SonicLib Reference Manual

The SonicLib group

April 12, 2013

Contents

Chapter 1

Introduction to SonicLib

1.1 What SonicLib is for

The last years have seen a steady diffusion of three-dimensional ultrasonic anemometers, accompanied by a spread of applications from scientific research to environment and human health protection, siting and monitoring of wind power plants, and many more.

This is not surprising as instrument costs have greatly reduced while performance and robustness did improve. Compared to mechanical anemometers, ultrasonic anemometers enjoy many practical advantages. Among them, and with no claim of completeness:

- Ultrasonic anemometer are immune to "wind calms": they can record extremely slow or irregular winds.
- Having no moving parts an ultrasonic anemometer minimizes low-pass filtering and resonance on input signal.
- Most ultrasonic anemometers have accuracy and resolution greatly exceeding mechanical models.

Last but not least, ultrasonic anemometers may sample wind very fast, at rates of many tens of Hz. This characteristic, together with an inherent high sensitivity, allow using ultrasonic anemometers to measure directly atmospheric turbulence.

All of this does not come free: respect to mechanical anemometers, sonic anemometers produce a huge mass of data. Gathering it, recording in an organized manner and extracting useful information demand non-trivial processing.

SonicLib attempts to satisfy this need in a simple to use way, yet allowing users to understand the details of what's going on, if they want/need.

1.2 Why a new tool?

Much work has been done in the past on sonic anemometer data, and many excellent tools have been developed as a result. Mentioning them all would be quite an endeavour, but sure the question of why trying developing a *new* one makes some sense arises spontaneously.

These are our tentative, multiple answers:

- We would like to really learn, and there is no better way of achieving this objective than doing on our own. This is after all how SonicLib started: as the visible outcome of the atmospheric turbulence laboratory held at Physics department in University of Milan on 2012.
- Of the many treatments involving sonic anemometer data final processing is perhaps the most delicate: Were our computations sensible? Are the physical assumptions they base upon really satisfied by current data set? Is accuracy known? These, and other similar questions of paramount importance, devise sensible answers, much easier to obtain if we users "own" the code.
- Much work has been made, but much is to be still done, too. This is true of both the techniques and the foundations. We strongly feel a diffuse need exists of an idea exploration platform, which is as easy to modify and extend as possible. This platform is not likely to take the form of a product-like software package.
- Most existing software packages focus narrowly on one or few applications (typically on flux measurement in the context of finite duration experiments). But the scope of possible uses of sonic anemometers is much broader than that, and we feel the moment is mature for flexibility, in addition to excellence.

These motivations have led us to conceive SonicLib, and helped us to define some basic requirements on it:

- SonicLib should be useable both as a toolbox with which to compose processing procedures, and interactively as a sort of calculator. This is why we decided using an interpreted language, and not a compiler.
- All SonicLib source code should be freely available for inspection, study, extension and correction. That is, SonicLib should be at least *open source*. And in addition, it should be *easily understandable*, even if writing it this way might decrease its efficiency (but just note how even the tiniest netbooks in use to date may deliver massive, and almost always unused, computing power). Simple solutions are preferable, in our opinion, to complicate ones. Simple programs, by the ways, are easier to change, or throw away and replace, if they are wrong.
- As a corollary of the preceding point, we took the early decision of adopting a programming language which is as close as possible to the nature of the problems at stake. Our choice went to R, the publicly available implementation of the S statistical language. Its almost limitless support of computational statistics, backed by an immense and growing user basis, matches quite well the current "statistical" flavor of sonic anemometer data analysis and processing. Besides, R offers huge graphical and scientific computing power.

All that said, we understand being in debt with the research groups and individuals who preceded us. To them all goes our gratitude. Thank you all, and forgive our immense ignorance and immodesty.

We're sincerely and authentically in love with the subject, however.

1.3 Why R?

We might have decide to develop SonicLib as a Fortran library, and maybe a number of small utility programs written once again in Fortran.

That would no doubt have allowed us to reach a great efficiency, and may have been a great occasion to show our programming prowess.

But this would have missed our most important concern: doing something easy to understand and use.

To date many incredible software packages exist of a very high level, like Python and SciPy, or Octave, or Matlab. Respect to Fortran they are more expressive, in the sense they require much less writing to reach the desired result.

R (see [32]) is one of these high level languages. The reasons we adopted it for SonicLib are:

- R finds its natural niche in the field of data analysis, and ultrasonic anemometer data processing is a special case of it.
- R provides extensive support for computational methods, statistics and graphics: all these play an important role in ultrasonic anemometer data processing.
- R is supported by the work of a very large comunity of volunteers, and is a healthy and evolving project.
- R is open source, and hence gives an opportunity to understand the innards without the refrain coming from the infringement of commercial secrets.

1.4 Style matters

Using R, a high level language, is a wonderful occasion, but possibly also a nightmare: expressivity may be used to write code which is shorter to read, easier to understand, less cumbersome to memorize (in our heads, not computer RAM banks), and lovely to understand. But also, by adopting the most terse and idiomatic language constructs would make code a real mess (possibly very efficient).

In SonicLib our top priority is being understood, and communicate. Efficiency is "also" important, but comes second: would a conflict arise, readability and understandability will prevail.

As you will see by inspecting SonicLib sources, code has been written "the simple way", sometimes despite speed of execution or even "elegance". This is entirely intentional: the coding equivalent of a curvy and a bit dull readable handwriting, opposed to those scrawls which show a lot of personality but do not convey useful information.

The possible loss of efficiency might be a problem, but we feel it is really not. Today's reasonable portable computers pack enough power to largely exceed the specifications of supercomputers some the authors have met in their student life in the Eighties of last century. Used the right way they are fast enough to let us forget any efficiency concern.

For the very same reasons we invested most of our time in this Manual: as humble as it is, it got no less than 70% of total development time. Another advantage of the expressivity and conciseness of R: less code to write means having more time to spend communicating with people.

1.5 The SonicLib Development Process

At least initially we strive to deliver SonicLib as a single R source file, named `SonicLib.r`, and complemented by a reference guide (this document).

The official source and reference manual are distributed through the project site, `http://www.boundarylayer.eu`.

Developers, in addition to maintaining `SonicLib.r` and issuing new releases of it, also work on experimental code which is shared through e-mails among work group members, on a voluntary basis. After consensus has settled on experimental code, this is moved to the `SonicLib.r` file, and a new version is then released to the public.

This way of doing is maybe not the most secure, but as far as the work group dimension is "small" its simplicity outweights its possible drawbacks.

SonicLib is the result of co-creation among people who may act both as leaders or followers, as needed/possible/desired. After all nothing requires the figure of leader to be unique in an organization. As a community, we organized in a way some personnel management consultants would name a *web of inclusion* ([3]). Inside of it anyone of us has an opportunity to settle where their attitudes and experience are best fitting. And to change, if they will, as they evolve.

This fits well with the multi-faceted (and related) cooking pots we must stir almost simultaneously:

- Learning the theory and techniques.
- Experimenting new ideas.
- Writing code.
- Performing tests.
- Writing documentation.
- Finding and organizing data sets.
- Relating with people inside and outside the work group.
- Building harmony and community within.
- Finding ways to present (to graph, film, describe) sonic anemometer data and processing results.

1.6 Symbol Glossary

f_s Sampling rate (s).

L Obukhov length.

n Number of data actually present in a sonic raw data files.

n_s Number of data expected in a (hourly) file, identical to $3600 \cdot f_s$.

t_a Averaging time, defined as length of averaging period expressed in seconds.

α First rotation angle in eddy covariance, or individual rotation angle in planar fit.

β Second rotation angle in eddy covariance, or first plane rotation angle in planar fit.

δ Third rotation angle in eddy covariance, or second plane rotation angle in planar fit.

Chapter 2

SonicLib Workflow

2.1 Why a Workflow?

Many different ways exist when performing data processing, and this is specially true in cases like the processing of ultrasonic anemometers, where steps are many and some of them might be interchanged.

All these methods may be broadly classified in two blocks:

- Invoking a fixed sequence of steps within an automatic data processing program, or
- Performing the steps interactively, checking intermediate results.

Of these blocks the first is gaining diffusion along with large networks gathering data continuously in unattended mode, where it is imperative to process data as soon as they are available in sufficient number, not being realistic in most cases to simply transfer bulk raw data files to a data processing center.

