# Correlating HiSPARC Data

Michael McEllin

10 June 2019

## Introduction

This note explains how one goes about creating programs to handle scientific data. It is based on over 35 years of experience in this field, during which I have constructed extremely complex software to deal with very extensive amounts of data.

Given that my work involved the safe operation of nuclear reactors, you should not be surprised to learn that *avoiding mistakes* was exceptionally important. It took priority over getting things done quickly, but, perhaps surprisingly, much of my accumulated experience of writing software suggests that if takes particular care to avoid errors, *things do happen more quickly anyway*. Finding and correcting mistakes does more than anything else to slow progress and make delivery targets uncertain. I discovered that treating even non-critical software as if lives depended on getting it right meant that I delivered stuff faster than those who believed that they could save time by taking short-cuts.

Making mistakes during the construction of software happen because nearly all genuinely useful software is intrinsically complicated. They have an irreducible level of complexity - something that is essential to the nature of the problem and which we cannot simplify any further by clever design. We can, however, make it more complicated than they need to be by poor design. Mistakes in software happen because we do not really understand what we are doing. (People are surprisingly good at fooling themselves into thinking they understand the effects of the code that they author - and even more surprised when it starts to behave in unexpected ways.)

The central aim of writing good software is therefore to make the software design as simple as it can possibly be, and to make the operation of the program as transparent as possible to anyone who needs to look at the code.

In my experience, and it is considerable, most of the mistakes that I discover in my own code are eliminated before the first test run of the software. I continually read my code and explain it to myself, and very often find that I cannot do that. If I cannot explain it, I do not understand it and it is therefore very likely to contain errors. (An American software developer working in the laid-back West Coast environment, claimed that he used to explain stuff to a programming buddy but when the buddy left he took his dog into work and explained stuff to the dog. This turned out to be just as effective at finding errors. When the dog died, he explained stuff to a photograph of the dog. This turned out to be just as effective at finding errors.)

## Understand the problem

*If you do not understand which problem you are trying to solve it is very unlikely that you will get the correct answer.* (Advice applies also to dealing with exam questions.)

It is quite astonishing how many professional software developers ignore this advice. Next time you hear about a Government computerisation project being billions of pounds over budget - or failing completely (they come along about once a year) this is the most likely cause. There is some excuse when you are dealing with governments, because their problems are very complex, and by the time you start understanding what they want, they have probably changed their mind.

There is no excuse for smaller-scale problems, but (in my experience) 90% of scientific programmers start writing their program before they have written down a clear statement of what

they need to do. This need not be a big deal: in small programs I often write it into the comment lines at the start of the program.

Let us suppose need to use HiSPARC data downloaded from their web interface in order to answer the specific question of whether there is any correlation between cosmic ray event rate and atmospheric pressure and/or temperature - and perhaps other parameters. This is a fairly vague target and we need to be a bit more specific.

My first look at the HiSPARC data lets me see several of things immediately:

- The data I think we need arrives on my computer as two different files. One contains weather data logged every five seconds; the other contains information about cosmic ray events which occur on average every few seconds, but at irregular intervals.
- In both cases the files are "ASCII" format, which means that they can be interpreted by computers as strings of characters, and in this case divided into multiple lines by special "line-end" non-printing characters.
- In both cases the files contain data arranged in columns, including information such as the time at which the data was recorded and for example, in the case of the weather file, columns such as barometric pressure, and outside temperature (plus other stuff in which we may not be interested). The columns are in "tab-separated" format, which means that the characters in column are divided from adjacent columns by the special ASCII tab-character. The file looks a bit like spreadsheet data, and in fact it is easy to load this data into Microsoft Excel: it has a data import tool that will drop tab-separated columns of data into separate spreadsheet columns.
- I also note that I can find out what the data in each column is supposed to be representing by looking at the comment lines inside the file. (In particular, it is very clear about the physical units.)
- Although two of the columns contain date and time in a conventional format, both files also have a column labelled "timestamp " which is described as "Unix timestamp[1]". This is a widespread convention for labelling scientific data with a time. It means the number of seconds since that start of 1970, which is the way the internal clock works on Unix and Linux computers (which are the most widely used by scientists for data analysis). This will prove to be the most convenient way to see whether our data refers to the same time period.

