.25
```

With this command, you can read the binary file prcp.cmap and pick out the data in the grid cell nearest Tokyo (36.25 N, 138.75 E), as shown in Figure 6-2. If you are interested, look at ~/unix_semi/src/pointcmap.f and study the source code.

Example 2

Find the yearly average precipitation for the world.

Now find a yearly average precipitation level for the whole world. Consider which data from Table 6-1 you could use and which commands from Table 6-2 you could use. Applying avecmap to the yearly average precipitation data (prcp.cmap) would seem likely. Type in the following command.

```
% avecmap prcp.cmap -999
```

With this command, you can add up all the data in the binary file prcp.cmap and divide by the number of data points to calculate the average value. A missing value (–999 in this example) is excluded from the addition of the data. In fact, this calculation is not really accurate, but you will be addressing this point in the exercises.

Example 3

Find what areas in the world have higher precipitation levels than Tokyo.

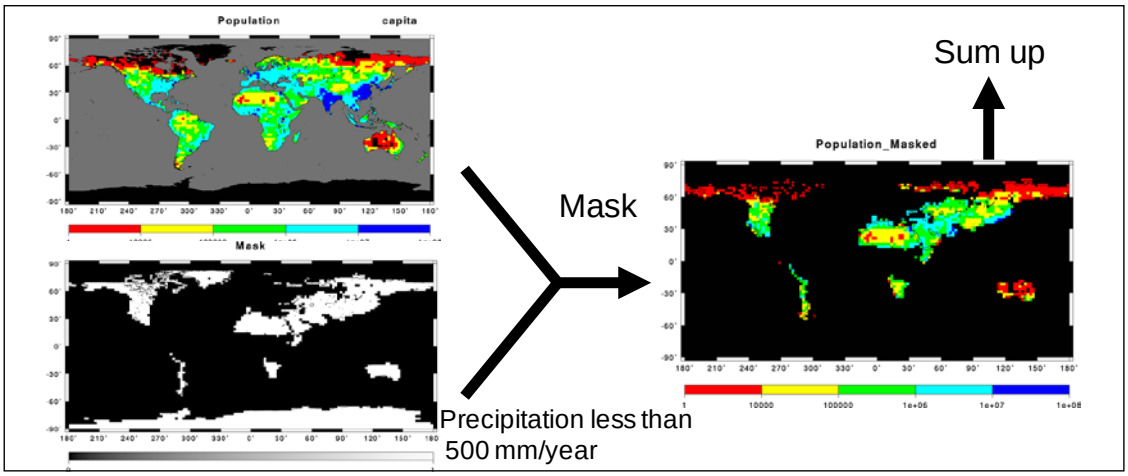


Figure 6-3 Calculation in Example 4

In Example 1 you should have found that the yearly precipitation for Tokyo is 1530 mm/year. Therefore, you should first extract land regions where the yearly precipitation data (prcp.cmap) is at or above 1530 mm/year from the area data (lndara.cmap).

```
% maskcmap lndara.cmap prcp.cmap ge 1530 temp.cmap
```

Then calculate the total of the extracted area data.

```
% sumcmap temp.cmap 0
```

Example 4

Find the population living in regions with yearly precipitation of less than 500 mm/year.

Similarly to Example 3, extract regions in which the precipitation data (prcp.cmap) is less than 500 mm/year from the population data (pop.cmap).

```
% maskcmap pop.cmap prcp.cmap lt 500 temp.cmap
```

Then find a total population of the extracted grid cells (See Fig 6-3).

```
% sumcmap temp.cmap 0
```

Exercises

1. Calculate the yearly average precipitation for the world more accurately. In Example 1 you used the command avecmap to find the yearly average precipitation, but this takes a simple average of all the data. However, the world being round, the areas of the globe represented by the grid cells are smaller at higher latitudes (see the area data in Table 6-1). Therefore, you should apply weightings by area to find the average value. Think of how the data should be processed, and then think of how to combine the commands in Table 6-2 to perform this processing. Hint: the land surface area data (grdara.cmap) in Table 6-1 and procmap in Table 4-2, which multiplies together binary files, will be useful. At one point in this exercise, you will have to divide a certain value by a certain other value. Do this by hand, using a calculator.
2. If yearly average precipitations on land are categorized as in the following table, how much area does each of these regions have? Moreover, how many people are living in each of these regions? Fill out the following table.

Precipitation (mm/year)	Population (persons)	Area (km^2)
0–50		
50–100		
100–500		
500–1000		
1000–2000		
2000–		
Total		

(Solutions to the exercises are in ~/unix_semi/lesson6.)

Chapter 7

Fortran (2): Input and Output of Time Series Data in Fortran

Chapter 7 will describe Fortran programming techniques for efficiently processing time series binary files. As mentioned in Chapter 5, you can continue study of the basic grammar of Fortran with the references in Appendix D. The source code introduced in this chapter is in ~/unix_semi/src/. Refer to it as necessary.

7.1 Input and Output of Time Series Files in Fortran

In Chapter 4, you studied the conversion of monthly binary files to years. You used the command mon2yearcmap for this processing. The command mon2yearcmap reads data from January to December for each year, from the start year to the end year that are provided as arguments, and outputs averaged data for each year. In order to take account of the numbers of days in the months, the data for each month is multiplied by the number of days in the month when the data for that month is being added, and the final sum is divided by the number of days in the year (365, or 366 in a leap year). Open the source code, which is in ~/unix_semi/src/mon2yearcmap.f. In this chapter, this source code is divided into four blocks and explained.

```
 1:      program mon2yearcmap
 2: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 3:      implicit none
 4: c parameter
 5:      integer         nx, ny
 6:      parameter       (nx=144, ny=72)
 7: c variable
 8:      integer         ix, iy
 9:      integer         iyear, iyearmin, iyearmax, imon
10: c temporary
11:      character*128   ctmp
12: c function
13:      integer         iargc          !! Default function
14:      integer         igetday        !! Original function
15:      integer         len_trim       !! Original function
16: c in
17:      real            rmondat(nx,ny) !! Monthly data array
18: c out
19:      real            ranudat(nx,ny) !! Annual data array
20: c local
21:      integer         inofdaymon     !! Number of days in a month
22:      integer         inofdayyear    !! Number of days in a year
23: c files
24:      character*128   cdir, cexp, crun, csuf
25:      character*128   cifname, cofname
26:      integer         ilen
27:      character*128   clen
```

In the first block, variables are declared.[30] The text strings after !! in lines 13 to 22 are comments. Standard Fortran grammar does not have a way of starting comments partway along a line, but !! can be used with a great many compilers. This feature is convenient, so use it generously to make your code easier to understand.