Automatic sequences are fixed however, and their rigidity makes their use as an easy to follow use narrative of quite little interest.

Because of this, we'll imagine processing to occur interactively.

Interactive data processing is by definition flexible. But nonetheless, if perused will tend to crystallize along a more defined "workflow". *This* can provide a better use narrative.

Of course many workflows are possible, as we've seen for chain of automatic data processing programs. Se we'll pick one at random, and use it as a metaphor of what could be accomplished interactively using SonicLib building blocks. The workflow adopted, as many other, may be used as a basis for a possible automatic procedure.

2.2 Reference Workflow

Our reference workflow is defined within the context of a measurement campaign, in which data from a single sampling point are retrieved from time to time.

The major steps of the reference workflow are:

- Original data, maybe in binary form, are converted to a convenient ASCII form, namely SonicLib standard raw data file. As a first effect of this conversion, data may be visually inspected or plotted. This possibility is very powerful, as it allows to see whether data collection has generated or not some macroscopic error, and to perform the first open minded explorations.

- SonicLib raw files are sequentially read and "averaged", so that a set of mean, variance and covariance values is obtained. This step is supported in fact by a specific function in SonicLib, `average.sonic.file.set`. Averaging may be viewed in various ways: as an artifice to filter a "noisy" signal, and as a way to "extract" useful information from the huge mass of raw data. The averaging step needs a choice, by the users, of an averaging step, whose choice may yield in some occasions quite different patterns.

- During averaging a first set of quality indicators, as number of elementary data used per average, relative non-stationarity of wind components, non-steadiness for covariances), have been generated. Now they may be used, in addition to some key averages and covariances, to locate macroscopic deviates and exceptions. This allows to form a first impression about overall quality of data, and locate specific SonicLib raw data files where to look for in greater depth. Depending on the objectives of the campaign, this may also be the point where a subset of "good" data is elected and put apart for further study. Or in alternative, when to flag some data as "bad" in view of their imputation)thet is, replaced with an estimate, as needs to be made for example when preparing inputs for Lagrangian dispersion models).

- Why exceptional data are so exceptional? In some cases budget restrictions do not encourage answering such a non immediately productive question: you may just remove them from the official set, after all. Yet reflecting on outliers may help fostering a deeper understanding of the data set. For example you may discover site-related idiosynchrasies (like that large beautiful oak on SW responsible of generating massive turbulence with her leaves), or a sonic anemometer showing the first sign of failure, or an almost-systematic trouble with power supply. SonicLib supports investigate on these "whys" by means of routines which can stitch consecutive hours together, extract specific blocks from an hour, plot actual values and spectra, investigate auto- and cross-correlation. These very same instruments may of course be also used to study less exceptional data, gaining a better understanding on the campaign.

- Averages may be interesting to an expert eye, but quite seldom are informative to the layperson, or the mindless rigidity of a dispersion model. To get some sense from them they should be translate to physically meaningful quantities like energy fluxes, turbulence and stability indicators, or precision wind vector. This is done by invoking one of the processing routines like eddy covariance (SonicLib supports both classical and planar fit flavors) or wind turbine specific calculations.

- Once physically meaningful data have been obtained (and possibly "good" data have been separated from junk) the residual problem exists of delivering them in a form suitable for use. This may be well organized data files. Or graphs, suitable for inclusion in a scientific paper or technical report. Data restitution is addressed by SonicLib in various functions, many of them allowing extensive customization.

2.3 Workflow: A More Specific Example

Maybe, a specific and hands-on example of workflow is easier to understand than a zillion words. So let's imagine we desire to use SonicLib to analyze a data set in view, say, of a scientific paper or a technical report.

First of all, we need to know our starting data. In which format are them? Almost surely, they are not written in SonicLib raw data format (described in section 3.1), and rather are available in some other form.

The first question then is, whether a converter from this form to SonicLib raw format is available. If not so, then the first step is writing one (and, possibly, contributing it to the SonicLib group).

But suppose a converter already exists. Then, we first need using it - maybe after having read the appropriate section in chapter 5 of this Manual.

The actual mechanism will in general differ somewhat depending on input form (and then converter), but in most cases what will actually happen is that some program is run automatically on the original file set, to produce a collection of SonicLib files containing exactly the same information, just in a different format allowing SonicLib to access the data.

In this (indeed very delicate) step a bit of organization helps. As most converter procedures operate *en masse* on all files in a directory, producing a SonicLib file set in another directory, it is highly advisable to place all original file of the same campaign in a single directory, or a seto of related directories, with no other data in them. And also, it makes sense to store converted data in another, separate and specific directory. This is not difficult, if an operating system like UNIX, Linux or Windows is used, with their extensive support for creating and managing directories.

To be specific, let's imagine our input files are in SLT form, with all data of the Second Capocaccia 2012 Campaign in directory `/raw/Capocaccia/2012/Second`. We first decide the converted data will be in (say) directory `/datasets/Capocaccia/2012/Second`, and then run procedure `convertSTL.py` which enumerates all the hours within the input directory, launching on all of them the actual converter `readslt`.

As you will discover in section 5.1, in order to run `convertSTL.py` you need to collect all information about your data, especially how many columns have been gathered in addition to the sonic quadruples, and which unit conversion rule to apply on each of them. This information will result in a small configuration file, whose contents is read by the conversion procedure and used to generate the SonicLib raw data files.

If this little decision has been taken correctly, after running the program `convertSTL` you will find in the destination directory as many SonicLib data files as the number of hours in original data.

The next step is might be to checking data quality in some simple way. But unfortunately the number of raw data may be immense, much larger than a human might cope with: at 10 Hz, a sonic anemometer produces 36000 data rows per hour, and a typical campaign may endure a couple weeks, that is around

14 by 24 hours, that is 336 hours, or more than twelve millions data records. What may realistically be checked at this point is plainly the file size, trying to locate hours during which an abnormal amount of data has been collected.

Luckily, SonicLib allows to combine this, and other more sophisticated, data quality checks with mass averaging: function `average.sonic.file.set` reads sequentially, in the right time order, all SonicLib files in `/datasets/Capocaccia/2012/Second`, and computes all averages on the time basis selected by user. While doing so, it also counts the number of data per hourly file, compares it with the minimum acceptable threshold, and discards automatically all non-conforming hours.

On the hours surviving this first check, determination of non-stationarity is also performed on key data column.

If `average.sonic.file.set` terminates successfully, then memory contains a new data item holding all results.

As most intermediate results in SonicLib, the output from `average.sonic.file.set` is an R list *belonging to a class* (in this case, `sonic.avg.data`). If you are a "final" user, then the result belonging to a very specific class means very little. But to SonicLib this attribute is of great help, as any step checks its input to be of the right type: if it is, then all rights and the desired computation is done; if it is not, processing stops immediately and, if an appropriate option has been specified, a nice message is printed to the R session informing on what happened.

But before to proceed with the second processing step, perhaps we like to check data quality. As explained before this is possible by checking averages directly (for example by plotting them and looking for anomalous values) or using non-stationarity indicators to lead the way. This step, of a very visual nature, greatly benefits of the extensive graphic capabilities of R. Some ultrasonic anemometer specific graphs (for example autocorrelation, cross-correlation, ogive) are implemented within SonicLib as extensions to R, in a way straightforward enough to allow easy customization would this be needed.

What to do, once data quality has been checked using averaged values? The answer depends of course on context - namely, on what we want to do with data. In our very specific case, writing a paper, we may be free to reject data not fulfilling our quality criteria (this may not happen in other cases, like for example using sonic data to feed a lagrangian pollutant dispersion model).

In order to steer rejection with minimum hassle, it may be convenient to assign each row in averaged set a *score* weighting for example the "number of troubles". Then, you may use the powerful R indexing on condition capabilities to retain only data with a "trouble load" smaller than some predefined positive value.

Once you have decided which data to retain in your analysis, it comes finally the time to actually process them. In a common case, this means to perform eddy covariance over the averaged values, which in SonicLib is made using function `eddy.covariance`. Other data processing functions are available in SonicLib, meeting the needs of other application domains (for example, wind energy). This broader than usual scope is one of the distinctive features of SonicLib, compared to other ultrasonic anemometer data processing libraries - but in this example we'll not pursue this way.

Rather we notice that function `eddy.covariance` has taken as input a data item of class `sonic.avg.data`, worked on it, enriched it with (many) additional columns, and made the result available as another data set, this time of time `sonic.eddy.cov.data`.

