Mem. S.A.It. Suppl. Vol. 8, 78

Memorie della
Supplementi

Introduction to LINUX OS for new LINUX users

– Basic Information Before Using The Kurucz Codes Under LINUX–

M. Taşkın Çay

İstanbul Üniversitesi, Fen Fakültesi, Astronomi ve Uzay Bilimleri Bölümü, 34119, Üniversite, İstanbul / Turkey.
e-mail: taskin@istanbul.edu.tr

Abstract. Recently the ATLAS suite (Kurucz) was ported to LINUX OS (Sbordone et al.). Those users of the suite unfamiliar with LINUX need to know some basic information to use these versions. This paper is a quick overview and introduction to LINUX OS. The reader is highly encouraged to own a book on LINUX OS for comprehensive use.
Although the subjects and examples in this paper are for general use, they to help with the installation and running the ATLAS suite.

Key words. Operating Systems: LINUX – Model Atmospheres: ATLAS9, ATLAS12

1. Introduction

1.1. Why should I use LINUX OS?

LINUX is now a widely used operating system. For a general user there can be many reasons to prefer LINUX OS for everyday use;

1- it is free,

2- it comes with many programs including an office suite,

3- security in LINUX is high,

4 - it is an open source operating system - you can change the system as your wish; even create your own private LINUX distribution.

For a scientific user we may add three more important reasons;

1- it is a very stable system - LINUX can easily handle multiple tasks and runs for a long time without crashing,

2- it allows parallel programming,

3- it turns an ordinary PC to an UNIX workstation.

1.2. Which LINUX distribution should I use?

Today, hundreds of LINUX distributions can be found: RedHat, Slackware, SUSE, Mandrake, ...etc. Although the names are different, all LINUX distributions still have the same kernel. Thus in theory any program should run under any LINUX distribution. Nevertheless, using the pre-compiled binary executables to run a program generally requires the same LINUX distribution under which they have been compiled. The best way to be sure that a program will run in your PC without trouble, is having the source code of the program and compiling it yourself in your PC.

The question can simply be answered as *you can use any LINUX distribution you enjoy*. It is only a matter of comfort.

2. Basics of LINUX

2.1. Users, user rights and file permissions

LINUX is a multi-user and multi-tasking operating system. Thus the system can be used by more than one user in the same time and can perform multiple operations simultaneously.

The users of the system have limited rights to use the system and these rights are given by the system administrator who has access to the *root* account [1].

Each user has a private directory called *home*, in the system. This is the main working directory of the user. The users cannot reach and modify the other directories and files except for the shared ones.

On the other hand the root has unlimited rights; it can reach and modify all the directories and files. Although being unrestricted may seem funny, using the system as root can be dangerous; because you may delete an important file or make hazardous changes in the system accidentally. To avoid such troubles it is better to login to the system as an ordinary user for everyday use. In general we may say that:

- *Login to the system as user* for everyday use or private installation of a program etc.

- *Login to the system as root*, if you are going to make changes to the system (like upgrading the system itself) or system wide installation of a program etc.

2.1.1. Creating a user account

The easiest way to create a user account is to use the graphical user interface (GUI). Depending on the LINUX distribution and/or the desktop environment (GNOME or KDE) you are using, it may be found in different places, but in any case it should be reachable from the start menu. Fig. 1 shows adding a user account into the system in the SUSE Linux case.

If there is no reason to do so, the default home directory should not be changed.

[1] Management of the user accounts and file permissions can only be done by the root.

The most used *shell* and also the default is the *bash* shell, so it should also be left unchanged. A shell is a program that interprets the commands you have given to the system. For example, when you write *pico* in the *terminal screen* [2] (means in the *command-line*) then enter, the shell understands that you want to run the program pico that is a text editor, and starts the program if installed in your system. There are also several other shells such as *csh* (C shell), *sh* or *tsch* etc. Generally, using any shell is a matter of comfort and there are little differences between shell commands. Sometimes a specific program can require use of a specific shell. The user can switch between shells using *chsh* command. For example, typing

chsh csh

changes the shell to the C shell.

Several users can be grouped together to give them specific group rights (see the next section). A group can be added to the system with a in similar way to adding a user.

2.1.2. Permissions

Each file or directory in the system can be used by three types of user:

1- the owner of the file (the user),
2- the group,
3- the others.

