PIASD Programs for the Interactive Analysis of Strainmeter Data

Duncan Carr Agnew

IGPP
La Jolla, CA

1. Introduction

PIASD is a set of programs designed for the analysis of time series, with a particular focus on some of the processing tasks that are commonly applied to data from strainmeters and tiltmeters: tidal analysis, interactive editing, and spectrum computation. This system, like UNIX, uses a number of programs, each with a specific function; it delibrately avoids having a few programs (or just one) with many options. This puts more of an initial burden on the user, since there is perhaps more to learn; the payoff is the much greater flexibility from being able to put the programs together in different ways—and especially in being able to create scripts that use the programs.

A difference between this package and UNIX is that data is not piped between programs but instead is written to and read from disk files. The programs get needed parameters either interactively or from a script, usually in question/answer form, occasionally through command-line options. The system thus relies on the user to pass information between programs; so it is always helpful, and usually necessary, to keep some record of what is done. As described below, the best way to do this is to package all operations in scripts, which are easily made self-documenting.

2. Program Behavior

Many of these programs were written (as was UNIX) some time ago; and so some aspects of them may now seem somewhat old-fashioned. This section describes the two most obvious of these aspects: how the programs interact with the user, and how data files are named and structured. It is not just backwards compatibility that has led to the retention of the methods used: they also have significant advantages for use in a scientific setting.

2.1. Interaction with the User

The PIASD programs, like UNIX, (and such programs as GMT) are text-based—the interactive editing program *credit* aside. However, instead of using command-line arguments, the PIASD programs get the parameters they need by asking questions. This has the great advantage that the user does not need to memorize (as with GMT) a host of command-line options; a vague memory that (for example) *power* computes power spectra is enough to get started. After you start the program it will ask for what information is needed, as in this example:

```
%power
 Number of data file; 100
 Locations of first and last points ; 1 1000
 Length of sections (le 20000) ; 200
 Remove mean and trend from each section? (y/n); y
 Type of windowing (-1 for more information); -1
 Available windows are
   0 - none
   1 - triangular taper
   2 - 1 + cos (hann) taper
   3 - 4pi prolate spheroidal taper
   4 - triangular, 50% overlap
   5 - hann, 50% overlap
   6 - 4pi prolate, 70% overlap
 Type of windowing (-1 for more information); 5
 Number of output file; -100
  28. degrees of freedom.
  95% confidence limits are  2.6 and -2.0 db.
     51 points in file    -101
%
```

In this example and others `this font` is what is printed by the program; **`this font`** is what is typed by the user. This example illustrates the style used in PIASD, including the use of help lists when is appropriate. (It must be said that the flag for these lists is not consistent). Of course, you need to have some idea of what the program is doing: in the case of *power*, computing the spectrum by dividing the data series into sections, windowing each section and taking the Fourier transform, and then averaging the results. Obviously, knowledge of this method of spectrum analysis is needed to make full use of all the options; just for reference, tapering option **5** is usually a good choice.

2.1.1. Using Scripts

Besides ease of use, the biggest advantage of this style is that it makes it easy to create well-documented scripts which can combine a whole series of steps. Indeed, this is probably the biggest advantage: Given how often things have to be done over, having a script leads to considerable long-run efficiency. While setting up a script for the first time takes a little longer, all reruns are fast. Creating a script also documents what was done, in a way impossible with a GUI, and does so almost automatically (of course, adding comments in the script is helpful, and not difficult.) This turns out to be a huge advantage when (not untypically) some piece of analysis needs to be revisited, or reused and modified, some time after it was first done: you do not need to remember your steps, the script has them all recorded.

To create a script, you first run the Unix *script* command, so that everything typed is then logged in the file *typescript*. Once you have run the program, type **`exit`** to stop the logging. Then the program *rescript* is run to create a script that can be executed to rerun the program. For example, typing

```
%rescript typescript % > power.script
```

produces a file (*power.script*) that looks like

```
power << XXX
100         # Number of data file
1 -1            # Locations of first and last points
200         # Length of sections (le 20000)
y           # Remove mean and trend from each section? (y/n)
5           # Type of windowing (-1 for more information)
-100            # Number of output file
XXX
#
# -------------------------end of power
#
```

This file uses the UNIX ''here document'' syntax available in most UNIX shells to input the information to the program. The questions from the program are turned into comments: anything after the # is ignored by the routine that reads input for the program. *Rescript* separates questions from answers using the semicolon that ends all questions.