Two questions may arise at this point.

First, we may wonder how to preserve our results: after all, all processing steps we've performed in R did produce data in memory. We saved nothing, at least apparently.

And then, maybe more importantly, we may ask ourselves what to do with data.

We'll attempt some possible and sensible answers in the next two subsections, from a purely computer-usage standpoint. Of course the questions are deeper, and involve a part dealing with scientific meaning, opportunity, and stuff like that. But in this example let us be plain and simple. We promise to deal with some more cogent subjects in later more operational sections, to the extent reasonable in a Manual.

2.3.1 Preserving with results after we got them

Fine! You just have completed your first eddy covariance processing, and placed it into an R variable `ec`. Maybe this seems not that a great endeavor (after all the computer performed all the hard work for us), but sure it did cost some work.

More important: suppose to throw away the result `ec`. If you really needed this data set the problem arises to rebuild it: how may you *guarantee* you will obtain the very same result?

In theory the answer is trivial: if you give the functions exactly the same data and options then the computer will reconstruct your results with digital accuracy. The real question is: how may you be sure you will use the same parameters? This seems silly, when you imagine having to rebuild a data set after five minutes. But wait five months, maybe under referee pressure, and you'll agree with us.

This is an example of the classical data processing nightmare of (lack of) *configuration management.*

To overcome this issue there exist many little tricks. We share a couple (if you already know them, please forgive our silliness).

A first way to be able to redo a computation exactly in the same way as before is by embedding it in an *automatic procedure.* This approach works, as far as we have seen, using R in general, and not only with SonicLib. Additionally: if instead of using the bare R terminal session you resort to an IDE (like for example RStudio) then it is very easy to perform computations interactively, copy them line by line, and pasting to a script file which can be executed in future at will. Giving this script a name easy to remember or distinguish helps somewhat; one of us for example packs all processing command for a paper in a script named `process.r`. Using the same name for any different campaign allows to get at the processing script with mindless certainty, something helpful under stress.

A second way is, saving results permanently. R allows to save data in many ways. Maybe the simple is saving the session when you leave R, by answering a `y` after command `q()`, when R asks whether you want to save the current session.

As easy as this approach is, it also has a major drawback: the more you work with an interactive session, save it, then restore on the next day, the larger the set of defined variables becomes. R saves all these data in memory, and if you pay no attention you may end up with many tens of megabytes of physical memory allocated to data you whose meaning no one may remember.

A cleaner approach is saving data items by hand, selectively. In SonicLib, because of the way results are held in memory (that is: as lists, one of whose element is the data frame containing the "real data"), you may want to save the whole result variable, or the data frame only.

In the first case you may use the `save()` command. If you check R help you will discover `save()` takes some parameters. One of them is the name of the file you desire your data be saved to, and another is the variable you want to save. Data stored to disk using `save()` may be read back into memory using the sister command `load()`.

In the second case you will want to use command `write.csv()`, with the data frame only as the variable to save. Command `write.csv()` creates a CSV file you may read later on in a spreadsheet or a database program, extending your data analysis capabilities. The CSV file may be read back to memory using command `read.csv()`.

2.3.2 What to do with data, once we got them

No question, averaging and eddy covariance produce *many* columns of data. These columns are stored to member `data` in the variable helding results in memory as a *data frame*, a powerful R tabular data structure you may use in very many ways.

The simplest thing you may do is retrieving columns as they are:

```
u <- ec$data$u.avg
v <- ec$data$v.avg
w <- ec$data$w.avg
t <- ec$data$t.avg
time.stamp <- ec$data$t.stamp
```

You may then plot your data:

```
plot(time.stamp, sqrt(u^2+v^2),type="l")
```

Or, perform any kind of statistical analysis:

```
median(sqrt(u^2+v^2))
```

In short, once you have produced your data you have all the power of R to analyze, transform, read/write them. Only your creativity will be a limit.

2.4 Automatic Procedures: Usefulness and Danger

Measurement campaign of finite duration are just one of the possible uses of sonic anemometers. Another use, quite common on those days, is as part of measurement stations operating in continuous mode within a station network.

In this case, many station produce simultaneous streams of data continuously, in conditions often completely out the reach of an experimenter's control. The processing of this immensely huge flow of data can not be realistically made by a single (or multiple, for all what matters) human operator. By necessity we have to rely on *automatic procedures*.

Usefulness of automatic procedures is then obvious: they allow making some sense out of an otherwise unmanageable pile of meaningless numbers, at a sustainable cost.

But automatic procedures bear with them a sort of original flaw: their operation being fixed and predictable, they tend to induce laypeople perceiving them as invincible oracles. Real world often lacks this high level of reliability however, and sonic anemometer data processing is no exception.

Maybe, the most striking difference between an interactive workflow (like the one we adopted as reference) and an automatic procedure is the former allows a greater opportunity of *performing quality checks* on data and intermediate results. This can not be done at the same extensive level by an automatic procedure, which by its very nature is rigid.

By their very nature, wind measurements are highly site-dependent. Not only are sites different one from another in a myriad of details affecting for example rejection of outliers: they also change their characteristics with time. The data processing should reflect these evolving individualities: what's normal at a site might well be rubbish at another.

Just as a practical example, we may consider a measurement site intentionally placed within a plant canopy. There is perfectly normal that friction velocity exceeds the 30% of wind speed. An intelligent automatic procedure should be able to ascertain the "normality" of situation, and to *not* removed high u_* records from the "good data" set.

But what's here true of friction velocity may be true in another place for wind vertical speed from a direction, and at still another place for a strange negative peak in sensible heat flux.

The possibilities are so numerous (in the order of billions) that trying to devise an adaptive and sensible response to all of them would result in a monster of complexity. And since procedure complexity is inversely poportional to reliability, such a behemoth would almost surely be very smart in theory, but unusable in practice.

Maybe, a more realistic approach consists in designing automatic procedures which are flexible enough to allow treating the most common key cases by a few *configuration options*, and use it in the majority of "quiet" sites. Then, at sites where things are really awkward, the possibility exists of using ad-hoc procedures (also aautomatical).

The most appropriate value of configuration options then could be identified by means of *characterization campaigns*, one per site, aimed at understanding what specifically happens there to wind. Or, if you prefer, what "normal" means there.

Configuration options also bear an original sin with them: once established, users may tend to figure them out as fixed and immutable. Site characteristics change with time however, and as part of a sensible maintenance protocol configuration should be routinely adapted to the new circumstances. How? Of course, with new characterization campaigns.

Now: characterizaton campaigns are done on raw, not pre-processed data.

For characterization to be done and re-done without stopping station operation for months it is necessary to store raw data at any station for a long time (at least one year). This is made routinely and transparently by data loggers designed specifically for sonic anemometers (like for example Meteoflux Core, by Servizi Territorio srl), but might not be present a priori.

And once you have downloaded your bunch of characterization data, you still have to use an interactive workflow to explore data, define normality on solid statistical grounds, and devise revised values for configuration parameters.

We may agree, then, that automatic procedures do not really operate unattended, abandoned to themselves, relying on the ultrasonic anemometer inherent reliability to produce a stream of data more and more garbled as conditions change from when configuration parameters have originally been devised. Rather, in a healthy processing environment procedures operate as faithful robots the way human operators would, would they have time to do, with someone governing them by adjusting this knob here, moving that switch there, and checking all runs smoothly.

Then, in practice, automatic procedures and interactive workflow can not be blindly separated. Execution of the latter forms the basis of making the former tick.

Chapter 3

Data Types

This chapter contains the description of the most important data types used within SonicLib, as for example the standard file format for ultrasonic anemometer raw data. Other, less important data structures are described as part of individual function descriptions.

3.1 Standard Ultrasonic Anemometer Raw Data Files

Raw ultrasonic anemometer data are used within SonicLib mostly in the form of standard formatted CSV archives conforming to the naming specification

```
YYYYMMDD.HH.csv
```

where

YYYY represents the year;

MM represents the month in year (01 to 12);

DD is day in month (01 to 31);

HH designates the hour (00 to 23).

Collectively the fields YYYY, MM, DD and HH identify a *hourly time stamp* which refers to the beginning of data acquisition period for given hour. More specifically, the file named `20120501.12.csv` contains data from 2012-05-01 12:00:00 to 2012-05-01 13:00:00, the first time limit included, the second excluded.

