Lab 0: Acclimation

ELEN E6884/COMS 86884: Speech Recognition

Due: September 16, 2005 at 6pm

1 Overview

The goal of this assignment is for you, the student, to become acclimated to the programming environment we will be using for the exercises of this course. The lab consists of the following parts:

- **Part 0: Learn basic UNIX commands and text editing**
- **Part 1: Get an ILAB account and log on**
- **Part 2: Set up your ILAB account**
- **Part 3: Familiarize yourself with the programming conventions used in this course**
- **Part 4: Compile a program using** SMK
- **Part 5: Learn how to debug with** GDB

All parts of this lab are required. (Note: you won't be able to access the file mentioned in the next paragraph until you log into your ILAB account in Part 1.)

All of the files needed for the lab can be found in the directory `~stanchen/e6884/lab0/`. Before starting the lab, please read the file `lab0.txt`; this includes all of the questions you will have to answer while doing the lab. Questions about the lab can be posted on Courseworks (`https://courseworks.columbia.edu/`); a discussion topic will be created for each lab. **Note:** The hyperlinks in this document are enclosed in square brackets; you need an online version of this document to find out where they point to.

2 Part 0: Learn basic UNIX commands and text editing

For those of you who are not familiar with Linux or other variants of UNIX, you need to learn how to use UNIX for basic tasks such as making directories, moving/copying files, redirection, etc. and for editing text files. You're basically on your own for this, but we did some quick Google searches and here are some pointers:

- Here is a [UNIX tutorial] we found in Google which looks reasonable. Type "UNIX tutorial" or some other relevant string into Google for pointers to more material.
- To edit text files, the two most popular tools in UNIX are EMACS and VI. The editor VI has weird key mappings, but is simple and compact. The editor EMACS is extremely powerful and also has some weird key mappings, but these can be largely avoided by running EMACS under X windows, in which case EMACS acts pretty much like a generic text editor.

  Here is an [EMACS tutorial] and a [VI tutorial]; use Google to find out more. You can also try the built-in EMACS tutorial by typing `emacs` and hitting the keystrokes "Ctrl-h" and then "t". For the uninitiated, if you are running X windows, our recommendation is EMACS; otherwise, choose one randomly and switch if you don't like it. If you don't know if you are running X windows, ask the person sitting next to you.

3 Part 1: Get an CISL/ILAB account and log on

In the first class, you should have submitted information that will let us create a computer account for you on the ILAB computer cluster (also known as a "CISL" account). If you haven't done this, contact one of the professors ASAP to get this process started.

When an account has been created, a username and password will be E-mailed to the address that was provided to us. This may take some time; we will give you an update at the next lecture if the accounts have not been created by then. If it takes overly long for the accounts to be created, the deadline for this exercise will be extended.

Once you have a username and password, try to log onto a machine in the ILAB cluster. The cluster is located at Mudd 1235 but you need badge access to access the room, so most of you will need to log in remotely. To log in remotely, use SSH as TELNET is not supported. If you don't have SSH, visit the [OpenSSH web site] for more information on installation and such. The machines are named `micro1.ilab.columbia.edu` through `micro15`.

If you are able to log in successfully, you have completed this part of the lab.

4 Part 2: Set up your ILAB account

In this section, we discuss the things you need to do to set up your account for this course. By default, you will be assigned the shell BASH. If you don't know what I'm talking about, you are probably using BASH. If you are using a different shell, then you will have to adjust the commands in this section appropriately, but if you are using a different shell, you should know how.

First, check if you already have a `.bash_profile` and/or `.bashrc` file in your home directory (*e.g.*, via `ls -a ~`). For the ones that don't exist, copy the versions of these files from `~stanchen/e6884/lab0/`, *e.g.*:

```
cp ~stanchen/e6884/lab0/.bash_profile ~
cp ~stanchen/e6884/lab0/.bashrc ~
```

If one or both of these files already exist, manually merge the contents of the version we supply with the existing version. Also, copy over or merge in the file `.gdbinit` from the same directory.

You can type `. ~/.bashrc` (or logout and login again) to have these changes take effect.