The text box on the next page shows an example of the weather data.

In this form, I cannot easily correlated data in the cosmic ray event file with the data in the weather file, firstly, because the data is recorded at different times, in one case at regular intervals in the second at irregular intervals. Secondly, because the cosmic ray events are recorded individually and we want to correlate with the average rate at which events occur over specific time intervals.

I am going to need to rearrange the data so that I know that average rate of cosmic ray events at some particular time when I have measured weather data. In fact, because the weather does not change all that rapidly, I do not need to data every five seconds and I can probably replace a very large number of readings with a much smaller number containing averaged weather data. (Furthermore, since we only get two or three cosmic ray events in each five second period averaging over that interval is probably meaningless.) My first thoughts are that averaging weather data over an hour is probably good enough and counting the number of events in each hour is probably also sensible. We might, however, want to keep this averaging interval flexible.

---

[1] The HiSPARC website sometimes refers to this as GPS timestamp, because it is also the time value read from the GPS signal on each HiSPARC detector array. For really accurate work you need to be a little bit careful, because you might come across data from other sources (which you may wish to correlate to HiSPARC) that is timestamped from "Universal Time" (UTC). GPS time does not take account of the leap seconds which are occasionally added to UTC in order to take account of the slowing down of the Earth. The equivalent UTC timestamp is currently about 8 seconds adrift from the GPS time.

**Text Box 1: Example of Weather Data**

```
# Event Summary Data - Weather
#
# Station: (501) Nikhef
#
# Data taken from 2019-05-21 00:00:00 to 2019-05-22 00:00:00
#
# HiSPARC data is licensed under Creative Commons Attribution-ShareAlike 4.0.
#
#
# This data contains the following columns:
#
# date:               time of event [GPS calendar date]
# time:               time of event [GPS time of day]
# timestamp:          time of event [UNIX timestamp]
# temperature_inside:   temperature inside [deg C]
# temperature_outside:  temperature outside [deg C]
# humidity_inside:      relative humidity of the air inside [%]
# humidity_outside:     relative humidity of the air [%]
# atmospheric_pressure: barometer data [hPa]
# wind_direction:       wind direction [deg]
# wind_speed:           wind speed [m/s]
# solar_radiation:      intensity of solar radiation [W/m/m]
# uv_index:           measurement for UV intensity [0-16]
# evapotranspiration:   amount of evaporated and transpirated water [mm]
# rain_rate:          amount of rainfall [mm/h]
# heat_index:           perceived temperature taking humidity into account [deg C]
# dew_point:            water vapor below this temperature will start to condensate [deg C]
# wind_chill:         perceived temperature taking wind into account [deg C]
#
# Values of -1 or -999 indicate a problem in that measurement or
# the absence of that sensor.
#
#
2019-05-21  00:00:01 1558396801 23.8 11.7 45  94  1009.6  224 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:06 1558396806 23.8 11.7 45  94  1009.6  223 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:11 1558396811 23.8 11.7 45  94  1009.6  240 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:16 1558396816 23.8 11.7 45  94  1009.6  209 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:21 1558396821 23.8 11.7 45  94  1009.6  244 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:26 1558396826 23.8 11.7 45  94  1009.6  242 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:31 1558396831 23.8 11.7 45  94  1009.6  205 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:36 1558396836 23.8 11.7 45  94  1009.6  238 0  0  0  -999  0  11  10.7  11.7
2019-05-21  00:00:41 1558396841 23.8 11.7 45  94  1009.6  213 0  0  0  -999  0  11  10.7  11.7
```

The crucial point here is that I must be able to process the input data so that whenever I have a weather observation, I also have a corresponding cosmic ray event rate estimate that applies at the same time.