Hourly time stamp used in file names is always referred to *local* time. Different anemometers and campaigns are distinguished attributing each of them a different directory.

Files in standard raw format contain five mandatory columns containing within-hour time stamp and ultrasonic anemometer quadruples, and zero to any optional columns containing analog values.

Columns are comma-separated, with dot used as decimal separator.

3.1.1 Mandatory Columns

Mandatory columns are five, with the following names and meanings:

time.stamp Within-hour time stamp $0 \leq t < 3600$, in seconds from beginning of hour; this data item should be accurate up to seconds, although fractional values are also in common use.

u Wind component aligned to the West-East axis in local geographic reference (m/s);

v Wind component aligned to the South-North axis in local geographic reference (m/s);

w Wind component aligned to the Downwards-Upwards axis in local geographic reference (m/s);

t Ultrasonic anemometer temperature (Celsius degrees).

3.1.2 Optional Columns

Optional columns may not exist in a file set. If on the contrary they do, then their names must belong the following list:

q Molar concentration of water ($\texttt{mmol/m}^3$);

c Molar concentration of carbon dioxide ($\texttt{mmol/m}^3$);

a Molar concentration of ammonia ($\texttt{mmol/m}^3$);

m Molar concentration of methane ($\texttt{mmol/m}^3$);

temp Temperature (Celsius degrees);

hrel Relative humidity (Celsius degrees).

Two optional columns with same name are allowed to occur.

3.1.3 Time stamps in SonicLib raw data files

The time stamp associated to each data record in a SonicLib data file is a very important data element, as it allows locating each sampled data in time.

Ideally, time stamps should be real numbers able to identify exactly the time instant a record has been produced. Data being collected sequentially in time, the sequence of time stamps should be monotonically increasing with respect to index value. In other words, if t_i and t_{i+1} denote the time stamps of two consecutive records, it should always be $t_i < t_{i+1}$.

Experience suggests however that real world time stamp are neither non-unique nor monotonically increasing.

About uniqueness: data collection equipment encode time stamps with a finite resolution. The reasons for this ranges from the finite resolution of existing (and future) real time clock chips found in computer systems, in the need of saving space on disk, and in many other reasons connected with engineering constraints and necessity of efficiency. That said, we can assume real world time stamps to be multiples of a fixed minimum time duration δt, whose length may be as long as 10 seconds in certain systems. Real world time stamps may then contain many repetitions, if time resolution δt is larger than $1/f_s$. For example, a sonic anemometer operated at 10 Hz will produce data records whose time stamps, recorded with a time resolution of 1 s, will repeat 10 times. We then have to admit the possibility that $t_i \leq t_{i+1}$.

This violation of uniqueness is not the only case of violation of increasing monotonicity. Real time clock circuits are in fact subject to drift, and sooner or later the time comes when their time must be adjusted. Depending on the direction of this adjustment, a gap or a wind-up in the sequence of time stamps may occur.

These possibilities lead us to introduce the concept of *time stamp sequential regularity* (sorry for this ugly name - maybe we'll find a better one).

A sequence of time stamps $\{ti\}$ is regular if the following conditions are met:

1. For any indexes i and j, with $i < j$, it is $t_i \leq t_j$.

2. In addition, $(j - i - 1) \cdot \delta t \leq t_j - t_i \leq (j - i) \cdot \delta t$.

A regular time stamp sequence is guaranteed to not contain winding or overlaps up to δt, which is sufficient in all practical applications.

Unfortunately, verifying directly that a sequence of time stamps is regular involves a number of operations in the order of n^2, where n denotes the length of the series. This is less than acceptable for daily use, as it is not uncommon to deal with time stamp sequences longer than 36000 samples.

Given the importance of time stamp series regularity this sounds a pity, and may lead to search for some more or less clever algorithm characterized by a linear complexity. This path is quite interesting, but we will not follow it.

In fact, it may be observed that in a healthy data acquisition system events leading to time series irregularity occur very rarely, the two most common being real time clocks adjustment, and gaps induced by failures in the sonic anemometer, the data acquisition system, or the cable joining the two. (Less common cases are also possible, like sonic data sampling rate reprogramming).

But when such events do not happen (and this is the majority of cases) it always happens that a data file will contain a number of data, N, closely approximating the value $3600 \cdot f_s$ for hourly-organized files (like the ones described in this section). In these normal cases, the sequence of time stamps is also regular, and it makes a lot of practical sense to use the predicate $|N - 3600 \cdot f_s| < \varepsilon$ as a proxy of regularity. Checking the size of a data file is in fact a trivial thing, which may be accomplished in a single step (in R all you have to do is issuing a command like `N <- length(d$data$time.stamp)`.

We notice *en passant* that this criterion is not absolutely exact. It is not difficult to devise cases in which $|N - 3600 \cdot f_s| < \varepsilon$ but the series is irregular. But these cases arise from the accidental combination of unfrequent cases like real time clock adjustment in backward direction plus a failure in the sonic suppressing emission of a number of data exactly equal to the product of sonic sampling rate and real time clock adjustment. The probability associated to these independent rare event is the product of their component probabilities, a number so small that we can safely neglect these combined events.

The problem then arises to identify sensible values for the tolerance ε. A reasonable tolerance should include the little clock divergences naturally occurring with existing technology.

An example of these irregularities is the actual data acquisition frequency may deviate slightly from the nominal. The practical reason this can (and does) happen is that event generation used to trigger sonic anemometer readings is often delegated to counters, as found in many microcontrollers. These counters generate pulses according to a time base established by the processor quartz, and this latter is not necessarily an integer sub-multiple of one second.

These deviations are usually small, but may reach a 0.02 %. Deviations like these are practically ininfluent on ultrasonic anemometer data processing, but must be taken into account when setting a sensible tolerance.

"Reasonable" default thresholds for tolerance ε are established, and documented, for routines dealing with data input. Their value may be altered to reflect more detailed knowledge. Upon reaching or exceeding the threshold data read functions reject the current hour, thus allowing all other SonicLib components to assume time stamps are regular.

3.2 The object type "sonic.raw.data"

Objects of type `sonic.raw.data` are created when reading raw data files, and are lists composed by two members:

data Data frame, containing columns with same names as the ones found in standard ultrasonic anemometer raw data files (see 3.1 for details).

sampling.rate Nominal sampling rate, specified as an argument of function get.raw.data (see 6.1).

3.3 The object type "sonic.avg.data"

Objects of type `sonic.avg.data` are created using function `average.sonic.data`. They are lists defined as follows:

data Data frame, containing averages and stationarity indicators as described later on.

delay Value of parameter `delay` as passed in call, for future reference.

spike Real number representing the spike detection threshold expressed in terms of sigmas and common to all data columns in SonicLib standard raw files; or, element of type `spike.detection.spec`.

spike.treatment String containing the name of spike treatment method used.

sampling.rate Nominal sampling rate, as coming from the data object `sonic.raw.data` from which averages have been computed (see 6.4).

averaging.time Averaging time used when processing data.

The `data` element in objects of `sonic.avg.data` class, of type "data frame", is composed by the following columns:

t.stamp Time stamp identifying the beginning of averaging interval.

n.data Number of data in averaging interval (should be indicatively equal to $f_s \cdot t_a$, the averaging time. It may differ a little bit, due to clock imprecision in sonic or data acquisition system (or both). If much smaller than expected value, then it is legitimate to suspect a gap in data (most often due to power failure, data acquisition system failure, or ultrasonic anemometer lock).

u.avg Mean value of wind component u, in m/s, computed on valid data.

u.min Minimum value of wind component u, in m/s, computed on valid data.

u.max Maximum value of wind component u, in m/s, computed on valid data.

v.avg Mean value of wind component v, in m/s, computed on valid data.

v.min Minimum value of wind component v, in m/s, computed on valid data.

v.max Maximum value of wind component v, in m/s, computed on valid data.

w.avg Mean value of wind component w, in m/s, computed on valid data.

w.min Minimum value of wind component w, in m/s, computed on valid data.

w.max Maximum value of wind component w, in m/s, computed on valid data.

t.avg Mean value of sonic temperature, in Celsius degrees, computed on valid data.

t.min Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

t.max Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

uu Covariance between wind components u and u (that is, variance of u) in m^2/s^2.

uv Covariance between wind components u and v in m^2/s^2.