For each of these kinds of users, there are three types of permissions: read permission (r), write permission (w) and execute permission (x). Permissions determine who can read the file (or the directory), who can write to the file or directory (this means who can modify the file or directory) or who can execute the file (it should be noted that in LINUX any file can be executable and it is not necessary to have an extension like *.exe*)

To see the file permissions in the current directory the *ls* command can be used with the *-l* flag. Please refer to the manual pages and information pages to learn about the syntaxes for

[2] The terminal screen should be reachable from quick launch bar at the bottom of the screen shown as a monitor icon.

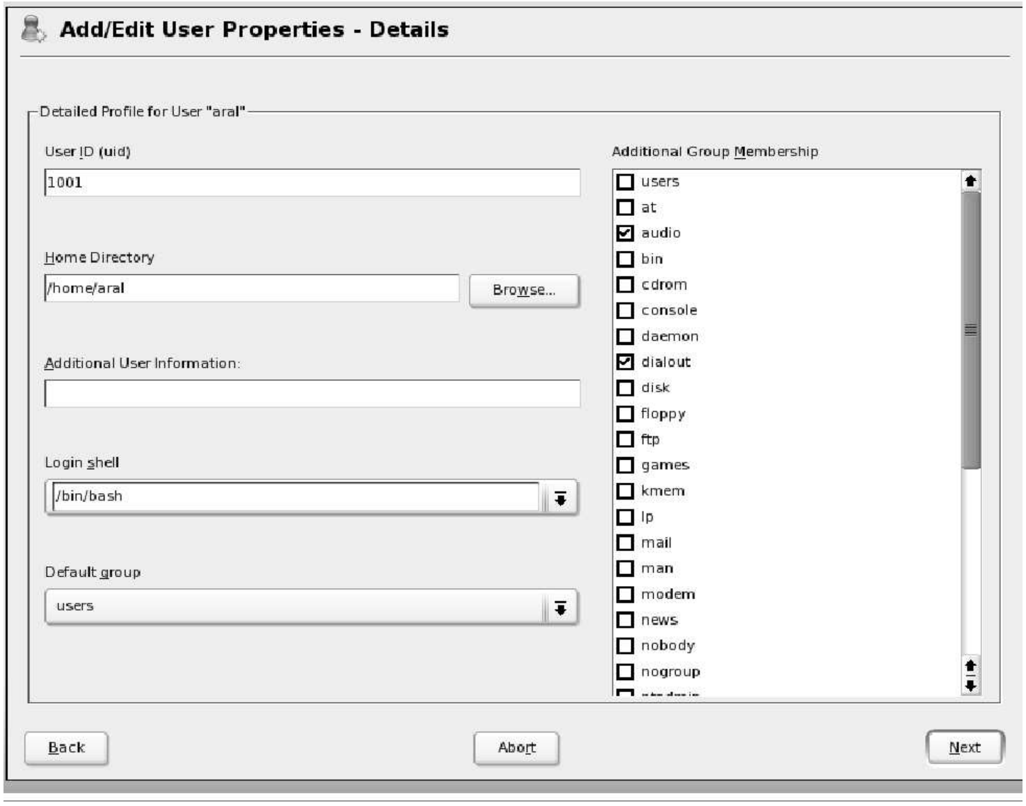

Fig. 1. A user account can be created using a GUI application.

any command and to get info about it. [3] When you type *ls -l* you will get an output something like in Fig. 2. In this example we see some files and a directory that are present in the current directory. For example, permissions for the file *readme.txt* are those:

-rw-r- -r- -

rw-: the owner can read and write but can not execute the file.

r- -: the group (means members of the group *users*) can read the file but can not write or execute.

r- -: all other users can read the file but can not write or execute.

On the other hand the file *kurucz* is a link pointing to the files at the location */usr/local/kurucz* directory. Permissions for this file are rwxrwxrwx meaning that all three types of users can read, write and execute this file. The letter *l* before the permissions indicates that this file is a link.

Ownership of a file or directory can be changed using the *chown* command:

chown -R taskin prog1.sh

would recursively change the ownership of the file *prog1.sh* (which is formerly belongs to root) to the user taskin. The same procedure is true for changing the group ownership using the *chgrp* command.

