CADLAB Week 9 Assignment: Introduction to Scilab

Hari Ramachandran, EE Dept, IIT Madas

12th September 2006

1 Introduction

Last week we were introduced to a technical editor, *LyX*. What we will do this week is to learn about a new tool, namely *Scilab*, and how to integrate *LyX* and *Scilab* together to make a great team.

By the end of this course, we hope to have you

- Creating "Literate" programs in *LyX*, and running and debugging the same using *Scilab* or *gcc/gdb/ddd*, etc.
- Creating circuit diagrams in *Xcircuit* and analysing the same in *spice* and *Scilab*. And importing these diagrams into *LyX*.

The idea is that today's engineer is not just a master of individual tools, but is a master at combining tools to create virtual tools that are even more powerful.

Need for Documentation

It is very important, as an engineer, to be conversant with the analysis and design tools of our profession. It is perhaps, *even more important*, to be able to communicate our findings to others. Many very competent engineers are unable to explain the logic of their programs to others.

You are all new to the world of programming, and this may not seem to be a major problem to you. However, in industry and in research, projects work in the following way:

1. Someone, (say a professor), comes up with an idea to tackle a particular problem. The idea needs to be looked at, to decide whether it is sensible, leads anywhere, and if it can be developed into a something useful.
2. The person first analyses the problem theoretically. Concludes that the idea is worth implementing.
3. An engineer writes code to simulate the problem, as well as the solution represented by the idea. Assuming that he is a competent fellow, he succeeds.
4. Rarely, a product results. Usually, the work stops here, since the idea by itself is not enough.
5. The engineer moves on to other tasks, and the idea sits there, waiting for more ideas, or waiting for the right problem.
6. The right conditions appear, and the idea is to be dug out and used. If we are very lucky, the engineer himself is dug up and reused. But frequently he has left and only the code remains. Another hapless engineer (you!) has to take up the code and modify and improve it and achieve the desired outcome.
7. But!
   (a) The code was written in a language that is now obsolete (say Fortran IV).

(b) The current version of the gcc compiler no longer compiles the code.

(c) The code compiles and runs for the original problem. But when you changed the input values to reflect the new problem, the code crashes.

(d) The code finally works, but now you need to modify it to make it work for the new problem - and you can make neither head nor tail of the original algorithm.

8. At this point, the approach is frequently to dump the code and start all over again. Painful! The question is whether something could have been done to avoid this problem?

The answer is that the link between the algorithm and its implementation (the code) needs to be documented. Unfortunately, documentation is the most hated occupation of people who makes things, i.e., engineers. What to do?

Note that I am not speaking about user documentation - i.e., documentation meant to teach non-technical people how to use a software. This is the documentation needed by other engineers to understand your work.

2 Literate Programming

The best (and in my opinion, the only) solution that has been discovered is to have the documentation done by the programmer, *while coding*. At the point when he is writing his program, the programmer is usually very clear why his code fragment is written the way it is. And he is even eager to tell everyone how clever he is. If he could only document it right then and there, the future of this piece of software will be secured.

But … there is always a but. Programming languages rarely make concessions to visually appealing documentation. We can, and should, put in comments inside our codes. However, comments are rarely sufficient for good quality documentation. The reason is that the logic of code is best apparent when it is not cluttered by comments, yet the code needs comments to explain its function. Additionally, the extras of good documentation - equations, tables, figures and plots cannot be put into the code this way.

Enter literate programming. In this form of programming, you create the program in LyX (as an example). The program is embedded inside a report that discusses the algorithm, has equations and figures and plots. Any crucial piece of the code can have a flowchart or pseudo code to explain it. When the program needs to be compiled and executed, we merely select "File→export→C" to generate the C code which is to be compiled and run.

Side Benefits of Literate Programming

Perhaps the most interesting side effect of Literate Programming is *faster and cleaner coding*. This is most unexpected. Here we are, spending time creating equations and tables and figures explaining what our code is supposed to do, and it speeds up coding?

Yes it does, and quite dramatically too. For very small programs, this effect is not visible. A 10 line program is self-documenting. Literate Programming does not really help. But when we cross the 100 line mark in C or Perl, programs become very obscure. As a consequence, they become difficult to debug. "Debugging" a program is the process by which you correct little errors in an apparently correct program to make it perform as designed. How can one debug an obscure code? On the other hand, if every thing in the code is described well, debugging becomes a pleasure.

But there is an even greater payback. When we do Literate Programming, we usually write an introduction before we start on the code. We import the algorithm (probably it was written in LyX too). We put down the pseudo code. *Then we start coding*. This prior planning is known to cut coding time greatly. Most programming bugs come from ad-hoc programming. You have an idea and start putting down code, planning as you code. Sometimes the plans are not thought through and bugs result. If you planned first, and then coded, errors are rare.

Literate Programming in this Lab Course

Traditionally, literate programming has been done with a text editor, creating a document in a markup language (similar to LaTeXor HTML). Tools convert this document into code and report. Perhaps one of the most interesting things about *LyX* is that we can do literate programming right here, and achieve the same results.

In the following weeks, we are going to learn how to integrate programming and documentation and how to document our code even as we program. In the first part of today's lab, let us set up *LyX* to handle literate programming.

1. Go to your home directory, and change to the *LyX* configuration directory by executing

   ```
   cd
   cd .lyx
   ```

2. Backup your existing "preferences" file:

   ```
   cp preferences preferences.old
   ```

3. Get the file "preferences" from the course website. This file has been set to include the literate programming settings.