uw Covariance between wind components u and w in m^2/s^2.

vv Covariance between wind components v and v (that is, variance of v) in m^2/s^2.

vw Covariance between wind components v and w in m^2/s^2.

ww Covariance between wind components w and w (that is, variance of w) in m^2/s^2.

ut Covariance between wind components u and sonic temperature in m C/s^2.

vt Covariance between wind components v and sonic temperature in m C/s^2.

wt Covariance between wind components w and sonic temperature in m C/s^2.

vel Mean horizontal wind speed according to instrument reference in m/s.

vel.max Maximum horizontal wind speed according to instrument reference in m/s.

resultant.vel Resultant horizontal wind speed according to instrument reference in m/s.

vel.sd Standard deviation of horizontal wind speed according to instrument reference in m/s.

vel3 Mean of wind speed elementary data in the instrument's horizontal plane, in m^3/s^3.

vel3.3 Mean of wind speed elementary data using all three wind vector components, in m^3/s^3.

u.j Mean of x component of horizontal wind versor, $\frac{u}{\sqrt{u^2+v^2}}$ (non-dimensional).

v.j Mean of y component of horizontal wind versor, $\frac{v}{\sqrt{u^2+v^2}}$ (non-dimensional).

dir.sd Yamartino estimate of standard deviation of wind direction in horizontal plane (degrees).

u.rns Relative non-stationarity of wind component u.

v.rns Relative non-stationarity of wind component v.

w.rns Relative non-stationarity of wind component w.

uu.nst Non-steadiness of covariance $\overline{u'u'}$.

uv.nst Non-steadiness of covariance $\overline{u'v'}$.

uw.nst Non-steadiness of covariance $\overline{u'w'}$.

vv.nst Non-steadiness of covariance $\overline{v'v'}$.

vw.nst Non-steadiness of covariance $\overline{v'w'}$.

ww.nst Non-steadiness of covariance $\overline{w'w'}$.

ut.nst Non-steadiness of covariance $\overline{u't'}$.

vt.nst Non-steadiness of covariance $\overline{v't'}$.

wt.nst Non-steadiness of covariance $\overline{w't'}$.

q.avg Mean molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

q.min Minimum molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

q.max Maximum molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

qq Variance $\overline{q'q'}$ of molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

uq Covariance $\overline{u'q'}$ between u wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

vq Covariance $\overline{v'q'}$ between v wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

wq Covariance $\overline{w'q'}$ between w wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

uq.nst Non-steadiness of covariance $\overline{u'q'}$ (present only if water concentration is in corresponding raw data file).

vq.nst Non-steadiness of covariance $\overline{v'q'}$ (present only if water concentration is in corresponding raw data file).

wq.nst Non-steadiness of covariance $\overline{w'q'}$ (present only if water concentration is in corresponding raw data file).

c.avg Mean molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.min Minimum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.max Maximum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

cc Variance $\overline{c'c'}$ of molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc Covariance $\overline{u'c'}$ between u wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

vc Covariance $\overline{v'c'}$ between v wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

wc Covariance $\overline{w'c'}$ between w wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

vc.nst Non-steadiness of covariance $\overline{v'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

wc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

a.avg Mean molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.min Minimum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.max Maximum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

aa Variance $\overline{a'a'}$ of molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua Covariance $\overline{u'a'}$ between u wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

va Covariance $\overline{v'a'}$ between v wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

wa Covariance $\overline{w'a'}$ between w wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua.nst Non-steadiness of covariance $\overline{u'a'}$ (present only if ammonia concentration is in corresponding raw data file).

va.nst Non-steadiness of covariance $\overline{v'a'}$ (present only if ammonia concentration is in corresponding raw data file).

wa.nst Non-steadiness of covariance $\overline{w'a'}$ (present only if ammonia concentration is in corresponding raw data file).

m.avg Mean molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.min Minimum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.max Maximum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um Variance $\overline{u'm'}$ of molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

vm Covariance $\overline{v'm'}$ between v wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

wm Covariance $\overline{w'm'}$ between w wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um.nst Non-steadiness of covariance $\overline{u'm'}$ (present only if methane concentration is in corresponding raw data file).

vm.nst Non-steadiness of covariance $\overline{v'm'}$ (present only if methane concentration is in corresponding raw data file).

wm.nst Non-steadiness of covariance $\overline{w'm'}$ (present only if methane concentration is in corresponding raw data file).

temp.avg Mean temperature in Celsius degrees (present only if thermometer data is present in corresponding raw data file).

hrel.avg Mean relative humidity in % (present only if hygrometer data is present in corresponding raw data file).

u.trend.slope Slope of linear trend of wind component u (present only if `trend.removal` is “linear”).

u.trend.constant Intercept of linear trend of wind component u (present only if `trend.removal` is “linear”).

v.trend.slope Slope of linear trend of wind component v (present only if `trend.removal` is “linear”).

v.trend.constant Intercept of linear trend of wind component v (present only if `trend.removal` is “linear”).

w.trend.slope Slope of linear trend of wind component w (present only if `trend.removal` is “linear”).

w.trend.constant Intercept of linear trend of wind component w (present only if `trend.removal` is “linear”).

t.trend.slope Slope of linear trend of sonic temperature (present only if `trend.removal` is “linear”).

t.trend.constant Slope of linear trend of sonic temperature (present only if `trend.removal` is “linear”).

q.trend.slope Slope of linear trend of water vapor (present only if `trend.removal` is “linear” and water is present).

q.trend.constant Slope of linear trend of water vapor (present only if `trend.removal` is “linear” and water is present).

c.trend.slope Slope of linear trend of carbon dioxide (present only if `trend.removal` is “linear” and carbon dioxide is present).

c.trend.constant Slope of linear trend of carbon dioxide (present only if `trend.removal` is “linear” and carbon dioxide is present).

a.trend.slope Slope of linear trend of ammonia (present only if `trend.removal` is “linear” and ammonia is present).

a.trend.constant Slope of linear trend of ammonia (present only if `trend.removal` is “linear” and ammonia is present).

m.trend.slope Slope of linear trend of methane (present only if `trend.removal` is “linear” and methane is present).

m.trend.constant Slope of linear trend of methane (present only if `trend.removal` is “linear” and methane is present).

temp.trend.slope Slope of linear trend of temperature (present only if `trend.removal` is “linear” and temperature is present).

temp.trend.constant Slope of linear trend of temperature (present only if `trend.removal` is “linear” and temperature is present).

hrel.trend.slope Slope of linear trend of relative humidity (present only if `trend.removal` is “linear” and relative humidity is present).

hrel.trend.constant Slope of linear trend of relative humidity (present only if `trend.removal` is “linear” and relative humidity is present).

u.spike.count Number of spikes detected averaging wind component along axis x (present only if spike.treatment— is `set.na`).

v.spike.count Number of spikes detected averaging wind component along axis y (present only if spike.treatment— is `set.na`).

w.spike.count Number of spikes detected averaging wind component along axis z (present only if spike.treatment— is `set.na`).

t.spike.count Number of spikes detected averaging sonic temperature (present only if spike.treatment— is `set.na`).

q.spike.count Number of spikes detected averaging water concentration (present only if spike.treatment— is `set.na` and water is present).

c.spike.count Number of spikes detected averaging carbon dioxide concentration (present only if spike.treatment— is `set.na` and carbon dioxide is present).

a.spike.count Number of spikes detected averaging ammonia concentration (present only if spike.treatment— is `set.na` and ammonia is present).

m.spike.count Number of spikes detected averaging methane concentration (present only if spike.treatment— is `set.na` and methane is present).

temp.spike.count Number of spikes detected averaging temperature concentration (present only if spike.treatment— is `set.na` and temperature is present).

hrel.spike.count Number of spikes detected averaging relative humidity concentration (present only if spike.treatment— is `set.na` and relative humidity is present).