```
 1: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 2: c Get Arguments
 3: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 4:      if(iargc().ne.6)then
 5:        write(*,*) 'mon2year dir exp run yearmin yearmax suf'
 6:        stop
 7:      end if
 8: c
 9:      call getarg(1,cdir)
10:      call getarg(2,corg)
11:      call getarg(3,crun)
12:      call getarg(4,ctmp)
13:      read(ctmp,*) iyearmin
14:      call getarg(4,ctmp)
15:      read(ctmp,*) iyearmax
16:      call getarg(5,csuf)
17: c
18:      ilen=len_trim(cdir)
19:      write(clen,*) ilen
```

The second block reads in the arguments. This program requires six arguments. The variables to which the arguments are assigned are character string variables called cdir, corg, crun, and csuf, representing a directory name, a project title, an experiment title, and a filename extension; and integer variables iyearmin and iyearmax, representing the start year and the end year. One of the restrictions of Fortran grammar is that arguments can only be character strings. Therefore, the start year and the end year are each temporarily read into a character string variable called ctmp as a character string, using call getarg, and then read into the integer variables iyearmin and iyearmax using read statements (line 13 and line 15). As you saw in Chapter 5, read and write statements are basically used in the forms write(device number,format) and read(device number,format).[31] In this case, the device number is the variable name, which means that the variable is being read or written.

Lines 18 and 19 count the number of characters in the value assigned to the variable cdir. This will be used in a later part of this program. Line 18 looks as though a one-dimensional array called len_trim is being assigned to a variable ilen. It actually means that a function called len_trim is called and provided with cdir as an argument. You can think of a function as a command within a command. As in this example, a function behaves like a single value (an integer, a real number, or a character string). The function len_trim calculates the number of characters in the argument (the source

[30] Refer back to Chapter 5 for variable declarations.

[31] In a read statement, the device number indicates where to read from. If a device number is 15, it represents reading from a file which was assigned a device number 15. In a write statement, the device number indicates where to write to. If a device number is 16, it represents writing to a file which was assigned a device number 16.

code is available at ~/unix_semi/src/len_trim.f). For example, if cdir='/home/naota', then len_trim(cdir)=11. In line 19, the integer variable ilen is converted to a different type of variable by being written to a character string variable clen. In the above example, this would convert the integer 11 (eleven) to the character string "11" (one-one).

```
 1: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 2: c Sum up
 3: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 4:      do iyear=iyearmin,iyearmax
 5: c clear variable
 6:       inofdayyear=0
 7: c clear array
 8:       do iy=1,ny
 9:         do ix=1,nx
10:           ranudat(ix,iy)=0.0
11:         end do
12:       end do
13: c
14:       do imon=1,12
15:         inofdaymon=igetday(iyear,imon)
16:         inofdayyear=inofdayyear+inofdaymon
17: c read data
18:         write(cifname,'(a'//clen//',a4,a4,i4.4,i2.2,i2.2,a5)')
19:     $    cdir,corg,crun,iyear,imon,0,csuf
20:        call read_binmat(rmondat,nx,ny,15,cifname)
21: c calclulation
22:         do iy=1,ny
23:           do ix=1,nx
24:            ranudat(ix,iy)
25:    $       =ranudat(ix,iy)+rmondat(ix,iy)*real(inofdaymon)
26:          end do
27:        end do
28:      end do
```

Reset values assigned to variables and arrays.

The third block and the fourth block use do loops to repetitively perform calculations. The do iyear=iyearmin,iyearmax in line 4 of the third block has an end do at line 12 of the fourth block. Thus, the processing between those two lines is carried out repeatedly.

The part enclosed by the dashed line will be described later. For now, look at do imon = 1,12 in line 14. The program loops with the variable iyear changing from iyearmin to iyearmax, and loops with the variable imon changing from the value 1 to the value 12. Thus, processing can be performed sequentially from January of iyearmin to December of iyearmax.

Line 15 looks as though it is reading a two-dimensional array called igetday, but is actually calling a function igetday and providing the two arguments iyear and imon. This function calculates how many days there are in a particular month of a particular year (the source code is available at ~/unix_semi/src/igetday.f). The function takes proper account of leap years when dealing with a February. For example, igetday(1988,2) returns the value 29.

Line 18 and line 19 are a write statement. The device number of this write statement is the character string variable cifname, which means that the result is to be written into the variable cifname. Up to now, only the * representing automatic settings has been

used in write statements, but in this case a manual setting is used. The character string enclosed in the parentheses inside the single quote marks—that is, a'//clen//',a4,a4,i4.4,i2.2,i2.2,a5—represents a format. This represents writing a string of clen characters: a string of four characters, another four-character string, a four-digit integer, a two-digit integer, a two-digit integer, and a five-character string. The initial a'//clen//' is a very unusual expression, representing a character string whose length is clen[32]; for example, a four-character string if clen is 4 or an eight-character string if clen is 8. The term a4 represents four characters (ascii), and i4.4 represents a four-digit integer that must be padded out to four digits.[33] The commas between the terms, as in "a4,a4" are there to aid understanding and can be ignored. In this manner, a write statement can be used to assign a character string with a particular format to a variable (in this case, a character string such as CMAP203_19870100.cmap was assigned to a variable cifname).

Line 20, after the call statement, looks as though it is writing a five dimensional array called read_binmat, but actually means that a subroutine called read_binmat is being called with five arguments. Think of a subroutine as another kind of command within a command. This subroutine has the function of reading the nx×ny array from the binary file whose filename is cifname, with the device number 15, and assigning the results to rmondat. It is described in more detail in the next section.

Line 24 and line 25 perform an operation that adds the data for each month into the variable ranudat.

Line 28 is the end do corresponding to do imon = 1,12, and ends that loop.

Finally, the part enclosed in the dashed line that was left out of the explanation will now be explained. When processing is repeated using a do loop, there are old values in arrays and variables. For example, if this calculation is performed from 1987 to 1988, firstly total values for 1987 are put into the variable ranudat. If nothing is done, these values will still be there for the processing of 1988, and the totals for 1987 and 1988 will be added together. Therefore, before performing the first addition in each year, the values in the array and the variable must be cleared. This operation is very important in processing for summing numbers.

[32] The symbol // means that a character string is joined to another character string. Formats must be described by character strings, so the character string variable clen is used instead of the integer variable ilen.

[33] For example, 25 is written as 0025.

