UNIX and C

Michael A. Saum

University of Tennessee
Department of Mathematics

Overview

- Why UNIX?
- Choosing an Editor
- The Shell Environment
- Directories, Files, and Processes
- Secure Shell
- Example 1: Basic and Simple
- Example 2: I/O
- Example 3: Dynamic Memory Allocation
- Example 4: Using a *Makefile*
- C, MATLAB, and FORTRAN

Why UNIX?

- The *best* programming environment for C and FORTRAN programming.
- Access to full LaTeX, MATLAB, MAPLE capabilities.
- Not susceptible to Windows virii (87% of all reports of infections during 2002 concerned Windows 32 viruses.)
- A well-patched and well-maintained Unix system is very well-secured against viruses.
- Secure access to fast machines.
- GUI (Graphical User Interface).

Choosing an Editor

- `vi` or `vim`: I use this editor because:
  - I'm used to it and comfortable with the command syntax.
  - The program itself loads quickly.
  - All UNIX machines have `vi`.
  - I don't care about GUI.
- `emacs`: GUI, extremely configurable. There is a learning curve here as with `vi`.
- `pico`: Plain text editor, same interface as `pine` (e-mail client). Non-GUI.
- `xedit`: Plain GUI text editor.
- **Choose One!**

The Shell Environment

- The *shell* is the program that is running and interpreting your commands when you open a terminal window.
- There are various shells available, `zsh` is the default for most users in the Math Department. Other shells include `csh`, `bash`, and `ksh`.
- The environment is a collection of *environment variables* which aid the shell in identifying defaults for various application programs.
- To see what environment variables are defined, type the command `env`.
- To get out of the shell, type `exit`.

The Shell Env., contd.

- To add your own environment variables, you can create (using your favorite editor) a file in your home directory named `.zshrc`
- An important environment variable one should set is the `PRINTER` environment variable. Add the following line to `.zshrc`:

```
export PRINTER=A001
```

- Other printers commonly used are A107, A317, and A317-duplex.
- One can write scripts which utilize what is called a *shell programming language*; different shells, different syntax. Access to environment variables is available.

Directories

- All UNIX directory structures are *tree structures* and have the following directories in common:

/	The **root** directory
/bin	Commands needed for minimal system operability
/etc	Critical startup and confi guration fi les
/lib and /usr/lib	Libraries for the C compiler
/usr/include	C Header fi les
/tmp	temporary fi le space (non-permanent!)
/home	yours and others home directories
...	many more!

- Directories are considered to be *files* of a special type.
- **Allways know where you are in the tree!**

Directory Commands

Managing directories is essential to productive programming on UNIX. *Note: Use the* `man` *command when you don't know exactly how a command works.*

Command	Description	Common options
`pwd`	Print Working Directory	see **man** pages
`mkdir`	Make a directory	see **man** pages
`rmdir`	Remove an empty directory	see **man** pages
`cd`	Change to a directory	no options - returns to home directory; `cd -` returns to directory you last issued `cd`; `cd ..` returns to the next higher hierarchial directory level
`ls`	List directory contents	see **man** pages, `ls -alrt` lists directories with maximum information on files, in reverse time sorted order
`du`	List directory disk useage	see **man** pages, `du -H` displays disk useage in Human readable form

Regular Files

- Regular files are just a *bag o' bytes*.
- Regular *hidden* files begin with a `.` (for example, `.zshrc`)
- `*` and `?` are *any* and *single* wildcard characters and can be used in any file name pattern (they are *expanded* by the shell). For example `*.c` would match any files that have a `.c` extension.
- There are two types of regular files, binary and ASCII.
- `file file_name` will tell you information about what type of file `file_name` is.

Files, contd.

- Common Extensions

Binary	executeables, `*.o`, `*.dvi`, `*.gz`, `*.z`, `*.tar`, image files, sound files, movie files, ···
ASCII	Text files, shell command scripts, `*.c`, `*.h`, `*.m`, `*.f`, `.*.cpp`, `*.ps`, `*.pdf`,`*.tex`, `*.html`, ···

- If you bring a file up in your favorite text editor and it looks like garbage, it is probably a binary file. Exit without saving.