3.4 The object type “sonic.eddy.cov.data”

Objects of type `sonic.eddy.cov.data` are created upon application of eddy covariance or planar fit. A `sonic.eddy.cov.data` object is a list, containing the following items:

data Data frame, containing results of eddy covariance processing.

delay Value of parameter `delay` as passed in call, for future reference.

sampling.rate Nominal sampling rate, as coming from the data object `sonic.raw.data` from which averages have been computed (see 6.4).

averaging.time Averaging time used when processing data.

spike Real number representing the spike detection threshold expressed in terms of sigmas and common to all data columns in SonicLib standard raw files; or, element of type `spike.detection.spec`.

spike.treatment String containing the name of spike treatment method used.

station.altitude Height above mean sea level assumed for station (m).

weibull.alpha Shape parameter of Weibull distribution better fitting the data set.

weibull.beta Scale parameter of Weibull distribution better fitting the data set.

weibull.statistic Kolmogorov-Smirnov statistic corresponding to the Weibull distribution better fitting the data set.

weibull.probability Probability that the data set is distributed according to the better fitting Weibull distribution.

lognormal.mean Log-normal mean of lognormal distribution better fitting the data set.

lognormal.sd Log-normal standard deviation of lognormal distribution better fitting the data set.

lognormal.statistic Kolmogorov-Smirnov statistic corresponding to the log-normal distribution better fitting the data set.

lognormal.probability Probability that the data set is distributed according to the better fitting lognormal distribution.

processing A string, whose value may be "eddy.covariance", "eddy.covariance.3" or "planar.fit".

The list item "data" is a superset of the homologous element found in `sonic.avg.data`. The additional columns contain information specifically constructed during eddy covariance processing. The columns in "data" are:

t.stamp Time stamp identifying the beginning of averaging interval.

n.data Number of data in averaging interval (should be indicatively equal to $f_s \cdot t_a$, the averaging time. It may differ a little bit, due to clock imprecision in sonic or data acquisition system (or both). If much smaller than expected value, then it is legitimate to suspect a gap in data (most often due to power failure, data acquisition system failure, or ultrasonic anemometer lock).

u.avg Mean value of wind component u, in m/s, computed on valid data.

u.min Minimum value of wind component u, in m/s, computed on valid data.

u.max Maximum value of wind component u, in m/s, computed on valid data.

v.avg Mean value of wind component v, in m/s, computed on valid data.

v.min Minimum value of wind component v, in m/s, computed on valid data.

v.max Maximum value of wind component v, in m/s, computed on valid data.

w.avg Mean value of wind component w, in m/s, computed on valid data.

w.min Minimum value of wind component w, in m/s, computed on valid data.

w.max Maximum value of wind component w, in m/s, computed on valid data.

t.avg Mean value of sonic temperature, in Celsius degrees, computed on valid data.

t.min Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

t.max Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

uu Covariance between wind components u and u (that is, variance of u) in m^2/s^2.

uv Covariance between wind components u and v in m^2/s^2.

uw Covariance between wind components u and w in m^2/s^2.

vv Covariance between wind components v and v (that is, variance of v) in m^2/s^2.

vw Covariance between wind components v and w in m^2/s^2.

ww Covariance between wind components w and w (that is, variance of w) in m^2/s^2.

ut Covariance between wind components u and sonic temperature in m C/s^2.

vt Covariance between wind components v and sonic temperature in m C/s^2.

wt Covariance between wind components w and sonic temperature in m C/s^2.

vel Mean horizontal wind speed according to instrument reference in m/s.

vel.max Maximum horizontal wind speed according to instrument reference in m/s.

resultant.vel Resultant horizontal wind speed according to instrument reference in m/s.

vel.sd Standard deviation of horizontal wind speed according to instrument reference in m/s.

vel3 Mean of wind speed elementary data in the instrument's horizontal plane, in m^3/s^3.

vel3.3 Mean of wind speed elementary data using all three wind vector components, in m^3/s^3.

u.j Mean of x component of horizontal wind versor, $\frac{u}{\sqrt{u^2+v^2}}$ (non-dimensional).

v.j Mean of y component of horizontal wind versor, $\frac{v}{\sqrt{u^2+v^2}}$ (non-dimensional).

dir.sd Yamartino estimate of standard deviation of wind direction in horizontal plane (degrees).

u.rns Relative non-stationarity of wind component u.

v.rns Relative non-stationarity of wind component v.

w.rns Relative non-stationarity of wind component w.

uu.nst Non-steadiness of covariance $\overline{u'u'}$.

uv.nst Non-steadiness of covariance $\overline{u'v'}$.

uw.nst Non-steadiness of covariance $\overline{u'w'}$.

vv.nst Non-steadiness of covariance $\overline{v'v'}$.

vw.nst Non-steadiness of covariance $\overline{v'w'}$.

ww.nst Non-steadiness of covariance $\overline{w'w'}$.

ut.nst Non-steadiness of covariance $\overline{u't'}$.

vt.nst Non-steadiness of covariance $\overline{v't'}$.

wt.nst Non-steadiness of covariance $\overline{w't'}$.

q.avg Mean molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

q.min Minimum molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

q.max Maximum molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

qq Variance $\overline{q'q'}$ of molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

uq Covariance $\overline{u'q'}$ between u wind component and molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

vq Covariance $\overline{v'q'}$ between v wind component and molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

wq Covariance $\overline{w'q'}$ between w wind component and molar concentration of water in $mmol/m^3$ (present only if water concentration is in corresponding raw data file).

uq.nst Non-steadiness of covariance $\overline{u'q'}$ (present only if water concentration is in corresponding raw data file).

vq.nst Non-steadiness of covariance $\overline{v'q'}$ (present only if water concentration is in corresponding raw data file).

wq.nst Non-steadiness of covariance $\overline{w'q'}$ (present only if water concentration is in corresponding raw data file).

c.avg Mean molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.min Minimum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.max Maximum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

cc Variance $\overline{c'c'}$ of molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc Covariance $\overline{u'c'}$ between u wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

vc Covariance $\overline{v'c'}$ between v wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

wc Covariance $\overline{w'c'}$ between w wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

vc.nst Non-steadiness of covariance $\overline{v'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

wc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

a.avg Mean molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.min Minimum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.max Maximum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

aa Variance $\overline{a'a'}$ of molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua Covariance $\overline{u'a'}$ between u wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

va Covariance $\overline{v'a'}$ between v wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

wa Covariance $\overline{w'a'}$ between w wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua.nst Non-steadiness of covariance $\overline{u'a'}$ (present only if ammonia concentration is in corresponding raw data file).

va.nst Non-steadiness of covariance $\overline{v'a'}$ (present only if ammonia concentration is in corresponding raw data file).

wa.nst Non-steadiness of covariance $\overline{w'a'}$ (present only if ammonia concentration is in corresponding raw data file).

m.avg Mean molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.min Minimum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.max Maximum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um Variance $\overline{u'm'}$ of molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

vm Covariance $\overline{v'm'}$ between v wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

wm Covariance $\overline{w'm'}$ between w wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um.nst Non-steadiness of covariance $\overline{u'm'}$ (present only if methane concentration is in corresponding raw data file).

vm.nst Non-steadiness of covariance $\overline{v'm'}$ (present only if methane concentration is in corresponding raw data file).

wm.nst Non-steadiness of covariance $\overline{w'm'}$ (present only if methane concentration is in corresponding raw data file).

temp.avg Mean temperature in Celsius degrees (present only if thermometer data is present in corresponding raw data file).

hrel.avg Mean relative humidity in % (present only if hygrometer data is present in corresponding raw data file).

u.trend.slope Slope of linear trend of wind component u (present only if `trend.removal` is “linear”).

u.trend.constant Intercept of linear trend of wind component u (present only if `trend.removal` is “linear”).

v.trend.slope Slope of linear trend of wind component v (present only if `trend.removal` is “linear”).

v.trend.constant Intercept of linear trend of wind component v (present only if `trend.removal` is “linear”).

w.trend.slope Slope of linear trend of wind component w (present only if `trend.removal` is “linear”).

w.trend.constant Intercept of linear trend of wind component w (present only if `trend.removal` is “linear”).

t.trend.slope Slope of linear trend of sonic temperature (present only if `trend.removal` is “linear”).

t.trend.constant Slope of linear trend of sonic temperature (present only if `trend.removal` is “linear”).

q.trend.slope Slope of linear trend of water vapor (present only if `trend.removal` is “linear” and water is present).

q.trend.constant Slope of linear trend of water vapor (present only if `trend.removal` is “linear” and water is present).

c.trend.slope Slope of linear trend of carbon dioxide (present only if `trend.removal` is “linear” and carbon dioxide is present).

c.trend.constant Slope of linear trend of carbon dioxide (present only if `trend.removal` is “linear” and carbon dioxide is present).

