Exercises: Instructions and Advice

Instructions

The exercises in this course are primarily practical programming tasks that are designed to help the student master the intellectual content of the subjects and develop facility with every-day numerical programming for solving science and engineering problems.

The choice of computer language or computational system used to perform these exercises is left up to the student. Many students will use MATLAB® or (preferably) its open source equivalent Octave. However, it will certainly be possible, and in some cases more elegant, to solve the exercises using one of many other different languages. That is fully acceptable, but be warned that low level programming languages generally do not provide the matrix algebra and plotting libraries that will be required. Also, it will not be possible to provide identification of code errors in arbitrary languages as part of the feedback.

A question naturally arises as to what "built in" facilities of a big system like MATLAB are allowed as part of the solution. For example, MATLAB has built in ordinary differential equation solvers. The overall guiding principle of this course is that no-one should need to program matrix algebra, decomposition, inversion, eigenvalue finding, and so on. They should simply invoke it from the system (or libraries). However, we are going to learn how to fit data, solve differential equations, and lots of other things; therefore, producing the solution of the exercises from a potted fitting or solving routine will miss the point. It might be useful to cross check your solution against a built-in one. But if you are thinking of using a built-in routine for numerical solution, ask yourself the question: is this a matrix algebra function (if so, then you are fine), and is this bypassing the point of the current topic by using a potted routine (if so, then you are defeating the object).

Many exercises require the submission of your code. It should normally be submitted as a text file with the appropriate extension, capable of being run by the mathematical system or compiled by the compiler corresponding to what you have used.

When you are instructed to submit your output data, for example the fitting coefficients, or the solution vector. It ought to be submitted in a text (ascii) format file. Binary output is not portable. With luck MATLAB binary output from a command like `save -mat variable.mat var1 var2` will prove to be usable, but it's hard to guarantee. Therefore, always submit a text file format as well (e.g. one generated from the command `save -ascii variable.asc var1`).

Wordprocessing program formats are also not portable. Please avoid them except for providing your textual answers, and preferably save them to pdf and submit the pdf, which is far more portable.

Advice

Programming is a skill, and practically an art. Most science and engineering students need to become competent at it because data analysis and theoretical evaluation are done almost always these days numerically and computationally. Competence at some level is almost as necessary as being able to speak your native language. A better analogy for non-native

English speakers is that it is like English competence (which is practically vital for all high level scientists and engineers today). However, even if everyone needs to be able to speak and read, not everyone needs to be a Shakespeare.

This course is *not* intended to teach you programming. It assumes you have some level of competence, and it will help you to improve that competence. Here are a few points of advice that apply to almost any numerical programming task and any language, that may help you to be more productive in the exercises.

Scripting and saving

Many mathematical systems allow you to type commands interactively and see the results immediately. This is very useful. However, it is a mistake to *over-use* this facility. It is generally much more useful to have a file that contains the commands required for performing a particular computational task than it is to have the computational system in the state that would be arrived at by those commands. The reason is obvious; the file of commands enables you easily to reproduce the state at will, the reverse is not true (not even if there is a logging facility that is capturing your keystrokes!). Therefore it is almost always good policy to switch from interactive computation to editing a program and then running it, earlier in the development cycle rather than later.

Integrated Development Environments

It's very helpful to use an editor that is aware of the language you are writing in. It is then able to do elementary analysis of the program as you type, hinting when the syntax is incomplete or incorrect, indenting the format of the program in a suggestive way, and so on. MATLAB itself provides such an editor, built into its integrated development environment (IDE). It can be used to great effect.

Octave is more of a manual, text mode, version of MATLAB, but it also has a language aware editor. Actually it has no editor, but will by default start up the editor `emacs` if you give, for example, the command `edit file.m`. Emacs (on a properly configured Linux system) will recognize that a file ending in the extension `.m` is an Octave (MATLAB) file and will when visiting that file turn on the appropriate emacs mode. The same is true for other languages and extensions like `.c`, `.f` and so on. Being able to use a powerful editor like emacs is a useful skill. What it gives is an editor whose basic commands are the same, but which adapts to the language being written. It is in a certain sense a "poor man's" IDE, but sometimes poorer is richer. You don't have to learn a new editor for each different language or system.