Permissions of a file can be changed using the *chmod* command. Using this command requires a little bit of mathematics:

r permission has value of 4

w permission has value of 2

[3] For example, to learn about how to use the *ls* command, in the terminal screen type: *man ls* . Also compare with *info ls*

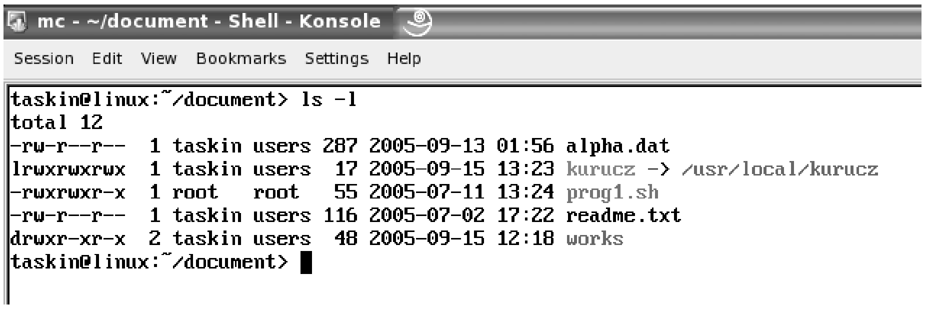

Fig. 2. Listing of a directory in the long format using the *-l* flag. The first column shows the permissions in ten digits; the first letter shows the kind of the file: *d* for a directory, *l* for a link, and – for a file. The next three letters indicates the user (owner)'s rights, following three are for the group rights and the last three for the others. *r* for the read permission, *w* for the write permission and *x* for the execute permission; blank (–) means no permission for the relevant type. The second column shows the number of links; for example, the *works* directory has two files while the others are only one file. The third and fourth columns show the owner and the group of the file. The fifth column indicates the length of the file in bytes. The last three columns are straightforward.

x permission has value of 1

Hence, for instance, if a file has permissions something like

rwxr-xr-x,

then its value is 755 (rwx: 4+2+1=7, r-x: 4+0+1=5, r-x: 4+0+1=5). So,

chmod -R 765 prog1.sh

would change the permissions of the file *prog1.sh* as rwxrw-r-x.

An alternative *numeric format* is the use of *symbolic format*. Let us assume that the file *prog1.sh* has rwxrwxrwx permissions. Then *u*,*g* and *o* show the user, the group and the others, respectively,

chmod -R g-w,o-w prog1.sh

would set the permissions as rwxr-xr-x (g-w means w permission is removed from the group. To add the permission again use g+w).

chmod -R g-x,o-w prog1.sh

sets the permissions as rwxrw-r-x. Note that since nothing is told about the user (*u*) in the examples the user will have the all rights. Again,

chmod -R +x prog1.sh

would give the x permisson to all three type of users. See the manual pages for details.

2.2. Directory structure of the system

It is useful to know about some standard directories of LINUX OS. Fig. 3 shows the standard directories in the root directory and most are system directories and cannot be reached by the users. Descriptions of some of these directories follows:

bin: The executables needed to boot the system are here. It is also home for some very commonly used LINUX commands. An executable not need to be binary, it can be a shell script. When you install a program, the executables should not be put here.

home: Includes all the individual directories of the users.

media: Removable drives (cdrom, floppy, zipdrive ...etc) are mounted here. ***mnt*** directory does the same thing. Some LINUX distributions only have a ***mnt*** directory. It is recommended to read the manual pages of the *mount* command to mount a drive to this directories.

opt: Includes the files which are not necessary to run the system. For example, the LINUX version of the ATLAS suite can only be compiled with Intel Fortran Compiler (ifc), so you would need it in your system. The ifc compiler is installed in the *opt* directory.

usr: Some other LINUX system files which are not necessary to boot the system are here. */usr/local* directory, which is our main interest in this paper, is for local program installations; Fig. 4 shows the structure of this directory.

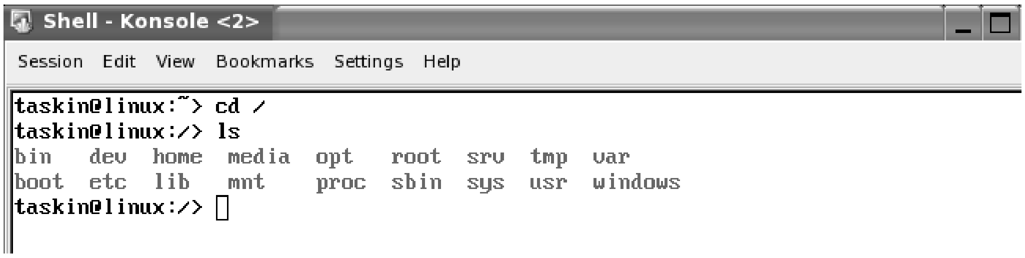

Fig. 3. Standard directories in the root directory. Please note that *root directory* does not mean the *root's home directory*; the root directory is at the top of all directories (as shown in the figure, the *cd* / command takes you from any directory to the top directory). Root's home directory can be seen in the figure with the name of *root*. Notice that, as an exception and apart from the other users, the home directory for the root is not under the *home* directory.