a.trend.slope Slope of linear trend of ammonia (present only if `trend.removal` is “linear” and ammonia is present).

a.trend.constant Slope of linear trend of ammonia (present only if `trend.removal` is “linear” and ammonia is present).

m.trend.slope Slope of linear trend of methane (present only if `trend.removal` is “linear” and methane is present).

m.trend.constant Slope of linear trend of methane (present only if `trend.removal` is “linear” and methane is present).

temp.trend.slope Slope of linear trend of temperature (present only if `trend.removal` is “linear” and temperature is present).

temp.trend.constant Slope of linear trend of temperature (present only if `trend.removal` is “linear” and temperature is present).

hrel.trend.slope Slope of linear trend of relative humidity (present only if `trend.removal` is “linear” and relative humidity is present).

hrel.trend.constant Slope of linear trend of relative humidity (present only if `trend.removal` is “linear” and relative humidity is present).

u.spike.count Number of spikes detected averaging wind component along axis x (present only if spike.treatment— is `set.na`).

v.spike.count Number of spikes detected averaging wind component along axis y (present only if spike.treatment— is `set.na`).

w.spike.count Number of spikes detected averaging wind component along axis z (present only if spike.treatment— is `set.na`).

t.spike.count Number of spikes detected averaging sonic temperature (present only if spike.treatment— is `set.na`).

q.spike.count Number of spikes detected averaging water concentration (present only if spike.treatment— is `set.na` and water is present).

c.spike.count Number of spikes detected averaging carbon dioxide concentration (present only if spike.treatment— is `set.na` and carbon dioxide is present).

a.spike.count Number of spikes detected averaging ammonia concentration (present only if spike.treatment— is `set.na` and ammonia is present).

m.spike.count Number of spikes detected averaging methane concentration (present only if spike.treatment— is `set.na` and methane is present).

temp.spike.count Number of spikes detected averaging temperature concentration (present only if spike.treatment— is `set.na` and temperature is present).

hrel.spike.count Number of spikes detected averaging relative humidity concentration (present only if spike.treatment— is `set.na` and relative humidity is present).

theta First rotation angle (designated as α in description at 8.1), in degrees (this parameter is present only if mode == “eddy.covariance” or “eddy.covariance.3”).

phi Second rotation angle (designated as β in description at 8.1), in degrees (this parameter is present only if mode == “eddy.covariance” or “eddy.covariance.3”).

psi Third rotation angle (designated as δ in description at 8.1), in degrees (this parameter is present only if mode == “eddy.covariance” or “eddy.covariance.3”; also notice its value will be 0 if mode == “eddy.covariance”).

alpha Individual averaging block rotation angle used in planar fit (designated as α in description at 8.1), in degrees (this parameter is present only if mode == “planar.fit”).

beta First fitting plane rotation angle (designated as β in description at 8.1), in degrees (this parameter is present only if mode == “planar.fit”).

gamma Second fitting plane rotation angle (designated as δ in description at 8.1), in degrees (this parameter is present only if mode == “planar.fit”).

u.avg.rot Component u of mean wind in rotated frame (m/s).

v.avg.rot Component v of mean wind in rotated frame (m/s).

w.avg.rot Component w of mean wind in rotated frame (m/s).

Dir Wind direction ($^\circ$ from North, povenance convention).

uu.rot Covariance between u and u wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

uv.rot Covariance between u and v wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

uw.rot Covariance between u and w wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

vv.rot Covariance between v and v wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

vw.rot Covariance between v and w wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

ww.rot Covariance between w and w wind components in rotated frame ($\mathrm{m}^2/\mathrm{s}^2$).

ut.rot Covariance between u wind components in rotated frame and temperature t (m/sK).

vt.rot Covariance between v wind components in rotated frame and temperature t (m/sK).

wt.rot Covariance between w wind components in rotated frame and temperature t (m/sK).

u.star Friction velocity (m/s).

H0.v Buoyancy vertical flux (W/m^2; used as an approximation of the sensible heat flux).

L Obukhov length (m).

z.over.L Stability parameter (adimensional, with reference height assumed to be equal to the anemometer's).

T.star Scale temperature (Celsius degrees).

Rho.air Air density (g/m^3).

Rho.Cp Value of $\rho \cdot C_p$ ($\mathrm{J/m^3 \cdot K}$).

Pa Estimate of local atmospheric pressure, given altitude above mean sea level and temperature (hPa).

wq.rot Covariance between wind component w in rotated frame and molar concentration of water ($\mathrm{m \cdot mmol/s \cdot m^3}$, that is $\mathrm{mmol/s \cdot m^2}$; this field is present only if a water measurement is taken).

c.d Molar concentration of air (mmol/m^3; this field is present only if a water measurement is taken).

q.d Ratio of water molar concentration to air molar concentration (adimensional; this field is present only if a water measurement is taken).

c.dv Molar concentration of moist air (mmol/m^3; this field is present only if a water measurement is taken).

t.c Thermal term in WPL correction (mmol/m$^2 \cdot$ s; this field is present only if a water measurement is taken).

Fq.molar Molar flux of water along the vertical direction (mmol/m$^2 \cdot$ s; this field is present only if a water measurement is taken).

Fq.mass Mass flux of water along the vertical direction (mmol/m$^2 \cdot$ s; this field is present only if a water measurement is taken).

mix.f Conversion factor from (mmol/m^3 to g/g; this field is present only if a water measurement is taken).

wt.cor Value of $\overline{w'T'}$ where T represents the real, not sonic temperature (m $\cdot$ K/s; this field is present only if a water measurement is taken).

H0 Sensible heat flux (W/m^2; this field is present only if a water measurement is taken).

Lambda Latent condensation heat for water (J/g; this field is present only if a water measurement is taken).

He Latent heat flux (W/m^2; this field is present only if a water measurement is taken).

wc.rot Covariance between wind component w in rotated frame and molar concentration of carbon dioxide (m $\cdot$ mmol/s $\cdot$ m^3, that is mmol/s $\cdot$ m^2; this field is present only if a carbon dioxide measurement is taken).

Fc.molar Molar flux of carbon dioxide along the vertical direction (mmol/m$^2 \cdot$s; this field is present only if a carbon dioxide measurement is taken).

Fc.mass Mass flux of carbon dioxide along the vertical direction (mmol/m$^2 \cdot$ s; this field is present only if a carbon dioxide measurement is taken).

wa.rot Covariance between wind component w in rotated frame and molar concentration of ammonia (m $\cdot$ mmol/s $\cdot$ m^3, that is mmol/s $\cdot$ m^2; this field is present only if an ammonia measurement is taken).

wm.rot Covariance between wind component w in rotated frame and molar concentration of methane (m $\cdot$ mmol/s $\cdot$ m^3, that is mmol/s $\cdot$ m^2; this field is present only if a methane measurement is taken).

3.5 The object type "sonic.wind.power.data"

Objects of type `sonic.wind.power.data` are returned by function `wind.power` on successful completion, ane are implemented as lists with the following contents:

data List, containing all time-stamped data hosting most of results (see later).

sampling.rate Nominal sampling rate, as coming from the data object `sonic.raw.data` from which averages have been computed (see 6.4).

averaging.time Averaging time used when processing data.

spike Real number representing the spike detection threshold expressed in terms of sigmas and common to all data columns in SonicLib standard raw files; or, element of type `spike.detection.spec`.

spike.treatment String containing the name of spike treatment method used.

mean.vel.stddev Overall data set mean of wind speed standard deviation (m/s).

turb.coefficient Overall data set turbulence coefficient (adimensional ratio).

cum.vel Cumulative empirical distribution function for wind speed.

processing String, containing the name identifying the processing performed (possible values are "2.rotations" and "3.rotations").