4. Edit the file and change every place "~" appears to the address of your home directory. This is quite important, for otherwise *LyX* will not work fully. Copy it to the ".lyx" directory:

   ```
   cp preferences ~/.lyx/preferences
   ```

5. Create the directory "~/tmp/lyx", where *LyX* creates auto save copies of your files.

   ```
   mkdir ~/tmp
   mkdir ~/tmp/lyx
   ```

This should now make your *LyX* ready for literate programming. I have put in entries for creating *C*, *Perl* and *Scilab* programs.

Note: Please turn on "instant preview" in your *Edit* -> *Preferences* -> *Graphics* page. This will allow formulae to appear correctly.

One potential problem that can rise is when you have your own customizations already in place. That is why we saved your old preferences file. Compare the old and the new preferences file and make sure that any preferences you added to *LyX* are added back to the new preferences file.

In the website is an example literate programming file, that implements a problem in *Scilab* that computes the PDF of crosstalk in optical channels:

```
sample-scilab.lyx
```

This file can be loaded into your tweaked version of *LyX* and you will see what seems to be a report with fragments of code in it. Select *View→scilab*. *Scilab* will start up and run the program. To have a copy of the code, invoke *File→Export→scilab*.

The following needs to be remembered when creating literate programs:

1. Set the document class to "article (Noweb)" or "book (Noweb)" etc.

2. Normal text is typeset normally. Tables, equations, etc go wherever needed.

3. Add code fragments by starting a new paragraph (type *Enter*), and selecting the "Scrap" environment.

4. The form of the code lines should be as follows:

```
<<*>>=
code lines go here
@
```

i.e., the first line must contain "<<*>>=" and the last line must contain "@". Lines should be separated by "Ctrl-Enter".

5. The program can be in different blocks of code. When it is generated, these blocks are automatically combined to create the program.

6. Any name can go into the "<<...>>=". That name identifies both the piece of code and the descriptive text preceding it.

7. If you use various names, you should have a master section like this:

```
<<*>>=
  <<Params>>
  <<functions>>
  <<plotting>>
  <<solver>>
  <<fileio>>
  <<main>>
@
```

   This ensures that all the code you wrote got created. The purpose of having this option is that I could write several <<solver>> blocks and use the one I wish at the moment, by including it in here.

8. To view the document, select *view→PDF* or any other normal view.

9. To run the program, select *view→scilab* (assuming you are documenting *Scilab* code).

Look through the "Extended Features" section in Help, to learn more about this mode.

3 Scilab

Scilab is a programmable scratch pad for scientists and engineers. It has the following capabilities:

1. It can handle complex numbers and complex functions.

2. It supports vectors and matrices as native data structures. Single line instructions can implement matrix addition, multiplication, inversion and the solution of simultaneous equations.

3. It supports polynomials and rational functions as data types. This means that some kinds of Laplace Transform problems can be done symbolically.

4. It has a large number of functions to plot various kinds of graphs.

5. It has a large number of functions for performing user input/output.

6. It supports "lists" which are generalised arrays whose elements can be of different types.

7. It has a large number of utility functions to make most engineering calculations straightforward to implement.

8. It can call *Fortran* and *C* routines and use their results. Thus, one of *Scilab*'s common uses is as a "front end" for numerical codes.

9. It has extensive control system analysis capabilities.

10. It has a complete spectral analysis toolkit, which is useful in digital signal processing tasks. Additionally, it has tools to design analog filters as well.
11. It has various statistical tools to analyse distributions.
12. It can deal with graphs and nets.
13. It has a simulation toolbox to simulate complex systems.
14. It can read, analyse and play sound files.

4 The assignment

1. Go through the introduction section of Scilab and learn how to create vectors and how to plot them. This is mainly section 2.2 of *intro.pdf*. Skim through the demo session in chapter 1, but skip the advanced stuff for now.

2. We shall compute the error function.

   (a) Construct a vector `x` consisting of values 0, 0.3, ... , 3.0. Define a vector `y` defined by $y = \exp(-x^2)$. Plot `y` vs `x`, using `plot2d`.

   (b) Integrate using inttrap `y` to obtain `z`, defined by

   $$z = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

   You will have to use a for loop to get `z` for different values of `x`.

   (c) Directly compute `w` from `x` using the built in function `erf`.

   (d) Use `xset` (look it up in the help) to create a new plot window, so that your previous plot will be preserved. Plot both `z` and `w` vs `x` in the new window.

   (e) The integral was not very accurate. Find the *maximum deviation between* `w` *and* `z`.(Look up the `max` function). This is the error in your calculation.

3. Use the code in problem 2 to obtain the error for different `x`, corresponding to different steps, `h`, i.e., $x = 0, h, 2h, \ldots, 3.0$. Create two vectors, one with values of `h`, and the other with corresponding errors.

   Plot the error versus step size in a new window.

4. Add noise to `y` using the `rand` function:

   ```
   rand('normal');
   y=y+rand(y)*0.1;
   ```

   Plot the new function. Integrate it to get $erf(x)$. Does the integral change? By how much. Do you understand this result?

5. Create a single script that does problems 3 to 5, with a *pause* command between successive problems. Convert it into a *LyX* document with embedded code. **Export the graphics to eps files and import into *LyX***. Make it a good report.

6. Create a *LyX* file and write a literate program in *C* to implement the above problem. Compare the *C* code with the *Scilab* code. Observe how *Scilab* reduces a lot of the routine coding tasks in *C*.

Create an archive containing both the *LyX* files (use: *zip archname.zip filelist*) and submit the same to the assignment website.