Data distribution

Some exercises provide you data. Each student gets different data! It is distributed in the form of two files. One is ascii (text). It can be read in by Octave (but not MATLAB, at least not without tweaking). The other (ending in an extension `.mat`) is binary but ought to be readable by both MATLAB and Octave. To read the data in a file `filename.mat` into MATLAB or Octave, simply save the file to the disk, and issue (or program) the command `load filename.mat`. It will create the variable with name specified in the saved data file

containing the data. You should try it out and then examine the data to ensure it's being read in properly.

If you are using some other system than MATLAB or Octave, you'll have a bit more programming to get the data read in. Use the ascii file rather than the binary file, and notice that there are comments inside it that describe the variables. Either edit the file into a form that your system understands, or else program your system to interpret the file automatically (way too ambitious, good luck!).

Parameters

Most programs have parameters that are fixed for certain types of problem and are used to describe other variables. For example, a vector or data set might have a length N that is known ahead of time. Suppose we have a problem where we know the length of the vectors is 6. You will be tempted to write commands like `b=zeros(6,1)` which in Octave generates a column vector of length 6 containing all zeros. You should almost always *resist the temptation to program values explicitly*. Instead you should define a parameter, e.g. `Nlen=6` (one command), near the beginning of your code and thereafter write `b=zeros(Nlen,1)`. The reason is simple. As the program is developed, it is most likely that there will be a steadily proliferating set of occasions where the length of the vectors is used. If you just write 6 for each of them, your code becomes increasingly difficult to change, when, sometime down the road, you decide you want to reuse it (or parts of it) for some other problem where the vector length is different. If you've systematically used a parameter `Nlen`. It is, by contrast, totally trivial to change that parameter in one statement and hence achieve reusable code.

Some of the exercises build up more complicated code through simpler stages and problems. If you use parameters liberally, you will find this development easy and natural. If you program values explicitly, you will struggle.

Incremental, Small-scale, Testing and Debugging

Another vital use for parameters is to enable incremental testing. Suppose you are solving a differential equation and need to evaluate the result at 1000 points across a range from 0 to 1. You program your code and yet things don't seem to be working correctly. What do you do?

The most important thing to remember about debugging is this. *You need to look at the data*. Yes, it is important to look at the program. But it is equally important during debugging to look at the data. Sometimes plotting it graphically is helpful, but sometimes not. But how do you look at 1000 numbers printed out? You can't usefully do so. It might help to look at a small sample of the whole 1000 data points. But that might not tell you where the error is. It is often far better to look at the whole of the data *for a smaller number of points*. If everything in your code that depends upon the number of points has been programmed using a parameter, then you can debug your code by putting the parameter to a small number (e.g. 10) and printing out the values for all relevant indices. Another big advantage of doing the testing for a data length of 10 rather than 1000 is that your code will probably run far faster. That makes it painless to run the code over and over again adjusting things and examining things till you diagnose the problem.

Enabling small-scale testing requires using parameters and avoiding explicit programming. For example, to construct an array of 1000 x values that goes from 0 to 1 you don't write `x=[0:999]/999`, even though that is perfectly correct. You write `x=[0:Nlen-1]/(Nlen-1)`. And this sort of remark applies to everywhere in your code. Then for small-scale testing you can trivially adjust `Nlen`, with the ability to put it back to the required value when the bugs have been identified and fixed.

Another aspect of debugging is that it works best if done incrementally. This means you develop a smaller section of code, and by putting in data printing or display statements as you go, you debug the section of code and develop confidence that it is doing something sensible. You can always comment out the data displaying statements later. Of course displaying the data as you go will likely be too painful if you start with a giant data length. Instead you do it on a small scale. When each section of the code is working on small-scale data, you incorporate it into the whole and move on to the next stage.

The world's most effective programmers do *not* write code that works right first time. (Nobody writes code that works right first time.) Instead they write the tests for the code either before or during the coding. That way, they know their code works, not because they think they planned it and typed it in correctly, but because it passes the tests. This may sound cumbersome, but it is an approach that time after time has been shown to produce results fast.

22.15 Essential Numerical Methods

Fall 2014