File Commands

- Know the options available for each command **before** using the command.
- File Commands

Command	Description	Common options
`ls`	List fi le sizes	see **man** pages; `ls -l`
`rm`	Remove fi les	see **man** pages; **WARNING: This really removes files from the system!**
`tar`	Archive fi les and directories into single fi le	see **man** pages
`gzip`	Compress fi le	see **man** pages; Saves disk space!
`zip/unzip`	Combo of tar and gzip	see **man** pages; Saves disk space!
`grep`	Search fi le for text pattern	see **man** pages; Very useful!
`find`	Find fi les which meet some criteria	see **man** pages; Very useful!

Processes

- Any program that is *running* in UNIX is considered to be a process.
- The operating system determines *what process* uses the CPU and other resources at *what time*.
- The command `top` will provide one with a continuous updated display of system resource utilization (i.e., why is my program running so slow).
- The command `ps` tells one which processes are currently running under the current shell.

Processes, contd.

- Associated with each process is a process id number (*PID*), which are displayed from both top and ps.
- To stop a runaway process, use the command `kill -9 PID`.
- Another nice command is `pstree`, which provides a different *picture* of the processes running than either top or ps.

Misc. UNIX Commands

- `lpr` is the command which prints files to a printer.
- Postscript files print fine on all printers in the Math Department.
- Raw text files sometimes print poorly on our printers, it is better to convert to postscript first or use `a2ps` or `enscript` or `mpage` to print.
- Use `acroread` to view `*.pdf` files.
- Use the **man** pages and the internet to find out more information on UNIX commands.
- Use `cat` to type an ASCII file to the display.
- Use `more` to view an ASCII file one screen at a time, `q` quits more.

Secure Shell

- SSH allows one to move securely from one machine to another with a single login. The only way to remotely log into Math Department machines is through `ssh`. The only two machines which allow remote access are `goliath.math.utk.edu` and `mathsun1.math.utk.edu`.
- `scp` allows one to copy file(s) from one machine to another.
- `sftp` also allows one to copy file(s) from one machine to another.
- Other *fast* machines in the Math Dept are `agnesi`, `fubini`, `fatou`, and `turing`.

Secure Shell, contd.

- `ssh` configuration (One time only)
  1. `ssh-keygen` Generate's unique key, requires passphrase.
  2. Ensure that the following files are in `~/.ssh2` subdirectory: `authorization` and `id_dsa_XXXX_a.pub`, where XXXX may be 2048 or 1024.
  3. `authorization` should contain the line:

     `key id_dsa_XXXX_a.pub`

Secure Shell, contd.

- You should ensure that the above two files in your `~/.ssh2` subdirectory are copied to your `~/.ssh2` subdirectory on `mathsun1` (currently filesystems are not shared between `mathsun1` and the rest of the department machines.)
- After Logging into a Math Department UNIX machine, open a terminal window and type: `ssh-add`. You will be prompted for a passphrase and should enter the passphrase you entered when you ran ssh-keygen.

Secure Shell, contd.

- As long as you are logged onto the machine, you may *jump* to any other machine in the Math Department by just typing:

  ```
  ssh machine_name
  ```

- *OpenSSH* issues. There are some incompatibilites between different software implementations of `ssh`, the main problems coming with the *OpenSSH* implementation. Beware! (Contact Ben Walker or myself if you are running into problems here.)