```
 1: c Get average
 2:       write(*,*) 'Total num of days:', ndayyear
 3:       do iy=1,ny
 4:         do ix=1,nx
 5:           ranudat(ix,iy)=ranudat(ix,iy)/ndayyear
 6:         end do
 7:       end do
 8: c Write results
 9:       write(cofname,'(a'//clen//',a4,a4,i4.4,i2.2,i2.2,a5)')
10:     &  corg,crun,iyear,0,0,csuf
11:       call wrte_binmat(ranudat,nx,ny,16,cofname)
12:      end do
13: c
14:      end
```

In the fourth block, line 2 to line 7 calculates the average value of the array. The processing of these commands can be represented as in Figure 7-1. Then, line 9 to line 11 writes the results. These lines call a subroutine called wrte_binmat, which performs precisely the opposite operation to read_binmat. This subroutine has the function of opening the binary file with the name cofname, with the device number 16, and writing the nx×ny array ranudat. It is described in more detail in the next section.

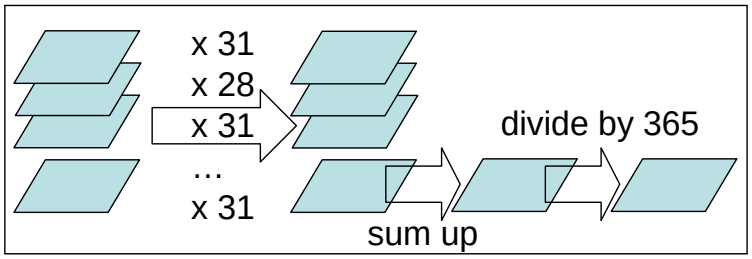


Figure 7-1 Calculation in mon2yearcmap

7.2 Fortran Subroutines

The command mon2yearcmap.f uses the two subroutines read_binmat and wrte_binmat. This section describes subroutines.

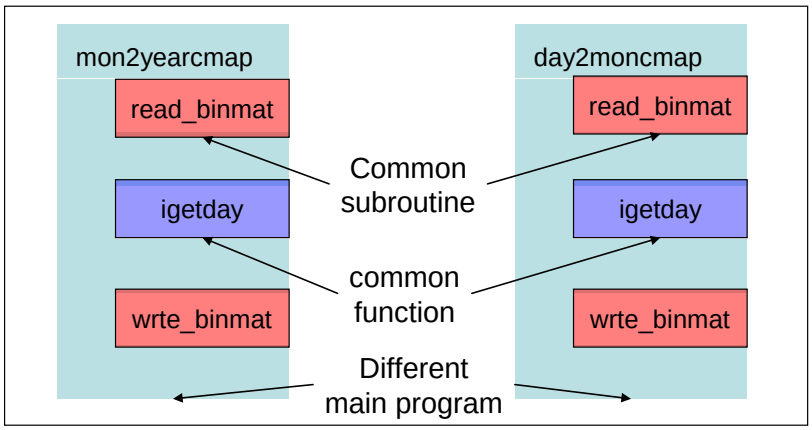


Figure 7-2 Subroutines, functions and programs

The command mon2yearcmap reads monthly average data and writes yearly average data. It would be good to create a command called day2moncmap that reads daily average data and writes monthly average data in the same manner. These two commands will have differences in timings of reading and writing of files but will be similar in terms of operations for reading and writing files and calculating the number of days in a month. As shown in Figure 7-2, the same subroutines and functions can be used with the different main programs mon2yearcmap and day2moncmap. Thus, subroutines are useful when you want to do the same processing in a number of programs.

When a subroutine is called by a program, the following format is used.

```
call subroutine_name (argument1, argument2, argument3, ...)
```

The arguments are handed over to the subroutine, and are returned to the program when the processing in the subroutine ends.

Have a look at the source code for read_binmat.f.

```
1: subroutine read_binmat(rdat,nx,ny,infile,cfname)
2: cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
3:      open(infile,file=cfname,access='DIRECT',status='old',recl=4*nx)
4:      do j = 1,ny
5:        read (infile,rec=j)(rdat(i,j),i=1,nx)
6:      end do
7:      close (infile)
8: c
9:      end
```

Line 1 declares that this is a subroutine. The terms in the parentheses are the arguments that are passed from a program. The arguments are handed over in the order they are written. It does not matter if the names of the variables are different between

the program and the subroutine, but the variable types and array sizes must strictly match. For example, in mon2yearcmap, the values in the variable rmondat in the program are handed over to the variable rdat in the subroutine. Lines 3 to 7 are exactly the same as the structure you studied in Chapter 5, so you should be able to understand them. When processing like the function here to “open a file and assign the contents to an array” is separated from the main program, it is called a subroutine.

7.3 make and makefiles

When a program and subroutines and functions are provided in separate files, as with mon2yearcmap, it is necessary to integrate them into a single program. The program that administers this operation is the make command. A file that provides instructions to the make command is a makefile.

When mon2yearcmap is compiled, the source code of the program mon2yearcmap.f, the source code of the subroutines read_binmat.f and wrte_binmat.f, and the source code of the functions igetday.f and len_trim.f are compiled by a Fortran compiler and translated into machine language, as object files. These object files have the names mon2yearcmap.o, read_binmat.o, wrte_binmat.o, igetday.o, and len_trim.o, respectively. Then these five object files are combined to form the single command mon2yearcmap.

The makefile for mon2yearcmap is at ~/unix_semi/src/Makefile. As well as compiling mon2yearcmap, this makefile describes various settings. To keep the following description simple, only matters relating to mon2yearcmap are described. Details of the grammar of makefiles cannot be described in this study text; if you wish to know more, look at the references in Appendix D.

```
 1:#############################################################
 2:#    Macro
 3:#############################################################
 4:RM     = /bin/rm        # path to command rm
 5:FC     = gfortran       # path to fortran compiler
 6:FFLAGS =                # option for fortran compiler
 7:#############################################################
 8:#    Suffix rule
 9:#############################################################
10:.f.o:
11:	$(FC) $(FFLAGS) -c $<
12:#############################################################
13:#    Dependency
14:#############################################################
15:TARGET1     = mon2yearcmap
16:COMPONENT1 = read_binmat.o wrte_binmat.o ¥
17:               igetday.o len_trim.o
18:#############################################################
19:#    Compilation
20:#############################################################
21:$(TARGET1) : $(TARGET1).o $(COMPONENT1)
22:	$(FC) -o $@ $@.o $(COMPONENT1)
```

A makefile is a script written in accordance with the grammar of make. This grammar differs from Bourne Shell scripts in a number of points, but the structures are similar.

Lines 1 to 3 are comments. In makefiles too, # introduces a comment. Lines 4 to 6 assign values to variables. In a Bourne Shell script, spaces must not be added before or

after an equals sign, but in a makefile spaces can be added. When a line continues onto the next line, as in line 16, a ¥ (backslash) is added to the end of the line and immediately followed by a line feed.

Lines 10 and 11 indicate how the object files with the extension .o should be created. The grammar will not be described here, but this case means that the object files are created by compiling Fortran source codes with the extension .f using the Fortran compiler assigned to the variable FC.

Lines 15 to 17 define variables relating to dependencies between the files, and assign values to the variables. The file being created here is the executable file mon2yearcmap. When creating mon2yearcmap, in addition to the object file of the program itself (mon2yearcmap.o), the four object files read_binmat.o, wrte_binmat.o, igetday.o, and len_trim.o are required. This can be described as mon2yearcmap being dependent on the other four object files. In this example, the variable to which the command to be created is assigned is TARGET1 and the variable to which the dependent object files are assigned is COMPONENT1.

Lines 21 and 22 instruct compilation. Line 21 shows the dependencies of the files and assigns values to variables, as follows.

mon2yearcmap : mon2yearcmap.o read_binmat.o wrte_binmat.o igetday.o len_trim.o

The symbol : (colon) means that the file on the left side is dependent on the files on the right side. That is, if a file on the right side is newer than the file on the left side, the new file should be recompiled. Line 22 indicates the manner of compilation and assigns values to the variables, as follows.

gfortran –o mon2yearcmap mon2yearcmap.o read_binmat.o wrte_binmat.o igetday.o len_trim.o

As you saw in Chapter 5, the first part, gfortran -o mon2yearcmap, means that the executable file is given the name mon2yearcmap instead of the name a.out. The object file names follow as arguments. These are joined into a single executable file by the Fortran compiler.[34]

In the code, the parts enclosed by dashed lines are tabs. Note that ordinary spaces would produce errors. When a make command is executed, it searches for a makefile in the same directory and ignores makefiles in other directories.

[34] More specifically, the Fortran compiler automatically uses a special UNIX command called a linker to integrate the object files.

Exercise

- Make a program that inputs a year and a month and outputs the number of days in that month. Give the file the name “program”. Compile and execute it, It should work as shown below.