I could choose to write the weather data and CR data (now on the same time grid) to two different output files, matching the input files for maximum flexibility in further data handling. However, given that we are dealing with only a few types of data (pressure, temperature, CR event rate) and we have at least some idea that we will, for example, probably be doing things like plotting pressure against CR event rate, it now seems sensible to write it out to a single file, again arranged in columns, but this time containing both the weather data AND the cosmic ray event rate. I know that having done this I would be able use this file as a feed to other programs, such as graph plotting software or statistical analysis software.

| year | month | day | hour | timestamp | temperature | pressure | eventRate |
|---|---|---|---|---|---|---|---|
| 2019 | 5 | 1 | 1 | 1556672400 | 9.29902234637 | 1018.70921788 | 2314.0 |
| 2019 | 5 | 1 | 2 | 1556676000 | 8.96456582633 | 1018.13123249 | 2372.0 |
| 2019 | 5 | 1 | 3 | 1556679600 | 8.83795518207 | 1017.72745098 | 2305.0 |
| 2019 | 5 | 1 | 4 | 1556683200 | 8.87380952381 | 1017.49229692 | 2369.0 |
| 2019 | 5 | 1 | 5 | 1556686800 | 8.67146853147 | 1017.3434965 | 2373.0 |
| 2019 | 5 | 1 | 6 | 1556690400 | 8.71358543417 | 1017.35966387 | 2324.0 |
| 2019 | 5 | 1 | 7 | 1556694000 | 8.93571428571 | 1017.29397759 | 2385.0 |

I have already achieved something significant: we now know enough to get moving on one half of the project. When we have finished that, we will have our data in a format that gives us a good deal of flexibility in choosing different ways to analyse the information. Furthermore, I happen to know (because of a lot of previous experience) that we will be able to choose between a number of graphing and statistical analysis programs that readily read data looking like this. We can decide exactly what analysis we do at a later time. This is a good thing! In scientific investigations our first examination of the data often suggests further questions that need to be addressed, so we really do want to make the second part of the work as flexible as possible, because we probably do not yet know everything we might want to do.

So my statement of the problem is a description of the data that I am to be given, and a description of the output file in fairly general terms, by which I mean that I know what types of data it must certainly contain, but I have not yet decided the exact order of the columns or indeed the entire set of columns. (I might find it convenient, for various reasons, to include more data that the absolute minimum requirement.)

It is very important to identify all the assumptions that you are making about the way data is presented to you, and the way you expect to hand it on. You may think that some of these are so obvious that they do not need to be stated explicitly - but this is not true. If you study computer science at university you would taught to think about these issues in terms of *pre-conditions* and *post-conditions*:

- A **pre-condition** is something that has to be true in order for your program to work correctly. For example, in our case it is a precondition that the third column in the data file represents a timestamp, the sixth a pressure, and the seventh a temperature. A rather less obvious pre-condition (perhaps) is that the weather data and the cosmic ray event data must cover the same time period. Just thinking about things like this immediately raises questions: what if the coverage does not quite align? Are we going to provide algorithms to just look at periods where there is overlap? Or are we just going to say: up to the user to get this right - the program will fail to give output if you provide input data that does not meet the pre-condition.
- A **post-condition** is something that will be true after the program has executed, and in our case will often be part of the description of the output data.

The nice thing about using pre and post conditions is that if you are chaining programs together (output from one acts as input to another, which is very common) we can easily see if the post-conditions of the first in line imply the pre-conditions of the next in line (or not!). I can assure you, based on my experience, that it is often the things people think are so obvious they do not need to be recorded that then get forgotten and subsequently cause serious problems.[2]

*Write all this down!* It is very easy, particularly when two or more people are working on the same problem to assume that you remember it all and have told everyone else about the important details.