5 Part 3: Familiarize yourself with the programming conventions used in this course

To get yourself familiarized with what the programming exercises will be like, we will go through a mini-exercise. To make it possible for you only have to write the interesting bits of code in a speech recognizer for the labs, we have written extensive amounts of "glue" code. Each program will be compiled from a large number of C++ files, but almost all of these files have already been written for you. We will just leave out parts from a file or two that you will have to fill in.

To see what this is like, let's get started on the mini-exercise. First, create a new subdirectory for us to work in and go there:

```
mkdir -p ~/e6884/lab0/
cd ~/e6884/lab0/
```

Next, let's copy over a couple files that we will need:

```
cp ~stanchen/e6884/lab0/Lab0_FE.C .
cp ~stanchen/e6884/lab0/.mk_chain .
```

The file `Lab0_FE.C` is the C++ source file that you will be editing for the exercise. The file `.mk_chain` is a file that we will need for compilation; it holds where all the other source files that we be compiled in are located.

Now, open the file `Lab0_FE.C` in a text editor. You might notice that there are a bunch of weird contructs in the file that you don't understand. Don't freak out yet; there will be plenty of time for this later. Look for the markers `BEGIN_LAB` and `END_LAB` near the end of the file. This is the only section of the file you need to understand. The rest of the file can be ignored, though you may want to read the comments there for your own edification. (If you are interested in exploring the related header files and source code, look in the following directories:

```
~stanchen/pub/zeeapi/inc/
~stanchen/pub/zeeapi/src/
~stanchen/pub/zeelib/inc/
~stanchen/pub/zeelib/src/
~stanchen/pub/e6884/inc/
~stanchen/pub/e6884/src/
```

For example, `Lab0_FE.H` is located in `~stanchen/pub/e6884/inc/`.)

In this exercise, you will be writing a simple signal processing module that takes as input a 2-D array containing a vector of floating-point numbers (or *features*) for each time unit (or *frame*) in a speech signal, and outputs a scaled version of the array. Since this is Lab 0, we are going to tell you what the answer is. Type/paste in the following code between the `BEGIN_LAB` and `END_LAB` markers:

```
for (int frm = 0; frm < inFrames; ++frm)
    {
    for (int dim = 0; dim < inDim; ++dim)
        outBuf[frm][dim] = inBuf[frm][dim] * scaleFactorM;
    }
```

Notice that the variables `outBuf` and `inBuf` behave like 2-D arrays in C. In reality, they are C++ objects, but you don't need to worry about this. Also notice that these arrays have already been sized correctly; we will do this for you whenever possible to make your life easier. The scaling constant `scaleFactorM` has also been mysteriously initialized for you. In fact, this parameter can be set on the command line of the programs that this file will be compiled into, but again, how this happens does not concern you at this time. Anyway, we are now done with the programming portion of this exercise.

In terms of the bigger picture, we can view signal processing in ASR as being comprised of a number of processing steps applied in sequence. Each processing module takes the matrix of values produced by the last module (consisting of feature values for each frame in an utterance) and generates a matrix of values to be fed to the next module. The above example implements a module that does simple scaling; in Lab 1, you'll be implementing a number of modules needed in producing MFCC features.

6 Part 4: Compile a program using SMK

Now, we are going to compile the file `Lab0_FE.C` we have just completed into the executable FETOOL, a program that prints out feature vectors for utterances. The traditional thing to do at this point would be to create a `Makefile` describing which files need to be compiled and linked together to form this executable. However, at IBM we have developed a program named SMK that uses artificial intelligence, fuzzy logic, and support vector machines to automatically decide which files need to be linked together, and this program has been made available to you, the student, through the magic of a license agreement.

To compile the program FETOOL, simply type

```
smk FETool
```

For this to work, you need to have the file `.mk_chain` that we told you to copy earlier in your current directory, and the directory `~stanchen/pub/exec/` must be on your path (which should be the case if Part 2 was completed successfully). Also, you should lean slightly closer to your terminal, because the program attempts to use ESP if its other algorithms fail. At this point, lots of gibberish should be printed on your screen, including the compilation and link commands that SMK is executing. If there are any compilation errors for `Lab0_FE.o`, fix them; otherwise, the executable FETOOL should be created in your current directory. For more information on SMK, refer to the [SMK User's Guide].