The script above differs (intentionally) from the first example in two ways. One is trivial: the **-1** answer to the `Type of windowing` question has been omitted, since there is no reason for the script to print out the list of options. The other change is that the terms to be taken from the file have been changed to **1 -1**. This is a common usage for most of the PIASD programs, and means ''take the second term, to be the last one of the file''. This usage makes the script independent of the actual length of the data file.

This example is deliberately limited to one program, with no variation. But more flexibility is easy. To begin with, it is not difficult to have the script include a whole chain of steps. If, for example, we wanted to create a digital filter, use it to filter data, and then find the spectrum, we could run *script*, then run the programs *filtds* (filter design) *deki* (filtering) and *power*, and finally **exit**. Then typing

```
%rescript typescript % > comb.script
```

will produce a script that, when executed, will run all all three steps. (Alternatively, we can go through the sequence `script`, run program, `exit`, `rescript` three times, adding each section to the overall script.)

The other source of flexibility comes from the use of shell variables in the scripts. For example, if we edit *power.script* by substituting `$1` for `100`, `$2` for `-100`, and `$3` for `200` (the section length) we have a general-purpose shell script which can be run as

```
%power.script file-in file-out section-length
```

which means that we can have a compact, command-line form of the program if we like.

Some of the PIASD programs do use command-line arguments; these will print out a short help message if none are provided.

As noted above, the answers to most questions (anything numeric) are read by a special routine (*ffin*) which will read any numerical format, with spaces, commas, tabs or carriage return being the allowed separators. This routine differs from F77 free format in that two or more adjacent commas are the same as one, rather than indicating a value of zero, and carriage return does not end the input. Also, the input routine will not let the program proceed until it has been given all its numbers; if for example, the program types, `"For ... type 1; "`, a carriage return will leave it waiting; you must type 1 (or 0) to proceed. While *ffin* does not in general take non-numeric input, there is one exception: the first non-numeric character is returned to the program. This feature is used in some programs.

2.2. Data Files: Names and Formats

As the examples above show, data files are referred to inside the programs using numbers: the data file is 100, and the output file -100. While the use of numbers is less flexible and less less mnemonic than names, it does, as I discuss below, have some advantages. But to start, the characteristics of data files are:

1. Data files consist of series of numbers (called terms, points, or series values) the first of which is term 1. Files may not exceed $2 sup 32$ terms in length: in practice, not much of a restriction (for short-

integer data, this would be a 64 Gb file). One way to think of such files is as a very long 1-dimensional vector; another is as a continuous (gap free) time series. There is **no** header or internal information on such issues as timing: for all the programs know, a series could be sampled once a decade or once every microsecond—or even just be a set of numbers, and not a time series at all. There are three types of files: short-integer (16 bits per term), long integer (32 bits) and floating point, or real (32 bits).

2. Files are referred in the programs by numbers. Ther actual filenames use the numbers, plus an appended .D, with prefixes as follows:
   a. Files containing short (16-bit) integers have a name starting with I, and have number values between 100 and 999 (which would be named `I100.D` and `I999.D` respectively.
   b. Files read as reals (32-bit) have a name starting with R, and have number values between -100 and -999 (which would be named `R100.D` and `R999.D` respectively.
   c. Files read as long (32-bit) integers have a name starting with L, and have number values between 1100 and 1999 (which would be named `L100.D` and `L999.D` respectively.
3. Files may be created in the system `/tmp` area. For such ‘‘temporary files’’, the numbering is 2100 to 2999 for short-integer files, -2100 to -2999 for real files, and 3100 to 3999 for long-integer files. The actual names include the pathname of whatever directory you are in, to avoid conflicts. For additional information see the man page for *niolib*, the programs that actually perform input and output.
4. Files in other directories than the one you are in can be accessed by adding a multiple of 10000 to the number (or of −10000 for real files). If this is done, the path of the relevant directory will be asked for. So, for example, if you wanted to combine data from three directories, you would use numbers in the 10000’s for the first directory, the 20000’s for the second, and the 30000’s for the third. **Note** that the directory name will need to be included in any script.