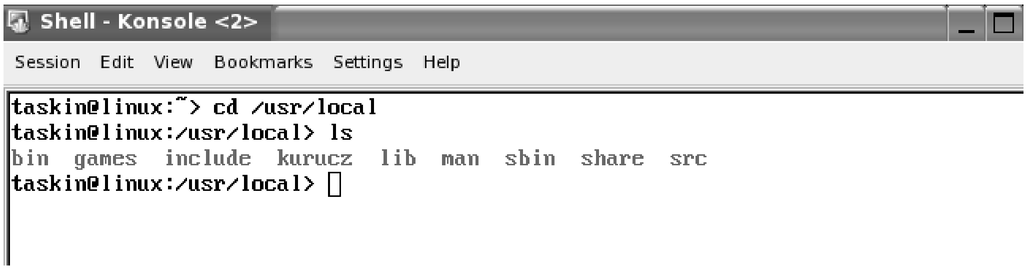

Fig. 4. The */usr/local* directory. Source files can be kept in *src*, compiled or pre-compiled binaries should be put into the *bin* directory. Manual pages, libraries and include files of the installed programs have to be put into the relevant directories. The ATLAS suite can be installed here for a systemwide and clean installation.

2.2.1. Why should I put my programs into the */usr/local* directory?

When you call an application (program) from the command-line by typing the name of the executable, the shell starts to search the executable through *some* directories – it does not look at all the directories. These *particular* directories are said to be ”on the path”. If your executable is not in a directory which is on the path, then the shell cannot find it and cannot start the program.

Thus you should put your executable on the path. The easiest way doing this is to put the executable into *usr/local/bin* directory which is on the path as default and used for this purpose.

Also there are ways to executing a program in other directories which are not on the path:

1-***Using ./ command*** : ./ command tells the shell that not to search path for the executable because it is in the current directory. Just type in the command-line that:

./nameoftheexecutable

but this is an uncomfortable way because the user cannot call the program from other directory than program's directory. To run the program from another directory each time the user has to write the exact adress (full paths) to the program such as,

/home/taskin/atlas/atlas9/atlas9_mem.exe

Sometimes writing the full path in script mode can be useful to ensure that the correct program is running.

2-***Adding the directory on the path***: This can be done by using the PATH environment variable (see Fig. 5). The PATH variable shows which directories will be searched for the executable when a program called from command-line. A directory can be added in two ways:

i-*Temporarily*: Type in the command-line that: export PATH=$PATH:/exact/adress/ofthe/directory.

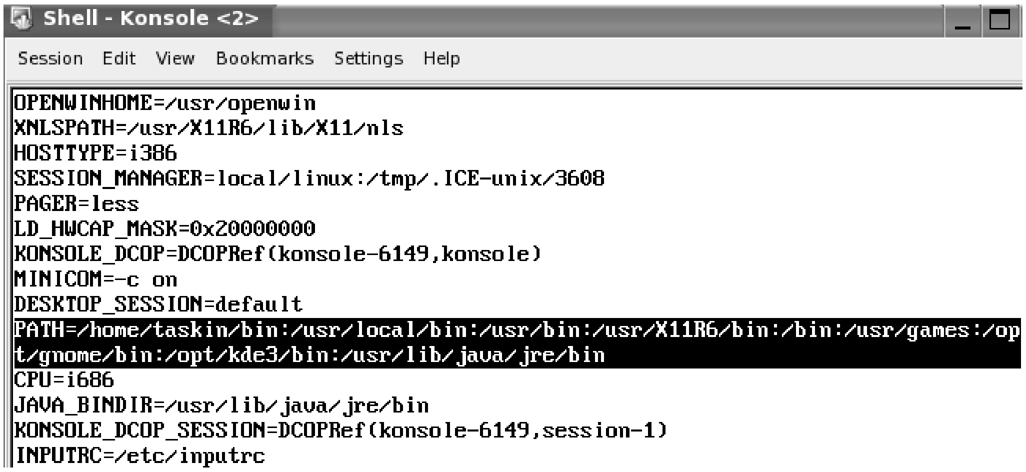

Fig. 5. The shell keeps track of a number of special variables, the *environment variables*. Environment variables can be listed on the screen using command *env* (note that this command is for bash shell; if you are using another shell, then the command can be different). The Figure shows some of the environment variables. Each directory that will be searched for the PATH variable, is separated with ":" sign.