Now, try running FETOOL with no arguments. It will print out usage information, including a description of all of its command-line arguments. Mastery of the user interfaces of speech recognition tools is notoriously hard to come by, so instead of attempting to foist this knowledge upon you, we will provide shell scripts that supply the appropriate command-line arguments for each experiment that you will need to run. Nevertheless, it is hoped that some of you, by examining these scripts and possibly some source code, will some day be able to figure out how to devise command-line arguments by yourselves.

For this lab, run the script `lab0p4.sh` by typing

```
lab0p4.sh
```

This file is also in `~stanchen/pub/exec/`, which should be on your path. This script prints out feature values for some example speech signal, using the code in `Lab0_FE.C` in the processing of the signal.

Save the output of this run in the file `p4.out` by typing

```
lab0p4.sh > p4.out
```

You will have to submit this output to us, as described in `lab0.txt`.

(To give a little more detail of what FETOOL does, to use it you must specify where a set of waveform files are located and what signal processing modules to apply and in what order. It then iterates through the waveform files, applies each of the specified signal processing modules to the waveform in turn, and then prints out the resulting feature values.)

7 Part 5: Learn how to debug with GDB

For those of you who are not robots or cyborgs, you will undoubtedly introduce bugs into your source code at some time or another. *Debuggers* are tools that facilitate the finding of bugs, and it will make your life much easier if you learn how to use one. In this section, we will introduce the GNU GDB debugger, though you are welcome to use whatever debugger you would like (hint: change the values of the `DBX` and `DFLAGS` environment variables). We strongly suggest against using the null debugger, *i.e.*, no debugger at all.

Normally, to start GDB you would type `gdb` *program*. Within GDB, you would type `run` *arguments* to start the program. However, to make it so you don't ever need to type in the prodigious argument lists used in the runs in the exercises, we support a different start-up procedure. With each script (*e.g.*, `lab0p4.sh`), we support the flag `-debug` which starts up the given program inside of GDB. In addition, all of the command-line arguments are cached somewhere so that you only need to enter `run` (or simply `r`) to start the program.

To learn the basics of GDB, check out this [GDB tutorial] or find other tutorials on the web via Google; you can also get some documentation within GDB by typing `help`.

For this part of the lab, you will need to answer the question posed in `lab0.txt`. To do this, we suggest you learn how to set a breakpoint (via the `breakpoint` or `b` command) and how to print the value of a variable (via the `print` or `p` command). Some other commands that you should know about (just on general principle) are `cont`, `step`, `next`, `where`, `up`, `down`, and `finish`.

Hint: it's generally easier to set breakpoints on line numbers rather than function/method names. If you do want to set a breakpoint on a method name, the syntax is

`b` *namespace*`::`*class*`::`*method*

You only need to specify *namespace* if the class is in a namespace. In our case, the class `FEScale` is in the namespace `Zee`, as will be most classes we will define in the labs. Another hint: learn the distinction between `step` and `next`. (In most cases, `next` is the more useful command.)

In Lab 1, it will be very useful to be able to examine the contents of objects like `inBuf` when in the debugger. You might want to see what the following commands do within GDB (when stopped in a context where `inBuf` is defined):

```
p inBuf.get_frame_count()
p inBuf.get_dim()
p inBuf[12][13]
p inBuf[12][13]@5
```

For future reference, in the interest of preempting debugging questions, here are three procedures you should perform before asking for help in finding a bug:

- **code review** — Carefully read each line of code to make sure it says what you intended it to say. People make a surprising number of essentially typographical mistakes.
- **data review** — For each variable with a nontrivial lifetime, read the code and make sure the variable is constructed, initialized, updated, and destructed correctly.
- **step through the code** — Step through each line of code in a debugger, examining variables to make sure they have the values you think they should have. You may need to step through the same code multiple times, to test different situations that may arise.

8 What is to be handed in

Make a copy of the ASCII file `lab0.txt` from the directory `~stanchen/e6884/lab0/`. Fill in all of the fields in this file and E-mail the contents of the file to `stanchen@watson.ibm.com`. This file also has instructions on how to submit the files created in Part 4.