```
% program 1995 3
31
% program 1995 2
28
% program 1996 2
29
% program 2000 2
29
% program 1900 2
28
```

(A solution to the exercise is in ~/unix_semi/lesson7.)

Chapter 8

Bourne Shell Scripts (2): Do Loops, For Loops, and If Statements

In Exercise 2 of Chapter 6, you had to type 42 or more commands into the terminal. Repeatedly inputting similar commands takes time and makes mistakes more likely, so should be avoided. This chapter describes some ways to cut down on this work.

8.1 Control Statements and Conditional Expressions in Shell Scripts

As described in Chapter 1 and Chapter 3, in UNIX, when commands are saved in a text file and permission to execute is granted, processing can be executed. A file that is written in accordance with Bourne Shell grammar in such a case is called a Bourne Shell script. Bourne Shell grammar includes control statements such as loops and if statements. These control statements can be used for more sophisticated processing. The main control statements used in Bourne Shell scripts are summarized in Table 8-1, and conditional expressions are summarized in Table 8-2.

Table 8-1 The main control statements used in Bourne Shell scripts

for loop	**for VAR in VALUES; do** **process** **done**
while loop	**while [condition]; do** **process** **done**

if statement	**if [condition]; then**
	  process1
	elif [condition]; then
	  process2
	else
	  process3
	fi

VAR: a variable name VALUES: a list of values [**condition**]: a conditional expression process: command(s)	

Table 8-2 Conditional expressions used in Bourne Shell scripts

Testing whether there is a file called FILENAME	**[-f FILENAME]** [35]
Testing whether there is a directory called DIRNAME	**[-d DIRNAME]**
A=B (when testing a character string)	**[A = B]**
A≠B (when testing a character string)	**[A != B]**
A=B (when testing an integer)	**[A –eq B]**
A≠B (when testing an integer)	**[A –ne B]**
A > B	**[A –gt B]**
A ≥ B	**[A –ge B]**
A < B	**[A –lt B]**
A ≤ B	**[A –le B]**
And	**-a**
Or	**-o**

8.2 Example 1: A shell script using a for loop

This is a Bourne Shell script that displays integers from 1 to 12 in the terminal, representing January to December, and outputs “Happy birth month!” with a birth month (which is set to 8 for August here).

```
 1: #!/bin/sh
 2: ##################################################
 3: MONS=”01 02 03 04 05 06 07 08 09 10 11 12”
 4: BIRTHMON=”08”
 5: ##################################################
 6: for MON in $MONS; do
 7:   if [ $MON = $BIRTHMON ]; then
 8:     echo $MON “Happy birth month!”
 9:   else
10:     echo $MON
11:   fi
12: done
```

In line 1, #!/bin/sh declares that this is a Bourne Shell script. Lines 3 and 4 assign values to the variables MONS and BIRTHMON, respectively. Line 6 is a control statement for repetition—a for loop. Here, for MON in $MONS; do means sequentially assigning the values in the list MONS to the variable MON and repeating the commands between do and done. In this case, the 12 values in MONS are assigned one after another to MON: MON=01, MON=02, ... MON=12. Line 7 performs the processing of a conditional test—an if statement. Here, if [$MON = $BIRTHMON];

[35] There must be spaces before and after the brackets [and].

then represents the condition “if the variable MON matches the variable BIRTHMON”. If this condition is satisfied, line 8 is executed, in which the command echo outputs character strings to the terminal. If the condition is not satisfied then, after the else on line 9, line 10 is executed.

8.3 Example 2: A shell script using a while loop

This is a Bourne Shell script that displays dates from January 1 (in the form 0101) to December 31 (1231) in the terminal, and outputs “Happy birthday!” with a birth date (which is set to February 28).

```
 1: #!/bin/sh
 2: ##################################################
 3: YEAR=2005
 4: MONS=“01 02 03 04 05 06 07 08 09 10 11 12”
 5: BIRTHMON=“02”
 6: BIRTHDAY=“28”
 7: ##################################################
 8: for MON in $MONS; do
 9:   DAY=1                      # refresh day
10:   DAYMAX=`nofday $YEAR $MON` # set the last day of the month
11:   while [$DAY –le $DAYMAX ]; do
12:     if [ $MON = $BIRTHMON –a $DAY = $BIRTHDAY ]; then
13:       echo $MON $DAY “Happy birth day!”
14:     else
15:       echo $MON $DAY
16:     fi
17:     DAY=`expr $DAY + 1`     # day for next iteration
18:   done
19: done
```

This Bourne Shell script builds on that in Example 1. Lines 9 to 18 are greatly changed. The for loop in line 8 assigns the values from 1 to 12 to the variable MON. In line 9, the value 1 is assigned to the variable DAY. Then, the command nofday is used to check the number of days in that month of that year (see Table 4-2). You studied this in the exercises of Chapter 7. As an example, the result of nofday 2005 01 is 31. The backquote marks (`) here represent “the result of executing the command enclosed between the backquote marks”. That is, for YEAR=2005 and MON=01, the content of the backquote marks is calculated as 31, and this result is assigned to the variable, as though the line was DAYMAX=31. The while loop in line 11 means that processing is repeated. Here, while [$DAY -le $DAYMAX]; do means to repeat the commands between the do and the done as long as the value of DAY is lower than the value of DAYMAX. The done that goes with this do is in line 18. You can see that line 17 says DAY=`expr $DAY + 1`. The command expr performs basic arithmetic. In this case, it adds 1 to the variable DAY and assigns the result to DAY. Note that there must be spaces between $DAY, +, and 1.

8.4 Example 3: Performing Exercise 2 of Chapter 6 with a single Bourne Shell script

As mentioned above, you had to type 42 or more commands into the terminal for Exercise 2 of Chapter 6. Now you will concentrate this processing into a single shell script. First, put the commands that you typed into the terminal for Exercise 2 of Chapter 6 into a text file. The reason for this is that you can make this file into a shell script just by granting permission to execute. Now, as an example, the following shell script will find populations living in areas with annual rainfalls of at least 0 mm/year but less than 50 mm/year, and of at least 50 mm/year but less than 100 mm/year.