# Decide How to Solve the Problem

I now know what data handling I must do, but not necessarily exactly how I will do it. I therefore ask myself a series of questions:

*Question 1*: *Do I actually need to write a new program?* If I were working in an industrial environment (or perhaps at a university or CERN) the answer might well be “No!”. This particular

---

[2] I once tracked down a serious failure of an important calculation route on a nuclear power station to an apparently innocuous change to a program: adding a space at the beginning of each line in an output file (containing data arranged in columns) which occurred unnoticed after an update to the computer operating system. This detail was not part of any description of the output file, and in any case it would have been assumed to be unimportant because the file was originally intended to be interpreted by human eyeballs. Unfortunately, years later, someone (without telling anyone else) though it was a good idea to read this data into another program, which assumed that the data columns were separated by spaces, so it now interpreted the first column of data (after the initial space) as the second column and so on. The smallest details may matter!

type of data handling problem pops up so frequently that there are commercial tools and sometimes open source software that can do much of what we need with only a little effort required to construct a controlling script. Writing your own software just creates more chances to make mistakes. We only do it when we need to do something that cannot be done in other ways. This does, however, occur fairly often.

*Question 2: How do I decompose the problem into simpler parts?*

Divide and rule is always good advice. We know that there are two different types of task we will need to do in order to make sense of the information:

1. Read the data in the format given by HiSPARC (this is something we do not control: it is a given).
2. Analyse the data for correlations between different physical parameters.

We now have to decide on the exact format of the data that joins the two parts of this work.

There will also be further decomposition of Part 1 (see below) into the data reading, averaging and output tasks.

*Question 3: What is the computing "ecosystem" in which I have to work?*

I probably want to pass my half-processed data onto other programs for further analysis (say, graph plotting or statistics). That means I probably want to write the output in a format that would be understood by the next program in the chain. I may also find that I have computers that support some programming languages, but not others, or that the people who need to adapt the software know some languages already, and are not prepared to learn new languages.

*Question 4: Which computer language do I choose?*

I have in the past written data processing programs in about ten or eleven different languages (that I can remember) not to mention various Linux scripting/pipelining tools: there is no perfect language suitable for every occasion, and most have advantages and disadvantages even for particular situations. Languages become popular and then drop out of use for no particularly good reasons.

I chose to use Python in this case for a number of reasons:

- It is very widely used by scientists and engineers. If you think you are going along this educational track, you *will* need to learn it at some point. (Most university engineering/maths/physics courses will include programming modules, often using Python.)
- Python has some built in features that make it particularly useful for reading data from one source and passing it to another program. (Scientists often need to read data from experimental apparatus and prepare it for further analysis.) For example, reading ASCII text files in which data is arranged by columns is particularly easy.
- There are extensive libraries of software written to deal with data analysis for many types of scientific problems. We can potentially save ourselves a lot of time using that stuff rather than writing it ourselves. (Caveat: school installations of Python sometimes do not include the really useful data handling libraries. Amongst these, there is, in fact, an extensive module specifically dedicated to handling HiSPARC data.)
- Schools think Python is a good teaching language. I agree. You can quickly get further with less effort than in most other languages, and in general the programs are easier for people reading the code to understand. Students tend, on average, to make fewer mistakes, and mistakes are easier for tutors to spot and correct. Some of the alternatives (e.g. Java or Javascript) were initially designed for non-scientific contexts (e.g. web applications) and come with associated baggage you have to take on, which means that they have bigger initial learning curves.

- It happens to be a language with which I am currently quite familiar[3]. I do not have to look up the language manuals quite so often.

Python's main disadvantage is that it can be relatively slow compared say, to C++ or Fortran, when dealing with very large amounts of data (gigabytes) or for relatively complicated processing. I do not think that we are in that regime.

*Question 5: What is the strategy for managing information?*

The experienced professional software designer tends to think *first* about how he can reflect the intrinsic structure of the information he is working with in the programming structures of chosen the language; he *then* thinks about the algorithms he will apply to that information. In contrast, novice designers tend to think first about algorithms - the active parts of a program - and then worry about how data will be structured.

In this case, our natural information structure is something that would look like a table, with columns of data, of different types (e.g. pressure, temperature) with the understanding that all the data values along a line in the table were logged at the same time, where the time would normally be recorded in one column. This is, of course, an extremely common structure in scientific data.

Python does, indeed, have an excellent module of library routines designed to deal with this type of data, called PANDAS. We are not using it because it is not part of the school's Python installation. (We are working on this!)

