Department of Mathematics and Computer Science
University of Southern Denmark, Odense

December 14, 2015
KSL

DM819 – Project Feedback

– on manuals, input formats, and program execution

Kim Skak Larsen
Fall 2015

Introduction

I did not make many explicit requirements in the project description; I simply assumed you would do something reasonable. Very few were close to doing it well; a lot were quite far from. Here are my opinions on how it should be done.

I've tested all approximately 15 programs. This will probably shock some of you, but every single little point I make below represents things I've seen in at least one program and usually in several!

Goal

Of course, we're all busy, so I want to be able to test your programs quickly on my own input. Additionally, I might want to test all your programs first in an unbiased manner before reading your reports. This is exactly the same situation as if you want to install one of a number of possible programs to solve a specific task. You want to quickly test them on your own input to see what works best, without reading a lot of documentation first. If you want to choose between 10 products, you would like to be able to test each of them using just a few minutes. This is quite possible, but the providers have to make logical choices to enable you to do that. Think about what you would want if you had to test 10 systems.

Report Organization

Make it easy to find the manual. You could put it first. You could also put it last and as your first page in your report after the title page place a table of contents where the word **MANUAL** appears so it's easy to spot.

Also make it easy to spot on the page where you place it. Don't hide it in the middle with no or an insignificant headline. Make a large boldface headline at the top of the page.

Place all relevant information together in the manual, i.e., how to run it and a description of the input format. Don't refer the reader to the appendix to find an example of the input format.

Input Format

How to describe it

Focus on clarity. Don't start with an inline description. Pull the format out, centered on the page, possibly in a typewriter font.

Be complete and precise in your description. Explain exactly what is allowed. For instance, if you state that a floating point number is allowed in a given position, clarify whether or not all integers are required to be followed by ".0". Explain exactly which type of white space is required/allowed.

After that, give a small example, listed as clearly as the description of the format.

How to define it

Be flexible where it's easy. There is no reason to require exactly one space or one tab between entries, when you could just as easily split your input line on all white-space. In particular, requiring a newline only between lines (in particular forbidding a newline as the last character in the file), is not helpful.

Also at the semantic level, you ought to be flexible. I can't imagine that it's the user who wants a system, where a line has to be specified with the points sorted on x-values, for instance.

Be succinct. An input file should not try to appear as mathematical notation. Thus, you should not have parenthesis around your points or use set notation for a pair of points representing a line. To represent a line segments, you need to specify four coordinates and that's all we need, other than some separator.

Use a standard format. Then one may not have to convert input files at all before running your program. And if one has to convert, it's probably easy. Tab-separated or comma-separated are obvious standard choices. Since you have to deliver something that runs on and will be tested on a Linux system, tab-separation is more in that tradition, but if it's comma-separated, one can convert in a few seconds.

Don't make it easy for yourself. You should make it easy for the user. It's probably not the user who has a great desire to write in the beginning of the file how many lines will follow. This is probably something you have decided to avoid a little bit of programming. Don't do that.

Output Format

Graphical flexibility

Be flexible! Don't decide that you're displaying a 100×100 coordinate system on the screen, independent of what the user inputs. The user wants to run your program on his own input. If that happens to be coordinates between zero and one, all the user will see is a mess in one corner. If the input data consists of coordinates in the range of millions, everything will be off screen. You should scale to accommodate the user.

If you're not flexible, then at least inform the user clearly in connection with specifying the input format that he has to use values in a certain range.

Graphical delivery

Make it easy for the user. It is not convenient that you write mixed information to std-out, some of which the user is told could function as input to R. It is also not convenient to receive an incomplete LaTeX document in the form of some tikz application. This was probably convenient for you, when you needed to include documentation into your report. However, the user initially just wants a pdf-file (png may be OK).

Execution

Compiling

Make it easy! If it's java, don't do anything such that more than `javac *.java` is required; similar for other programming languages. If you have compelling reasons for doing something more complicated, you should provide a script, and you should explain carefully why and what. The user doesn't feel like executing magical enchantments in his own file system without a careful explanation of what all programs and options do and mean.

And, of course, it should run on the user's system. In this particular case, this means that the final test must have been carried out on computers in IMADA's Computer Lab.

Input test data

Just to make it clear: of course, the user wants to test on his own data to see if things work. Unless the provider has done a terrible job, the program works on their own input data, so that's not what you want to test. Also, generating random input data is not a real test. Many algorithms perform fine with respect to both correctness and time complexity on uniformly distributed random data. The user has to be able to provide his own special case testing in order to be convinced.

Interaction vs. command-line

When implementing some system, the intended use may be interactive, but for testing, this is not convenient. For testing, we want self-contained test instances with documentable output.

Interacting with your program may be nice for an actual application one wants to use. Especially, if it's a nice graphical interface. However, for testing, answering questions such as which file name to use, how many lines there are, what the the query is, etc. is not a pleasant interaction. The same thing goes for requiring that input is stored on files with specific names, or overwriting the user's input file! All these things are quite time-consuming if one wants to run many tests.

Naturally, hard-coding information that ought to be specified in the input into the program is unacceptable.

Make it possible (and make it the standard option) to run your program from command-line, specifying one file; either as the one argument to your program or by reading from stdin. Thus, the input file must contain all information, including a possible query.

It's fine to provide options for your program so that it can be run in other modes as well.

Test documentation

Again, it can be pleasant to see a nice graphical, interactive implementation, but for testing, we want the following: We want a fast way to determine (possible) correctness in the form of graphical output. And we want to be able to get the precise values representing the result of a query for more detailed inspection. You could either deliver both immediately, or make the fast graphical inspection the default and require an option for the detailed textual representation.

If there is a problem with the program, it should be possible to report back that the program misbehaves on this particular instance. This is not possible if your implemention has interaction as the only possible input option. It should be easy to obtain an input file with the problematic input data and a pdf-file that documents the error.

In some cases, it might be nice to provide the user with additional test options, but this becomes quite application dependent. One example would be an algorithm that makes a random shuffle of input before executing. Here, it could be nice to provide the user with an option to turn off randomization for testing. You probably needed that feature yourself anyway when you tested your program.