```
1: # 0mm/year =< Precipitation < 50mm/year
2: maskcmap pop.cmap            rain.cmap ge   0 temp.pop.ge0.cmap
3: maskcmap temp.pop.ge0.cmap   rain.cmap lt  50 temp.pop.ge0.lt50.cmap
4: sumcmap  temp.pop.ge0.lt50.cmap  0
5: # 50mm/year =< Precipitation < 100mm/year
6: maskcmap pop.cmap            rain.cmap ge  50 temp.pop.ge50.cmap
7: maskcmap temp.pop.ge50.cmap  rain.cmap lt 100 temp.pop.ge50.lt100.cmap
8: sumcmap  temp.pop.ge50.lt100.cmap 0
```

Look for processes in the shell script that are the same and replace their contents with variables. For example, lines 2 to 4 and lines 6 to 8 do the same operations but differ in the upper limit (50 or 100) and the lower limit (0 or 50). Therefore, you can rewrite the shell script to replace the upper limits with a variable UL and the lower limits with a variable LL. Moreover, two sets of processing were needed in Exercise 2, for populations and for areas. Rewrite the script to replace pop with a variable DAT (for data). The rewritten script should be something like the following. You can see that lines 3 to 5 and lines 8 to 10 are identical.

```
 1: # 0mm/year =< Precipitation < 50mm/year
 2: DAT=pop; LL=0; UL=50
 3: maskcmap ${DAT}.cmap       rain.cmap ge ${LL} temp.${DAT}.ge${LL}.cmap
 4: maskcmap temp.${DAT}.ge${LL}.cmap rain.cmap lt ${UL} temp.${DAT}.ge${LL}.lt${UL}.cmap
 5: sumcmap temp.${DAT}.ge${LL}.lt${UL}.cmap 0
 6: # 50mm/year =< Precipitation < 100mm/year
 7: DAT=pop; LL=50; UL=100
 8: maskcmap ${DAT}.cmap       rain.cmap ge ${LL} temp.${DAT}.ge${LL}.cmap
 9: maskcmap temp.${DAT}.ge${LL}.cmap rain.cmap lt ${UL} temp.${DAT}.ge${LL}.lt${UL}.cmap
10: sumcmap temp.${DAT}.ge${LL}.lt${UL}.cmap 0
```

Now put together a do loop, a for loop, and an if statement to complete the shell script. First define a variable JOB, set LL=0 and UL=50 for JOB=1, and set LL=50 and UL=100 for JOB=2. Then use a while loop to make the processing repeat with the variable JOB changing from 1 to 2. Similarly, define the variable DAT and use a for loop to repeat the processing for pop and lndara. You will get something like the following.

```
 1: #!/bin/sh
 2: ##################################################
 3: for DAT in pop lndara; do
 4:  JOB=1
 5:  while [ $JOB –le 2 ]; do
 6:   if  [ $JOB -eq 1 ]; then
 7:    LL=0; UL=50
 8:   elif [ $JOB -eq 2 ]; then
 9:    LL=50; UL=100
10:   fi
11:   maskcmap ${DAT}.cmap        rain.cmap ge ${LL} temp.${DAT}.ge${LL}.cmap
12:   maskcmap temp.${DAT}.ge${LL}.cmap rain.cmap lt ${UL} temp.${DAT}.ge${LL}.lt${UL}.cmap
13:   sumcmap temp.${DAT}.ge${LL}.lt${UL}.cmap 0
14:   JOB=`expr $JOB + 1`
15:  done
16: done
```

By using variables and control statements to tighten it up, you have written a much more streamlined script.

Exercise

1. The original data (ASCII files) that you can download from the CMAP website is in ~/unix_semi/dat_org. Using the techniques that you used in the exercises of Chapter 2, write a shell script to convert these ASCII files to monthly binary files, each of 144×72×4 bytes. Give these binary files the same names as the files in ~/unix_semi/dat_org.

Hint

To adjust the beginnings of each number, use expressions below.