Instead, we will rely heavily on Python *dictionaries*. Dictionaries (sometimes called "content addressable arrays") are one of the most useful features of Python. (I would go so far as to say that if you write Python programs without employing dictionaries, you do not really understand Python programming.) Using a dictionary, I can lodge my data into a package, identified by a useful label, and then move it around the program as a bundle. For example:

```
processedData = {}                    # Create an empty dictionary

processedData["times"]       = weatherTimes # Store array of times
processedData["temperature"] = temperature  # Store array of temps
processedData["pressure"]    = pressure     # Store array of pressures
```

I can then move the "processedData" object round the program without having to remember what it contains or how it is arranged within the package. Furthermore, if I choose to add more information to the package later on, I do not need to change the parts of the program that just want to treat the package as a package. I always try to design my programs with an eye to possible future modifications, making the process as easy as possible.

# So, What Will *This* Program Actually Do (In Detail)?

This problem is mainly about reading data in one arrangement, over multiple files, selecting interesting content, and writing it out in another format that is suitable for further analysis. In

---

[3] I have worked in more programming languages than I can now easily remember: at least a dozen, of which perhaps half I would now be prepared to consider using on a new project. At any one time I have a good fluency in the two or three that I happen to be currently using. If I need to switch back to one of the others it usually takes me a week or two to remember all the languages quirks and useful tricks (and to avoid "speaking Java" when I really want to talk "C++" - or vice versa). It is a great mistake to think that you only ever need to learn one programming language. ("*To a man with a hammer, everything looks like a nail.*")

particular, of course, because we are interested in correlating different physical parameters measured at the same time we need to ensure that the output data is arranged so that we have a value for pressure and temperature at a timepoint when we also have a value for the average event rate.

In designing our data handling program we see without too much effort that the problem to be addressed divides into three parts:
1. Data Input: read the HiSPARC raw data files.
2. Average the data onto a common set of time intervals, having calculated the average arrival rate of cosmic rays.
3. Write the data to a file in a format that can be most easily used by later analysis software.

# Data Input

We can decompose this into two design decisions:
1. How do we tell the program where to find the HiSPARC data file it will read?
2. Read the raw data files, and package them.

Most scientific data handling occurs on computers running the Linux operating system. (Linux is an open source derivative of Unix, its commercial ancestor. They are very similar, but Linux is now much more widespread.) For a good many reasons, Windows is rarely the tool of choice.

Most of your encounters with software will be on Windows computers, and most of the time you will probably be talking to the program using a graphical user interface (GUI). Even if, for example, you wish to analyse data from column-organised data files (e.g. using the Open Source program statistical program PAST) you would probably be telling the program about the file it should use via a dialogue box, or perhaps cut-and-past. This is very convenient for simple stuff, but a pain in the neck when you have big datasets to handle with complicated processing that may take hours - and you do not want to be there at that time to give it a little shove along the road whenever needed. GUIs are also very computer-resource intensive, and scientists prefer to keep the computer power doing more productive work.

There are, indeed, many GUI-based programs on Unix/Linux these days, but Unix users and programmers long ago - in pre-Windows days! - developed a style of work known as “pipelining” which constructs complicated chains of data processing out of smaller components in such a way that what comes out of the back end of one can go into the front end of the next-in-line. Programs like this run “from the command line” not from a menu. We often give them some basic options as “command-line parameters”. On Windows, this means that you need to open a Command Tool window (sometimes known as a “DOS Window”) and at the command prompt type:

python eventsVsWeather.py <*weatherFilePath*> <*eventFilePath*>

where <*weatherFilePath*> and <*eventFilePath*> are place holders that you replace with the path to the data files. (Look up “path” if you do not understand this term.)

They may also read information on “standard input” and write to “standard output”. (These are common terms you will certainly encounter if you do any Linux programming, and they are also commonly encountered in Windows programming - though they don’t have quite the same foundational basis.) Linux programs can easily be daisy-chained or “pipelined” such that standard output from one program becomes standard input for the next-in-line.