Algorithms and Language Choice

- It is always wise to spend time before programming to determine the algorithms to be implemented and the language to implement them in.
- If one has quick *what if* programming, then usually MATLAB is the best choice.
- If one has vector or matrix intensive work, MATLAB may be the best choice.
- If one desires nice integrated graphics capability, MATLAB may be the best choice.
- If one has to optimize for speed or memory, then C or FORTRAN is probably the best choice.
- If one has to link with other libraries and applications, C is probably the best choice.
- If one has to have the program work on a wide variety of platforms, C is probably the best choice.
- **Choose wisely!**

Program Organization

- One has to determine *a priori* the general overall structure of a program.
- Every program has a *main* function or entry point.
- For most computational programming projects, top-down programming as opposed to object oriented programming is preferred.
- Modularize your program! It is much easier to *debug* five lines of code in a subroutine as opposed to debugging 100 lines of code in the main routine.
- Be consistent in naming of variables, use variable names which make sense.
- Do not be afraid to comment your code.

Managing the Projcet

- Keep separate programs in separate directories.
- Archive older versions of your program so if you have to retrieve an earlier version, you can.
- Reuse code you have written before when you can.
- One can use `a2ps`, `enscript`, or `mpage` to format and print your source code.

Edit-Compile-Run

- The Edit-Compile-Run cycle is very familiar to all programmers.
- Make sure you are testing against problems where you know what the solution should be.
- Modularization makes benchmarking small segments of code much easier.
- No program ever works the first time.

Example Problem

- Suppose one is faced with the question:

  What is the distribution of dominant eigenvalues for Hilbert Matrices?

- This question serves as an excellent starting point to illustrate application development along with how to do different tasks in the **C** language.
- The approach here will be to utilize the *Power Method* to determine the dominant eigenvalue and associated eigenvector for a given matrix A.

Power Method

Require: Square Matrix $A : n \times n$.

Require: Starting Vector $X_0 : 1 \times n$.

Require: Tolerance ϵ, Maximum number of iterations M.

$X = X_0$

for $(i = 0; i < M; i++)$ **do**

$Z = AX$

$s = ||Z||$

$Z = (1/s)Z$

if ($||Z - X|| < \epsilon$ OR $||Z + X|| < \epsilon$) **then**

EXIT LOOP

end if

$X = Z$

$\lambda = \frac{Z^T AZ}{Z^T Z}$

end for

OUTPUT: Dominant eigenvalue λ, associated eigenvector X.

Example 1: Basic and Simple

- The first step is to ensure that one can calculate the scaler vector product of two vectors. This code is shown in `ex1.c`, the listing of which is included at the end of this presentation.
- To compile `ex1.c`,

  ```
  gcc ex1.c -o ex1
  ```

  will produce an executeable file named `ex1`. Note that if the `-o ex1` is not present, then the executeable will be named `a.out` by default.
- To run the program

  ```
  ./ex1
  ```

Example 1: Comments

- Lines 1-8 are comments.
- Lines 11-12 includes provide standard definitions and declarations for any **C** program.
- Line 15 is necessary for any standalone **C** program.
- Lines 18-22 declare and initialize *local* variables used in function `main`. If these lines were placed before line 15, they would be considered *global* variables and are visible from within any function defined in `ex1.c`.

Example 1: Comments, contd.

- Lines 20-21 define two vectors of length 10. In **C**, indexing to vector elements begins with 0 (contrast to MATLAB and FORTRAN where indexing begins with 1).
- Thus, to index over all elements in the vectors (line 28), the range of the i values in the **for** loop ranges from 0 to 9 (10-1).
- Line 25 illustrates how to print a character string.
- Lines 28-30 illustrate the basic **for** loop. Note that Line 29 is equivalent to the statement

```
a = a + c[i]*d[i];
```

Example 1: Comments, contd.

- In line 33, the `%g` format indicates that one is not too particular about how the number `a` is to be printed out. Other formats are `%e` (exponential) and `%f` (single precision floating point) and `%d` (integer), with various modifiers to indicate width and how many decimal places.
- In line 33, the `\n` format specification indicates that a *newline* is to be introduced in the output.
- Line 36 ends the program. By convention, if one returns a number other than 0, the program `ex1` is assumed to have not ended in a normal fashion.