```
% YR=2
% YR=`echo $YR|awk'{prontf("%2.2d", $1)'`
```

When you use last two digits to express a certain year, carefully examine, what should be done between 1999 and 2000. One option is to use different loops for before and after 2000. Another option is to add a if-statement, withdraw 100 when year exceeds 100 (i.e. three digits).

(A solution to the exercise is in ~/unix_semi/lesson8.)

Appendix A
Installing and Using Xming

You need software called a PC X server to log in to a UNIX computer from a Windows computer. The software mentioned in this study text, which is provided for free, is Xming. You should learn how to install this program and use it at a basic level.

A.1 Installing Xming

1. Go to the Xming website.[36]
2. Click on the link labeled “distributed”.

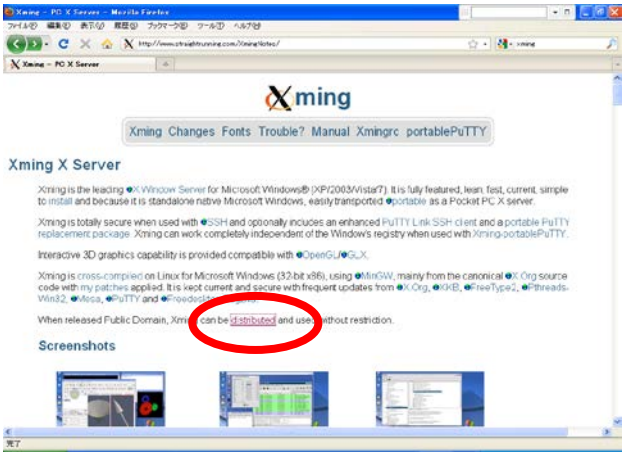


Figure A-1 Screenshot of procedure step 2

3. Click on the link labeled “SourceForge Project Xming”.

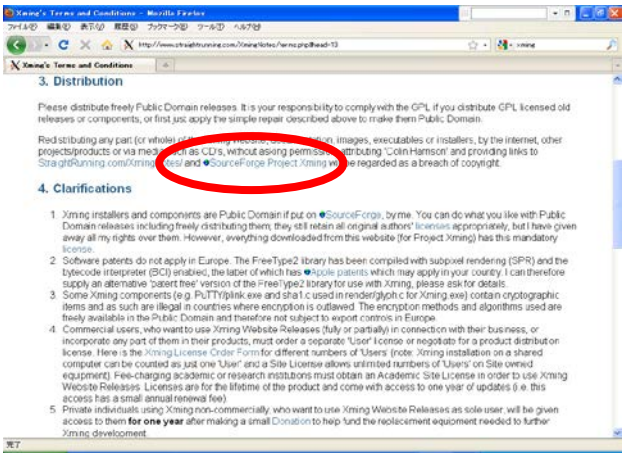


Figure A-2 Screenshot of procedure step 3

[36] http://www.straightrunning.com/XmingNotes/

4. Click the button labeled “View all files”.

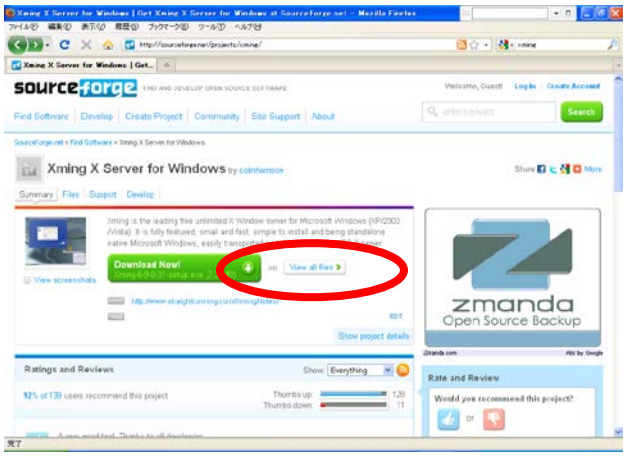


Figure A-3 Screenshot of procedure step 4

5. Click on “Xming-fonts”, and click on the “7.5.0.11” (or similar) that appears. Then click on “Xming”, and click on the “6.9.0.31” (or similar) that appears. Confirm that “Xming-fonts-7-5-0-11-setup.exe” and “Xming-6-9-0-31-setup.exe” appear in your browser

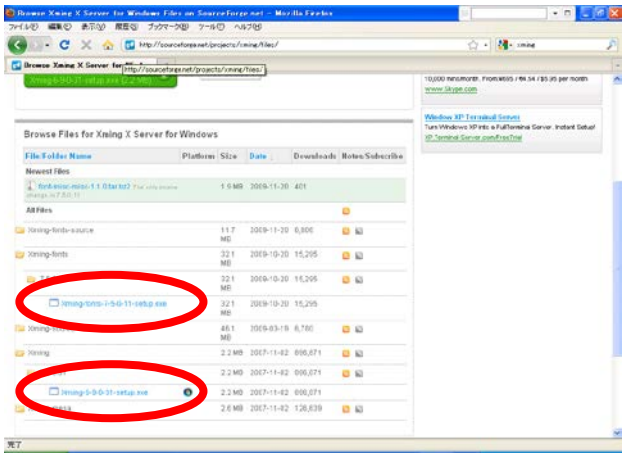


Figure A-4 Screenshot of procedure step 5

6. Confirm that Xming-fonts-7-5-0-11-setup.exe and Xming-6-9-0-31-setup.exe have been downloaded.
7. Execute Xming-fonts-7-5-0-11-setup.exe.[37] Basically just keep clicking “Next” until the installation is complete.

[37] N.B.: Installing Xming-fonts-7-5-0-11-setup.exe first is a smoother process. If you install Xming-6-9-0-31-setup.exe first, you must then shut down Xming before installing Xming-fonts-7-5-0-11-setup.exe.

Figure A-5 Screenshot of procedure step 7

8. Execute Xming-6-9-0-31-setup.exe. Basically just keep clicking “Next” until the installation is complete.

Figure A-6 Screenshot of procedure step 8

A.2 Using Xming

1. Start up Xming: Find the Xlaunch icon in the Start Menu and click on it.

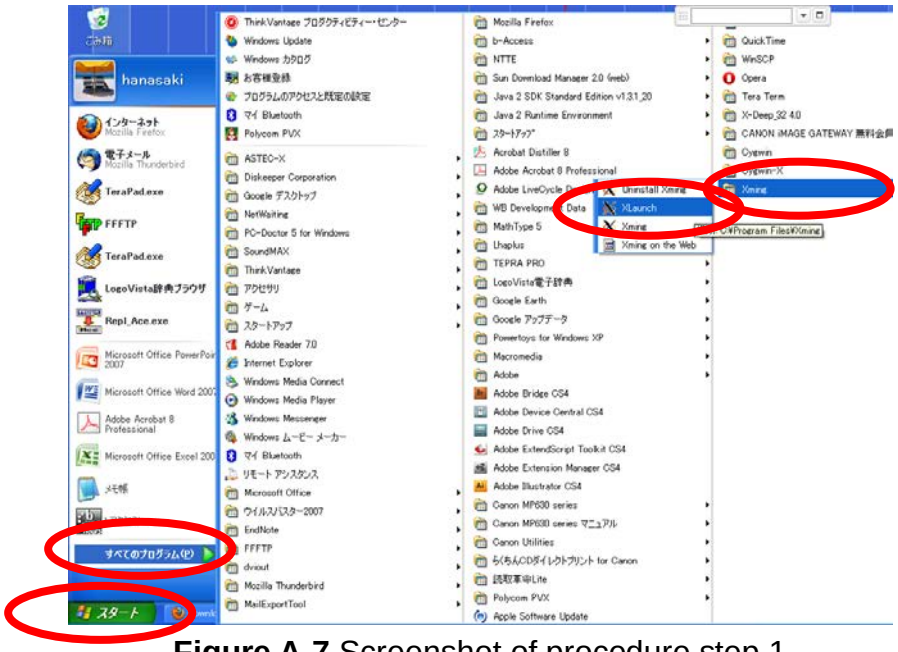


Figure A-7 Screenshot of procedure step 1

2. Select “Multiple windows”.

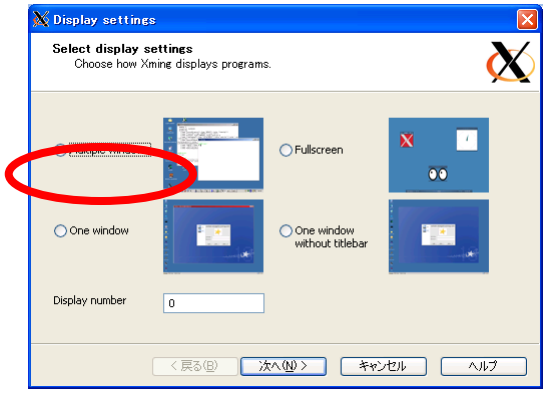


Figure A-8 Screenshot of procedure step 2

3. Select “Start a program”.

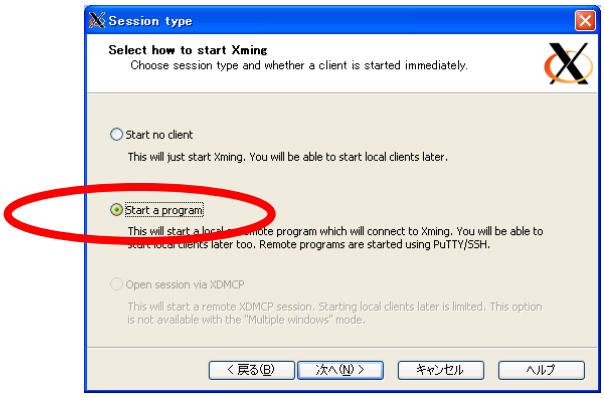


Figure A-9 Screenshot of procedure step 3

4. Select “Using PuTTY” and type the name of the UNIX computer you want to connect to in the “Connect to Computer” box. Leave the other two boxes empty.

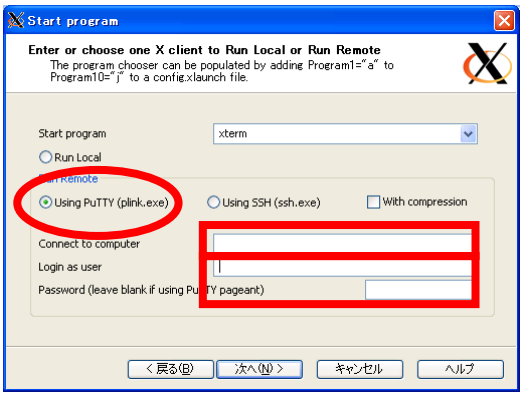


Figure A-10 Screenshot of procedure step 4

5. When the next window appears, just click on “Next” without making any selections.

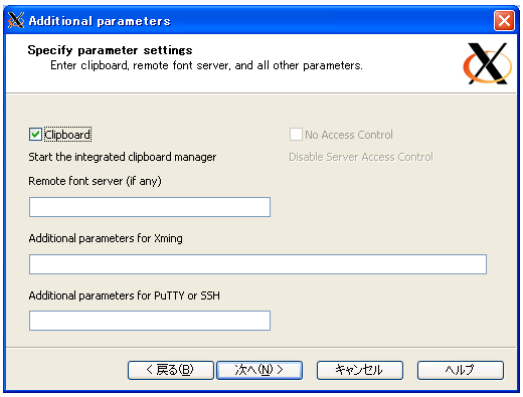


Figure A-11 Screenshot of procedure step 5

6. To save the current selections, click on “Save configuration”; you will be able to skip steps 1 to 6 the next time. Finally, click on the “Finish” button.

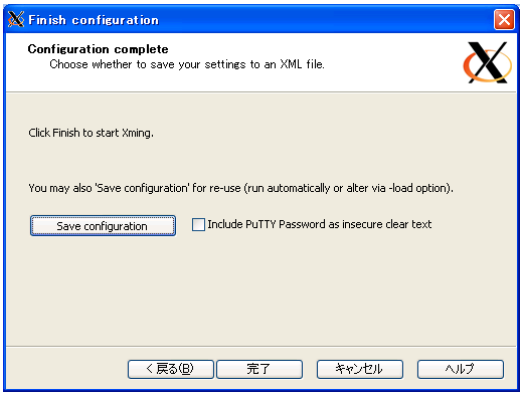


Figure A-12 Screenshot of procedure step 6

7. When you make the connection for the first time, the following window appears. Select “Yes”.

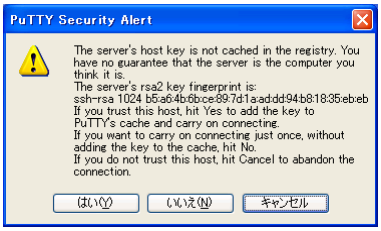


Figure A-13 Screenshot of procedure step 7

8. Input your ID (login name).

Figure A-14 Screenshot of procedure step 8

9. Input your password.

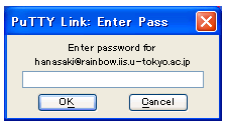


Figure A-15 Screenshot of procedure step 9

10. When the following screen appears, you have successfully connected.

Figure A-16 Screenshot of procedure step 10

A.3 Troubleshooting

If Xming has a problem, shut it down and restart it from Xlaunch. To shut down Xming, find the X icon in the taskbar, right-click on it, and select “Exit”.

Figure A-17 To exit Xming

If there is a problem with login, looking at the log (history) may help. Right-click on the X icon in the taskbar and select “View Log”.

Figure A-18 To view log of Xming

Appendix B
Configuring a UNIX Computer (Settings that do not require administrator rights)

This appendix describes configuration of the UNIX computer that you are using as a server. You do not need administrative rights to do the configuration in this appendix, but you do need a basic understanding of UNIX operations. If you have never studied UNIX before using this study text, ask an administrator to do the configuration for you.

B.1 Obtaining the Resources for this Study Text

Go to the home directory of the user who is using this study text, and download the resources for the study text.See Table 1-1 for the basic usage of the UNIX commands. First, launch the web browser.