Since the files themselves have no internal information or metadata, additional files need to be used for this. One particular format that many of the programs interact with is called the *header file*; this is summarizes information about a single file (or set of files) with the same timing (start, end, and sample interval). Header files are stored as if they were short-integer data files (so, numbered 100 through 999), but read using a special subroutine. See the program *header* for more information. A header file can usually be recognized by its length, which is 586 terms (1172 bytes).

Naming files with numbers, while it makes the file names short, certainly can also make the names somewhat cryptic. At the same time, being able to order and group numbers in different ways makes it easy to come up with conventional arrangements of files. Once these are used a few times, they become easier to remember than names. Here is an example, for the common situation in which there is a number of channels of raw data, each of which has associated with it a file of editing instructions (how these work is described in the next section), and a final ‘‘cleaned’’ file. A README for this set of files would look like

```
 Ch     Name      Data       Edits  Cleaned
-----------------------------------------
                  100h      200h
 1     LSM HDR     101       201      301
 2     LSM RCLP    102       202      302
    pred tides  -2102
      residual   2102        "      -302
 3     N LOA       103       203      303
 4     S LOA       104       204      304
 5     VAC         105       205      305
```

where the third column (`Data`) gives the file numbers for the raw data, and associated files: ‘‘the 100’s’’. The fourth column (the 200’s) is the edits files, and the last column is the cleaned data. This example also shows a common use for temporary files. The predicted tides corresponding to the strain data in `LSM RCLP` are in a temporary file, outside the directory, since these can be regenerated at any later time. The

data are all in 16-bit integer, but the preedicted tides are reals, as is the cleaned residual series (to avoid roundoff). Of course, it would be possible to develop names for all these files, but when this was tried (and it was), it turned out to be difficult to develop a consistent set of naming conventions.

Several strategies for working with this type of file naming are:

1. Maintain some documentation (a README file) in whatever directory you are working. The scripts *lsd* and *lsdt* are designed to help in this by listing the files as they are referred to, giving their series lengths rather than file lengths.
2. Use temporary files as much as possible for intermediate stages—this minimizes the number of files in a directory. If you develop the proper scripts, you could just keep the raw and edits files, regenerating the final cleaned data as temporary files when needed.
3. Develop consistent habits, of the type shown above, where certain ranges of numbers, and offsets between ranges, have certain uses. Another convention we have, besides that shown above, is to retain the highest range (990 to 999) for throwaway files that we do not want to have vanish (as the temporary files are set up to do).

3. Editing Data

The basic philosopy adopted in PIASD for data editing is to make it interactive, since only a few kinds of simple problems can be handled by automatic editing programs: notably steps in otherwise good data, and single-point spikes. Anything more complicated calls for the judgement of an expert, which has a flexibility that can be programmed only with much difficulty. The PIASD editing programs aim to make it as easy as possible to apply expert judgement, while minimizing the drudgery of keeping track of the editing information.

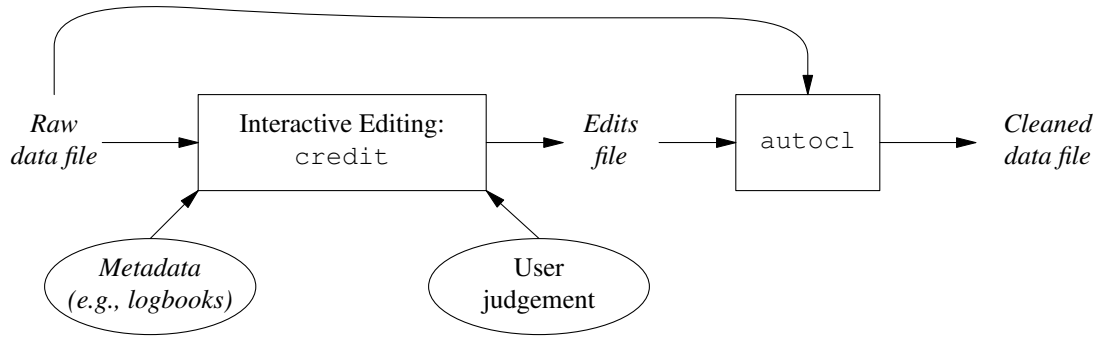

Figure 1

Figure 1 is a block diagram that shows how this is done. The most important part is the program called *credit* (for ‘‘cursor edit’’). This takes as input a data file (assumed to be a time series), displays it on the screen, and allows the user to indicate where edits should be made, both to discard data and, if necessary, to offset the series across such a gap. As edits are made, they are applied to the display, so the series appears to be edited. In fact, the original data are not changed at all: each time data are displayed, the raw data are read in from disk and the editing instructions applied to make the series plotted. The output of *credit* is an **edits file**, which contains the editing information only. This file is an *niolib* integer file; though it has a special format (described below) the user does not need to deal with this.