The directory will temporarily be added on the path until you logout from the system.

ii-*Permanently*: Write the same command into the *.profile* file and save. *.profile* file is a hidden file in your home directory (to see it use the *ls* command with *-a* flag in your home directory) and is read by the system each time you logon. Hence, when you logon the system the directory automatically added on the path..*profile* file can be opened by any text editor (for example, *pico, emacs, Kate, vim*... etc). Type the following command in the command-line while in your home directory

pico .profile

2.3. Links

Sometimes a program wants to read a necessary file from another directory apart from the current directory. In this case it might be desirable to put a link to the remote file from the current directory. Such a case can frequently occur while running ATLAS suite; For example, ODF files would probably be in another directory than the working directory of ATLAS9, to save space.

A link can be tought as a second file name pointing the original file and should not be confused with a copy of the original file. Hence, the same file is reachable with two different name. It can be created using the *ln* command:

ln originalfilename secondfilename

This kind of link called as *hard link*. In this case when the original file name is deleted, the file is reachable using the second file name.

Hard links sometimes can be problematic, and for normal users, if there is no other reason, it is highly recommended that using a *symbolic link* instead. A symbolic link can be created using the *ln* command with *-s* flag:

ln -s originalfilename secondfilename

In this case, when the original file is deleted the link is also unreachable (but the reverse is not true).

You can see and use the linked file from the current directory as if it was there, but actually it is not there physically. Since the ODF files require a large amount of disk space, putting a symbolic link to the ODF files instead of copying them into the current directory would be logical.

See the manual pages or more information on the *ln* command.

```
ln -s kapp00.ros fort.1
ln -s p00big2.bdf fort.9
#!ln -s /home/castelli/p00big2.bdf fort.9
ln -s molecules.dat fort.2
ln -s ap00t10000g40k2odfnew.dat fort.3
./atlas9mem_linux.exe<<EOF>ap00t10000g40k2odfnew.out
READ KAPPA
READ PUNCH
MOLECULES ON
READ MOLECULES
FREQUENCIES 337 1 337 BIG
VTURB 2.0E+5
CONVECTION OVER 1.25 0 36
TITLE  [0.0] VTURB=2  L/H=1.25 NOVER NEW ODF
SCALE 72 -6.875 0.125 10000. 4.0
ABUNDANCE SCALE   1.0000 ABUNDANCE CHANGE 1 0.9204 2 0.07834
 ABUNDANCE CHANGE  3 -10.94  4 -10.64  5  -9.49  6  -3.52  7  -4.12  8  -3.21
 ABUNDANCE CHANGE  9  -7.48 10  -3.96 11  -5.71 12  -4.46 13  -5.57 14  -4.49
 ABUNDANCE CHANGE 15  -6.59 16  -4.71 17  -6.54 18  -5.64 19  -6.92 20  -5.68
 ABUNDANCE CHANGE 21  -8.87 22  -7.02 23  -8.04 24  -6.37 25  -6.65 26  -4.54
 ABUNDANCE CHANGE 27  -7.12 28  -5.79 29  -7.83 30  -7.44 31  -9.16 32  -8.63
 ABUNDANCE CHANGE 33  -9.67 34  -8.63 35  -9.41 36  -8.73 37  -9.44 38  -9.07
 ABUNDANCE CHANGE 39  -9.80 40  -9.44 41 -10.62 42 -10.12 43 -20.00 44 -10.20
 ABUNDANCE CHANGE 45 -10.92 46 -10.35 47 -11.10 48 -10.27 49 -10.38 50 -10.04
 ABUNDANCE CHANGE 51 -11.04 52  -9.80 53 -10.53 54  -9.87 55 -10.91 56  -9.91
 ABUNDANCE CHANGE 57 -10.87 58 -10.46 59 -11.33 60 -10.54 61 -20.00 62 -11.03
 ABUNDANCE CHANGE 63 -11.53 64 -10.92 65 -11.69 66 -10.90 67 -11.78 68 -11.11
 ABUNDANCE CHANGE 69 -12.04 70 -10.96 71 -11.98 72 -11.16 73 -12.17 74 -10.93
 ABUNDANCE CHANGE 75 -11.76 76 -10.59 77 -10.69 78 -10.24 79 -11.03 80 -10.91
 ABUNDANCE CHANGE 81 -11.14 82 -10.09 83 -11.33 84 -20.00 85 -20.00 86 -20.00
 ABUNDANCE CHANGE 87 -20.00 88 -20.00 89 -20.00 90 -11.95 91 -20.00 92 -12.54
 ABUNDANCE CHANGE 93 -20.00 94 -20.00 95 -20.00 96 -20.00 97 -20.00 98 -20.00
 ABUNDANCE CHANGE 99 -20.00
ITERATIONS 15 PRINT 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
PUNCH 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
BEGIN                    ITERATION  10 COMPLETED
SCALE 72 -6.875 0.125 10000. 4.0
ITERATIONS 15 PRINT 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
PUNCH 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
BEGIN                    ITERATION  10 COMPLETED
END
EOF
mv fort.7 ap00t10000g40k2odfnew.new
rm fort.*
```