```
% cd ~
% firefox
```

Access to http://sites/google.com/h08model/home
Go to Manual page
Download material

Decompress the gzip compressed file and the tar archive.

```
% gunzip unix_semi.tar.gz
% tar xf unix_semi.tar
```

A directory called something like unix_semi_20120601 should have appeared. This directory name is rather long, so do the following to change it to unix_semi.

```
% mv unix_semi_20091209 unix_semi
```

Next, configure the environmental variables by editing .bashrc in the home directory.

```
% emacs ~/.bashrc
```

First set the path to ~/unix_semi/src. Thus, the sample commands in src can be used from any directory. Then change the environmental variable to set binary files that are inputted and outputted by gfortran to big-endian[38] (but set device numbers 25 and 26 to little-endian).

[38] This is because the input and output binary files of H08 are all coded in big-endian. However, there would be no problem going through this study text with inputs and outputs in little-endian. To do this, do not set this environmental variable but see section B.3.

```
1:##############################################################
2:# PATH                                                  ←comments : title
3:##############################################################
4:PATH=.:${PATH}:~/unix_semi/src                     ←setting of the pass
5:##############################################################
6:# FORTRAN                                                ←comments : title
7:##############################################################
8:export GFORTRAN_CONVERT_UNIT="big_endian;little_endian:25-26"   ←setting
```

Then compile the sample commands. The directory src includes the source code of commands used in this textbook. These commands become executable after you compile programs from the code.

```
% cd ~/unix_semi/src
% make veryclean
% make all
```

This completes preparation.

B.2 Using the Intel Fortran Compiler

If you want to use the Intel Fortran Compiler instead of gfortran, do the following.

1. Because inputs and outputs are coded in big-endian, add the following line to ~/.bashrc. This also sets device numbers 25 and 26 to little-endian.

   export F_UFMTENDIAN="big;little:25,26"

2. Next, to change (to byte) the unit of recl values, which is set for binary outputs using write statements, add the following line to ~/.bashrc.[39]

   alias ifort='ifort –assume byterecl'

3. Amend ~/unix_semi/src /Makefile as follows.

   Line 10: FC = ifort -assume byterecl

   Line 12: FL = ifort -assume byterecl

4. Recompile the code in ~/unix_semi/src.

```
% cd ~/unix_semi/src
% make veryclean
% make all
```

B.3 Input/Output in Little-endian

If you want binaries to be coded in little-endian rather than big-endian, do the following. In particular, as of December 2009, if gfortran has been installed by using apt-get on a Core 2 Duo machine running Ubuntu (one kind of Linux operating system), there is a problem with gfortran commands not being properly executed if the

[39] In gfortran, the unit of recl values in write statements is bytes, whereas the default in the Intel Fortran Compiler is records. If the environmental variable was not changed, real values would be stored in 16 bytes and there would be mistakes in calculated values.

environmental variable is configured to GFORTRAN_CONVERT_UNIT=“big_endian”. In this case, all binary files encoded as big-endian in the resources must be changed to little-endian.

First, delete the setting of the environmental variable GFORTRAN_CONVERT_UNIT in ~/.bashrc (add # to the beginning of the line. If you wish to undo, just remove "#".). Then recompile all the programs in ~/unix_semi/src.