My program for handling HiSPARC data will write its output to standard-out. That means it will scroll into the command window, unless it is redirected to file using a right-chevron, for example:

python eventsVsWeather.py <*weatherFilePath*> <*eventFilePath*> **>** <outputfile>

On Linux I might pipeline the data straight to a graph-plotter as with:

python eventsVsWeather.py <*weatherFilePath*> <*eventFilePath*> **|** gnuplot <graphspec>

You can also do this on Windows if you have the **Powershell** installed.

My design intention for handling data input to the program is therefore:
1. Read two command-line parameters to get two file names.
2. Read the weather data file and package the contents into a dictionary, where the keys to the dictionary will be the names of the data columns in the file.
3. Read the cosmic ray event data and package the contents into a dictionary, where the keys to the dictionary will be the names of the data columns in the file.
4. Return the two packages for further data handling.

I know that the input files are ASCII text, with tab characters separating the columns. Hence, I can use the built-in "regular expression" module, and particularly the "split()" function which will take a line from the file and turn it into an array of "words" separated at the tab characters.

I can then append each of these words onto the end of a list accumulating the data for each column. Here I will use the **append()** method built-in to Python lists. At present, I will not worry that this is a relatively inefficient operation. It is easy to understand and hard to get wrong. If (and only if) efficiency becomes important (e.g. perhaps if we want to read several years worth of data) I might consider more efficient - but inevitably more complex and error prone - algorithms. (In reality, if I wanted to process ten years worth of data I would use and entirely different route to getting information from the HiSPARC database.)

# Data Averaging

There are simple and complicated approaches to data averaging.

The simplest approach is to divide the full time interval into shorter periods, and average all the data in each period, replacing, say, one hours worth of data with one data point. The disadvantage of this approach is that is the average values you calculate can depend on exactly where you set the boundaries of the periods - there is no obvious best way to do it.

A more sophisticated approach is to take a "running average". In this case the average values that are output are calculated from a "window" around the current time and this window moves forwards as the time moves forwards. Each data value may be used many times in the averaging process. The advantage of this approach is that it produces a smooth output result, but it is a bit more complicated to implement.

Further complications can arise if the input data values are not evenly spaced in time. Averaging evenly spaced values is easy: add them up and divide by the number of values. When they are not evenly spaced we need to first multiply each value by the time interval that it represents, add all those up, then divide by the total time interval.

You also ought to worry a little bit about whether you think that the average you have calculated is to be considered representative of the time in the middle of the averaging period, at the beginning, or at the end. (All options can be sensible in different circumstances.)

At least to begin with, we will take all the easy options. The weather data is evenly spaced so we can use the add-up and divide by the number of values algorithm. We will also just take hourly periods (no running averages) and just take the average value to represent the whole hour (labelled by the time at the end of the period).

For calculating the average event rate, we will take the periods used by the weather data averaging and simply count the number of CR events in this period, and divide by the time interval.

# Evaluation of the Design

The principle advantage of this design is that is uses no complex methods. Everything should be as easy to understand as it is possible to make it. The principle disadvantage is efficiency of processing and lack of flexibility to develop the design in a more sophisticated direction.

There are several reasons why this design may need to be developed.

Firstly, it used the data files downloaded from the HiSPARC public database, via its web interface. These come as ASCII format tab-separated column files. This is an inefficient format for moving large amounts of data (typically about 3x as large as moving data using a binary encoding). This would start to matter if we wish, for example, to download several years worth of cosmic ray event data, when the files would probably grow to Gigabyte size. Files of this size take a long time to move over the Internet, occupy large amounts of space on disk and also take longer to read into programs.

There is an alternative and more efficient way to download and use HiSPARC data, employs the HiSPARC-specific SAPPHIRE Python module, which brings the data across in the binary “Hierarchical Data Format” (well known in the scientific world). Data in these files is internally labelled can easily be extracted using modules in the Python library.

More sophisticated approaches come with their own disadvantages, in particular there are bigger learning curves. Although one eventually finds it possible create more complex processing routes with fewer lines of code (all the real work is done in the library modules) you need to take some time finding which library routines do the job you need done, and you also have to understand exactly how to feed them with information in the right format, and how to make sense of the information they generate. This is a particular skill (but an extremely useful one) that is somewhat different to general programming skills.