Once an edits file has been created, the program *autocl* (‘‘auto-clean’’) is used to read in the edits and the raw data, and write out a file of ‘‘cleaned’’ data, with the edits applied. The editing information is thus kept separate from the data; there is, intentionally, no ‘‘flag value’’ to indicate where data are bad.[1] Separating the edits from the data has turned out to be extraordinarily useful, since then the editing information can be used in a variety of ways. For example, in performing a tidal analysis, the program for fitting harmonics reads an edits file to find out where to assume there are ‘‘gaps’’ in the data: that is, terms to be skipped in the analysis.

[1] Though *autocl* can create a file with this, and *gsedit* read such a flagged file and produce an edits file.

For historical reason (the need to deal with long series while using only short integers), the edits file format is packed unusually. The first term is the number of edits; each edit is specified by a triplet of integers. Each triplet gives (in order) the number of terms from the preceding edit (for the first edit, the origin) to the last good point preceding the bad section, the offset to be applied to the data folowing the edit relative to that before, and the number of bad points + 1. An edit file containing the numbers 2 521 −211 19 2763 500 67 thus has 2 edits, with terms 523-540 and 3305-3370 being bad; 540 is 521+19+1, and 3305 is 540+2763+2. Also, a total offset of −211 would be applied to terms 541-3304, and 289 (= 500−211) to terms 3371 onwards. If the edits are stored as short-integer files, the maximum gap length is 32767 terms, and the largest offset is ±32767 units. Both restrictions can be overcome if the edits file is long integer; note, however, that the restriction to integer means that offsets less than ±1 cannot be done. This is rarely a problem with actual unscaled data.

4. Plotting Data

Aside from the screen plots made within *credit* and *datred*, this package does not include any software for plotting data, on the assumption that any user is likely to already have a favorite one. I use, almost exclusively, the *plotxy* program developed by R. L. Parker and Loren Shure, since this allows complete control of the appearance of the plot, controlled by a fairly set of commands. This program is descibed at http://mahi.ucsd.edu/parker/Software/software.html. A version is available that reads *niolib* files directly. Otherwise, to convert such files to ASCII that most plot packages will read, see *putdat*.

5. Additional Documentation

All of the programs are, as noted above, self-documenting, usually through the questions they ask; those that use command-line arguments will print a short description of usage if they are run with no arguments. Fuller descriptions are contained in the *man* pages, which also cover some of the subroutines they call: notably *header(5)* which describes the contents of the header files, and *niolib(3)*, which gives more details about the data files. There is an index and permuted index to these pages. For the interactive-editing program *credit* there is also a tutorial.

6. Program History

The (distant) ancestor of this set of programs is a package called BOMM, developed at La Jolla in the 1960's by Sir Edward Bullard, Flicki Oglebay, Walter Munk and Gaylord Miller. This was a command-line system (albeit with the commands punched on cards), whose spirit (somewhat) names (more often) and code (occasionally) lives on in these programs. The continuity is perhaps unsurprising, since BOMM was developed for processing of long time series, particularly of sea-level and weather data: there are many similarities with strain data. The conversion to an interactive system took place in the late 1970's, on a PDP-11 system: for what it is worth, the same machine being used at the same time for the early versions of UNIX. This development was done for processing low-frequency strain and seismic data, and was undertaken by a number of people including Duncan Agnew, Jon Berger, Bob Parker, and Karen Young. The initial conversion to a UNIX-based version was done by Agnew at CIRES in 1980, and developments have continued to the present. A distribution was made available in 1981, under the name PITSA; unfortunately, this acronym was also used by Scherbaum and Johnson, a little later, for a suite of seimic data analysis programs, and so has had to be dropped. PITSA/PIASD is now on its sixth operating system.

REFERENCE

E. C. Bullard, F. Oglebay, W. H. Munk and G. R. Miller (1964). *A Users Guide to BOMM: A System of Programs for the Analysis of Time Series*. (La Jolla: IGPP).