Example 2: I/O

- Now we will modify `ex1.c` to produce `ex2.c`, which illustrates how modifying existing code can help to make the program more useful.
- Here we will add the following functionality to our program:
  - Read in Matrix and Vector from input data file.
  - Define and use functions in a modular fashion.
  - Perform Matrix Vector Multiply.

Example 2: Comments

- By creating a separate directory `ex2` with the same parent directory that `ex1` has, one can copy `ex1.c` in `ex1` to `ex2.c` in `ex2`. Thus, one can use `ex1.c` as a *template* upon which modifications will be made to produce `ex2.c`.
- Lines 15-17 define *Function Prototypes* which describe the general form of these functions which will be detailed later in `ex2.c`. These declarations should go before the *main* function declaration. Their purpose is to allow the **C** compiler to detect errors in passing parameters or return values during the compilation process.

Example 2: Comments, contd.

- Line 15 declares a prototype which does not return any value `void`, but passes three arguments to a function called `ReadInp`. Arguments 2 and 3 are passed as *pointers*, i.e., addresses of the variables to be passed. Note that in function prototype declarations, there is no need to give variable names, just the types of the return values and the arguments.
- Note that in line 16, the function `MyDotProd` returns a value of type double.
- Line 24 declares a two dimensional array or matrix A which is dimensioned 4×4.
- Line 32 calls the function `ReadInp` which is defined in Lines 54-70.

Example 2: Comments, contd.

- Line 35 calls the function `MatVecMult` which is defined in Lines 91-102.
- Lines 38-45 prints out the results. Note the width and format specifiers in the `printf` statements.
- Note that the function `MyDotProd` has both a function prototype and the source code is present (Lines 76-86), but is never called from anywhere. This is ok, because we will be using it later and it made sense to modularize after working on `ex1.c`.

Example 2: Comments, contd.

- In any **C** program, there are three files which are always set up: `stdin, stdout, stderr`.
- By default, `scanf` reads input from `stdin` and `printf` prints output to `stdout`. The shell assigns `stdin` to the keyboard and `stdout` to the screen.
- These defaults can be changed at runtime through what is called *redirection*.
- To compile `ex2.c`,

```
gcc ex2.c -o ex2
```

Example 2: Comments, contd.

- To run `ex2`, assuming an input file has been created called `ex2.inp` (listing provided in handouts)

```
./ex2 < ex2.inp > ex2.out
```

  will read in data from `ex2.inp` and write all output to a file `ex2.out`.
- This allows great flexibility in redirecting `stdin` and `stdout`.
- Modularization is good.
- It is a good idea to make the `main` program as clean and readable as possible.

Example 3: Memory Allocation

- Now we will modify `ex2.c` to produce `ex3.c` adding more functionality.
- Here we will add the following capabilities to our program:
  - Control Input File.
  - Introduce *global variables*.
  - Two Dimensional arrays.
  - Dynamically allocate memory for variables.

Example 3: Comments

- In the previous example, two dimensional arrays were clumsy to deal with, i.e., the arrays were dimensioned in the main function with a fixed size.
- This example illustrates how one can dimension any variable storage only when one needs it, which allows for one to more closely manage variable storage.
- In addition, if one has a function which requires a long argument list, the code can usually be made more manageable by turning *local* variables into *global* variables.

Example 3: Comments, contd.

- To compile `ex3.c`,

```
gcc -o ex3 -lm ex3.c
```

- The `-lm` option to `gcc` indicates that the *math* library should be made available (that is where the `sqrt` and `pow` functions reside).
- Note that **man** pages exist for most **C** built-in functions.
- There is no need to redirect the input file to `stdin` as the code is reading in a specific file name.

Example 3: Comments, contd.

- Line 13 specifies the `math.h` file should be included (needed for sqrt and pow functions).
- Lines 21-26 declare global variables, which are visible from within any function defined in `ex3.c`.
- Line 21 declares a variable of type `FILE *`, i.e., a pointer to a file.
- Note the assignment of a variable to `NULL` is basically setting it equal to zero.
- One can think of & as the *address-of* operator in **C**.