Fig. 6. A sample script for running ATLAS9. The first four lines assign the necessary files to the Fortran input files using symbolic links. Hence, these files can be read by the program from the current directory with the suitable input file name. # sign in the third line tells the shell to ignore this line. The sixth line executes ATLAS9 in the current directory and reads (<< sign) the remaining lines (up to the end of file (EOF)) into ATLAS9 as input. After the model calculation finished, it writes (> sign) the output creating a file named ap00t10000g40k2odfnew.out. Then in last two lines it moves (renames) the fort.7 output file into the ap00t10000g40k2odfnew.new and removes (deletes) all the Fortran input/output files.

2.4. Shell scripts

Linux shells are programmable. This roughly means that several shell commands can be written to a file which then can be executed like a program. Hence, you can avoid retyping a sequence of commands each time for an operation. The following example shows the steps of writing a script:

1- Open a new file. *.sh* or *.com* extention can be added to the filename to indicate that it is a shell script, but this is not a must:

pico example.sh

2- Write the commands. For example, the following commands open a new file named output and write the word *hello* into the file then create a symbolic link between the *output* and *outputlink*. Hence, when the script *example.sh* runs, two files (*output* and *outputlink*) will be created in the current directory.

echo "hello" > output

ln -s output outputlink

3- Exit from the *pico* saving the file.

4- Make the file executable –otherwise the file is just a text (ASCII) file.

chmod + x example.sh

Now the file can be executed,

./example.sh

Because this script is written using the *bash* shell commands, it may not run under other shells (for example under the C shell). To avoid such a problem, the following line can be added to the top line, before the commands:

#!*/bin/bash*

For the other shells, the situation is straightforward; if the shell is written in C shell for example, then this line will be #!*/bin/csh* ... etc.

2.5. Installing a program

A program to be installed can come in two varieties:

1- *As a RPM installation file* [4] : In this case typing

rpm -i filename.rpm

is enough to install the program. Even simply clicking on the file can start the installation.

2- *As a pre-compiled binary or source code*: If it is pre-compiled binary, then the binary has to be put in a directory that is on the path. For example */usr/local/bin* is ideal for this purpose. If there are also additional files like library files to run the program then they also have to be put into a proper directory like */usr/local/lib*.

If the program comes as a source code, it looks something like *filename.tar.gz* or *filename.tgz*; the package is archived as tarball and compressed with gzip. In this case the **general** follows:

i- Uncompress the file. In the case of gzip this can be done typing

gunzip filename.tar.gz

In other cases of compression (for example, .Z or .bz2,...etc.) use the suitable procedure to uncompress the file.

ii- Open the tarball. To do that type

tar -xvf filename.tar

iii- Read the installation instruction file and the readme file. These files generally include important information about the installation of the program.

iv- If there is a *Makefile*, make the necessary changes in this file and save. *Makefile* can be edited using any text editor.

v- If there is a configuration file type

./configure

vi- Type *make*

vii- Type *make all* and/or *make install*.

After all those steps have been completed, the program is installed and should be running properly.

[4] Some of the linux distributions can use different kind of package management systems other than RPM (RedHat Package Management). For example, in Debian Linux case installation file has *.deb* extention and can be unpackaged using *dpkg* utility.

3. A sample shell script for ATLAS9

A sample script for running ATLAS9 is shown in Fig. 6. The reader may prepare his/her own script.

Acknowledgements. This work was supported by the Research Fund of the University of Istanbul. Project number: UDP-529/07062005.

I thank Dr. Barry Smalley who read the draft of this paper and made comments concerning its clarity.

References

Kurucz, R. L. 1993, CDROM 13, 18
Sbordone, L. et al. 2004, MSAIS, 5, 93
Sbordone, L. 2005, MSAIS, 8, 61