```
% cd ~/unix_semi/src
% make veryclean
% make all
```

Next, do the following to convert the files in ~/unix_semi/dat_bin.

```
% cd ~/unix_semi/lesson8
% sh exercise.sh
```

Finally, do the following to convert the resources in ~/unix_semi/lesson6.

```
% cd ~/unix_semi/lesson6
% sh asc2bin.sh
```

Appendix C
Configuring a UNIX Computer (Settings that require administrator rights)

This appendix describes configuration of the UNIX computer that you are using as a server. You need administrative rights to do the configuration in this appendix. Ask an administrator to do the configuration for you.

C.1 Minimum UNIX Requirements for Working with this Study Text

The following environment is required for working through this study text. Computer administrators, please provide this environment to users using the study text.

Table C-1

OS	UNIX or a UNIX-based OS
Login shell	Any, but the descriptions in this study text assume bash.
Communications	Either remote login by ssh or X11 Forwarding must be possible.
Software	Access to the following software: make gcc (GNU C compiler) GMT (Generic Mapping Tool) ImageMagick gfortran (GNU Fortran compiler)

C.2 An Example of Setting Up a UNIX Environment (Mac OS X 10.6)

For reference, this is how I set up the above-described environment on a Mac Mini server (with Mac OS 10.6 Snow Leopard Server) in November 2009.

1. To set up ssh
   - Edit the file /etc/sshd_config and set “X11Forwarding yes”. Without this setting, images and the like cannot be transferred to a Windows computer.
   - Edit the same file and set “PasswordAuthentication yes”. Without this setting, there will be password authentication errors when logging in with Xming or other PC X server software.

2. To create an account (in Mac OS X)
   - Launch the Workgroup Manager application. The root password is required.
   - Click on the key icon at the top right, enter the password again, and you can add a user. (Your current username is “root”.)
   - Click on “New User” in the toolbar, and you can add a user.
   - You cannot create a home directory from the Basic tab, so configure the home directory in the Home tab.[40]
   - The other tabs can be left empty. Finish by clicking on the “Create Home Now” icon to save.
3. Other settings (particular issues in Mac OS X)
   - With the default settings, if you try to log in from Xming or other PC X server software, “bash: xterm: command not found” is displayed and you cannot connect. Editing the file /etc/paths and adding “/usr/X11/bin” to the path should work, but it does not work well, so create the file .bashrc in each home directory and add the line “PATH=$PATH:/usr/X11/bin”.
   - With the default settings, the keyboard operation Ctrl-Space is assigned to launching Spotlight, but this will cause problems when using Emacs. Select the “Keyboard/Mouse” icon in System Preferences and change the shortcut assigned to Spotlight in the Keyboard Shortcuts tab.
   - If you cannot type backslash (¥), use Option-¥ (Alt-¥).
   - To stop Spotlight, add “SPOTLIGHT=-NO-” to /etc/hostconfig.
4. To install make and gcc (in Mac OS X)
   - In the default settings, make and gcc are not present. First download Xcode from the Apple website (you need ADC membership to do the download).[41] After downloading, double-click on Xcode.mpkg to install. Thus, make and gcc are installed.
5. To install Fink (with the Mac OS X version of apt-get)
   - Go to the Fink website.[42] As of November, 2009, the binary cannot be installed on Mac OS 10.6 (Snow Leopard), so install from the source. Many settings are available when installing, but they can all generally be left at the defaults.
   - Install FinkCommander (0.5.5). This does not work with the default settings. In Preferences > Paths > Perl, change the path from /usr/bin/perl to /usr/bin/perl5.8.9 and it will work.
6. To install GMT and ImageMagick
   - Use Fink to do the installations. However, ImageMagick may be installed more quickly by other methods.[43]

---

[40] It does not work if attempted in the same way as /home/guest.
[41] http://developer.apple.com/tools/xcode/
[42] http://www.finkproject.org/
[43] The ImageMagick website (http://www.imagemagick.org) recommends installing from MacPorts, which is very similar to Fink, when installing on Mac OS X. To do this, first install a package from the

7. To install gfortran

From the gfortran website,[44] download the binary for Mac OS 10.5 (Leopard). It works well with Mac OS 10.6 (Snow Leopard) too.

MacPorts website (http://www.macports.org). Then set the environmental variables as instructed.

[44] http://gcc.gnu.org/fortran/

Acknowledgements

Acknowledgements in the first edition (English PowerPoint material)

This study text was created as a supplementary resource for the course in UNIX for Beginners that was given at the Oki Laboratory of the University of Tokyo Institute of Industrial Sciences in the summer semester from 2002 to 2005. It presents the minimum knowledge required for students with no prior knowledge of programming or UNIX at all to start research at the Oki Laboratory.

I would like to thank the students who attended my unpolished UNIX course: C. Apirumanekul, Y. Arai, A. Aslam, J. Cho, N.S. Farzin, D. Ikari, T. Inuzuka, Y. Ishizaki, Y. Koiwa, T. Kokubo, K. Kubo, M. Lin, C. Manusthiparom, T. Okazawa, W. Saita, T. Sakimura, Y. Suga, K. Takagaki, Q. Tang, and N. Utsumi, T. Yamada.

The Oki Laboratory conducts research into water circulation systems at the global scale. This requires manipulation of large volumes of time series gridded data, so the content of the study text concentrates on this issue. I have devised it such that, by suitable modification of the content, it can be of immediate use in research.

I created this study text in crude, functional English, to moderate the language barrier for the hard-working students who come to the laboratory from abroad each year. The English in the study text probably has many mistakes in grammar and usage, and the content is probably more difficult to understand because of my poor English. Thank you for your tolerance.

July 5, 2005
Naota Hanasaki

Acknowledgements in the second edition (Japanese Word document)

Previously, I created the supplementary materials for the UNIX course in English. I did this with the aim of reducing the language barrier to the many students from abroad at the Oki Laboratory, but there were many complaints from the Japanese students, who found the computer terminology hard to understand in English. Moreover, the supplementary materials were created in Microsoft PowerPoint for use as visual aids when I was giving my lectures. As a result, there were complaints that there was not enough explanatory text, making it hard to use for revision or self-study. Therefore, in June and July 2007, with the help of the excellent translator and editor Tomoko Nitta, I reworked the English PowerPoint slides into a study text in Japanese. I was unable to give this as much time as I would have liked, so the explanations may be inadequate and

there may be gaps in the logic. Having read this study text, please send me your comments, good or bad, and I will endeavor to improve it.

July 31, 2007
Naota Hanasaki