Example 3: Comments, contd.

- Lines 81-83 make a call to `malloc` which requests a certain chunk of memory from the system.
- Lines 16,39, and 68 all declare `ReadInp` to be a function which does not return anything and nothing is passed to it as arguments.
- Lines 54-55 use sqrt and pow functions.
- Lines 72-75 test to see if the input file can be opened. If it can't be opened for input, the program stops completely.

Example 3: Comments, contd.

- Line 78 reads the first data item in the input file and stores in the variable n. Note that `fscanf(stdin, ...)` is the same as `scanf(...)`.
- Note that the call to allocate storage for the two dimensional array A is being requested for n^2 elements, each of the storage size associated with the data type `double`, or $8 \times n^2$ bytes.
- Lines 88,127 indicate how the one dimensional vector A is utilized as a two dimensional array, indexed by i and j.

Example 4: The Power Method

- Now we will modify `ex3.c` to produce `ex4.c` adding more functionality.
- Here we will add the following capabilities to our program:
  - The Power Method algorithm implementation.
  - Use of *Makefile*.

Example 4: Comments

- Lines 20-35 have additional function prototypes and global variables defined.
- Line 24 defines an output file where all `stdout` output will be directed.
- Line 44-47 open the log file for output.
- Lines 56-79 define an *outer loop* defining what size matrix to deal with.

Example 4: Comments, contd.

- Lines 59-63 set up the Hilbert Matrix A. Note in line 61 how the integer result $(i + j + 1)$ is being *cast* as a double.
- Line 70 calls a function to perform the Power Method on A, returning how many iterations it took to achieve convergence.
- Lines 139-174 define an *inner loop* with a `while` loop construct. The only way execution gets out of this loop is if the while criterion is satisfied (max iterations exceeded) or if convergence is obtained (lines 162-164).

Example 4: Comments, contd.

- In line 163, the `break` command exits the most recent loop.
- The variable lambda (declared local to function main in line 41) is passed to `PowerMethod` not by value but by the pointer to its storage using the & operator. When it is modified in line 172, it is changed not by value but by the pointer `*` operator. Thus, on return from `PowerMethod`, lambda has a new value and the local variable has been changed by the function.

The Makefile

- Use of a Makefile simplifies the Edit-Compile cycle by defining a set of rules which when coupled with dependcies of different source files can determine what has change and only recompile those programs.
- Makefiles are essential for large programming projects with multiple source code files.
- With a Makefile, compilation can be initiated without leaving the vi or emacs editors. This speeds up the Edit-Compile-Run cycle tremendously.

Makefile, contd.

- When writing a Makefile, realize that anything that is indented must be indented with a tab and not spaces.
- You may use the following Makefile as a template for simple programming projects.
- If you have a Makfile, all one has to do is type `make` to recompile the program. If one wants to remove all object files (`*.o`) and executeables, type `make clean`.

Example 4: Makefile

```
#
#  Makefile
#
#
SRC = ex4.c
LIBS = -lm
CFLAGS = $(INCS)
CC = gcc
EXECUTABLE = ex4
OBJS = ex4.o

ex4 : $(OBJS)
        $(CC) $(CFLAGS) -o $(EXECUTABLE) $(OBJS) $(LIBS)

clean:
        -/bin/rm -f $(EXECUTABLE) $(OBJS)
```

C, FORTRAN, and MATLAB

Thanks to Dr. Alexiades, a comparison of the different syntactical elements of C, FORTRAN, and MATLAB is provided in the supplemental handout material.

Summary Remarks

- If you are serious about programming in **C**, obtain a good **C** reference book.
- The definitive book is *The* **C** *Programming Language* by Brian Kernighan and Dennis Ritchie (They were instrumental in the writing of the first **C** compiler.)
- There are plenty of tutorials and references on the internet for any issues related to UNIX, C, or MATLAB.
- The next seminar will discuss what to do with the data a program generates, i.e., graphical display and post-processing analysis.