The `data` list contains the following data vectors:

t.stamp Time stamp identifying the beginning of averaging interval.

n.data Number of data in averaging interval (should be indicatively equal to $f_s \cdot t_a$, the averaging time. It may differ a little bit, due to clock imprecision in sonic or data acquisition system (or both). If much smaller than expected value, then it is legitimate to suspect a gap in data (most often due to power failure, data acquisition system failure, or ultrasonic anemometer lock).

u.avg Mean value of wind component u, in m/s, computed on valid data.

u.min Minimum value of wind component u, in m/s, computed on valid data.

u.max Maximum value of wind component u, in m/s, computed on valid data.

v.avg Mean value of wind component v, in m/s, computed on valid data.

v.min Minimum value of wind component v, in m/s, computed on valid data.

v.max Maximum value of wind component v, in m/s, computed on valid data.

w.avg Mean value of wind component w, in m/s, computed on valid data.

w.min Minimum value of wind component w, in m/s, computed on valid data.

w.max Maximum value of wind component w, in m/s, computed on valid data.

t.avg Mean value of sonic temperature, in Celsius degrees, computed on valid data.

t.min Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

t.max Minimum value of sonic temperature, in Celsius degrees, computed on valid data.

uu Covariance between wind components u and u (that is, variance of u) in m^2/s^2.

uv Covariance between wind components u and v in m^2/s^2.

uw Covariance between wind components u and w in m^2/s^2.

vv Covariance between wind components v and v (that is, variance of v) in m^2/s^2.

vw Covariance between wind components v and w in m^2/s^2.

ww Covariance between wind components w and w (that is, variance of w) in m^2/s^2.

ut Covariance between wind components u and sonic temperature in m C/s^2.

vt Covariance between wind components v and sonic temperature in m C/s^2.

wt Covariance between wind components w and sonic temperature in m C/s^2.

vel Mean horizontal wind speed according to instrument reference in m/s.

vel.max Maximum horizontal wind speed according to instrument reference in m/s.

resultant.vel Resultant horizontal wind speed according to instrument reference in m/s.

vel.sd Standard deviation of horizontal wind speed according to instrument reference in m/s.

vel3 Mean of wind speed elementary data in the instrument’s horizontal plane, in m^3/s^3.

vel3.3 Mean of wind speed elementary data using all three wind vector components, in m^3/s^3.

u.j Mean of x component of horizontal wind versor, $\frac{u}{\sqrt{u^2+v^2}}$ (non-dimensional).

v.j Mean of y component of horizontal wind versor, $\frac{v}{\sqrt{u^2+v^2}}$ (non-dimensional).

dir.sd Yamartino estimate of standard deviation of wind direction in horizontal plane (degrees).

u.rns Relative non-stationarity of wind component u.

v.rns Relative non-stationarity of wind component v.

w.rns Relative non-stationarity of wind component w.

uu.nst Non-steadiness of covariance $\overline{u'u'}$.

uv.nst Non-steadiness of covariance $\overline{u'v'}$.

uw.nst Non-steadiness of covariance $\overline{u'w'}$.

vv.nst Non-steadiness of covariance $\overline{v'v'}$.

vw.nst Non-steadiness of covariance $\overline{v'w'}$.

ww.nst Non-steadiness of covariance $\overline{w'w'}$.

ut.nst Non-steadiness of covariance $\overline{u't'}$.

vt.nst Non-steadiness of covariance $\overline{v't'}$.

wt.nst Non-steadiness of covariance $\overline{w't'}$.

q.avg Mean molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

q.min Minimum molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

q.max Maximum molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

qq Variance $\overline{q'q'}$ of molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

uq Covariance $\overline{u'q'}$ between u wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

vq Covariance $\overline{v'q'}$ between v wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

wq Covariance $\overline{w'q'}$ between w wind component and molar concentration of water in mmol/m^3 (present only if water concentration is in corresponding raw data file).

uq.nst Non-steadiness of covariance $\overline{u'q'}$ (present only if water concentration is in corresponding raw data file).

vq.nst Non-steadiness of covariance $\overline{v'q'}$ (present only if water concentration is in corresponding raw data file).

wq.nst Non-steadiness of covariance $\overline{w'q'}$ (present only if water concentration is in corresponding raw data file).

c.avg Mean molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.min Minimum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

c.max Maximum molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

cc Variance $\overline{c'c'}$ of molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc Covariance $\overline{u'c'}$ between u wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

vc Covariance $\overline{v'c'}$ between v wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

wc Covariance $\overline{w'c'}$ between w wind component and molar concentration of carbon dioxide in mmol/m^3 (present only if carbon dioxide concentration is in corresponding raw data file).

uc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

vc.nst Non-steadiness of covariance $\overline{v'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

wc.nst Non-steadiness of covariance $\overline{w'c'}$ (present only if carbon dioxide concentration is in corresponding raw data file).

a.avg Mean molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.min Minimum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

a.max Maximum molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

aa Variance $\overline{a'a'}$ of molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua Covariance $\overline{u'a'}$ between u wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

va Covariance $\overline{v'a'}$ between v wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

wa Covariance $\overline{w'a'}$ between w wind component and molar concentration of ammonia in mmol/m^3 (present only if ammonia concentration is in corresponding raw data file).

ua.nst Non-steadiness of covariance $\overline{u'a'}$ (present only if ammonia concentration is in corresponding raw data file).

va.nst Non-steadiness of covariance $\overline{v'a'}$ (present only if ammonia concentration is in corresponding raw data file).

wa.nst Non-steadiness of covariance $\overline{w'a'}$ (present only if ammonia concentration is in corresponding raw data file).

m.avg Mean molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.min Minimum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

m.max Maximum molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um Variance $\overline{u'm'}$ of molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

vm Covariance $\overline{v'm'}$ between v wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

wm Covariance $\overline{w'm'}$ between w wind component and molar concentration of methane in mmol/m^3 (present only if methane concentration is in corresponding raw data file).

um.nst Non-steadiness of covariance $\overline{u'm'}$ (present only if methane concentration is in corresponding raw data file).

vm.nst Non-steadiness of covariance $\overline{v'm'}$ (present only if methane concentration is in corresponding raw data file).

wm.nst Non-steadiness of covariance $\overline{w'm'}$ (present only if methane concentration is in corresponding raw data file).

temp.avg Mean temperature in Celsius degrees (present only if thermometer data is present in corresponding raw data file).

hrel.avg Mean relative humidity in % (present only if hygrometer data is present in corresponding raw data file).

u.trend.slope Slope of linear trend of wind component u (present only if `trend.removal` is "linear").

u.trend.constant Intercept of linear trend of wind component u (present only if `trend.removal` is "linear").

v.trend.slope Slope of linear trend of wind component v (present only if `trend.removal` is "linear").

v.trend.constant Intercept of linear trend of wind component v (present only if `trend.removal` is "linear").

w.trend.slope Slope of linear trend of wind component w (present only if `trend.removal` is "linear").

w.trend.constant Intercept of linear trend of wind component w (present only if `trend.removal` is "linear").

t.trend.slope Slope of linear trend of sonic temperature (present only if `trend.removal` is "linear").

t.trend.constant Slope of linear trend of sonic temperature (present only if `trend.removal` is "linear").

q.trend.slope Slope of linear trend of water vapor (present only if `trend.removal` is "linear" and water is present).

q.trend.constant Slope of linear trend of water vapor (present only if `trend.removal` is "linear" and water is present).

c.trend.slope Slope of linear trend of carbon dioxide (present only if `trend.removal` is "linear" and carbon dioxide is present).

c.trend.constant Slope of linear trend of carbon dioxide (present only if `trend.removal` is "linear" and carbon dioxide is present).

a.trend.slope Slope of linear trend of ammonia (present only if `trend.removal` is "linear" and ammonia is present).

a.trend.constant Slope of linear trend of ammonia (present only if `trend.removal` is "linear" and ammonia is present).

m.trend.slope Slope of linear trend of methane (present only if `trend.removal` is "linear" and methane is present).

m.trend.constant Slope of linear trend of methane (present only if `trend.removal` is "linear" and methane is present).

temp.trend.slope Slope of linear trend of temperature (present only if `trend.removal` is "linear" and temperature is present).

temp.trend.constant Slope of linear trend of temperature (present only if `trend.removal` is "linear" and temperature is present).

hrel.trend.slope Slope of linear trend of relative humidity (present only if `trend.removal` is "linear" and relative humidity is present).

hrel.trend.constant Slope of linear trend of relative humidity (present only if `trend.removal` is "linear" and relative humidity is present).

u.spike.count Number of spikes detected averaging wind component along axis x (present only if spike.treatment— is `set.na`).

v.spike.count Number of spikes detected averaging wind component along axis y (present only if spike.treatment— is `set.na`).

w.spike.count Number of spikes detected averaging wind component along axis z (present only if spike.treatment— is `set.na`).

t.spike.count Number of spikes detected averaging sonic temperature (present only if spike.treatment— is `set.na`).

q.spike.count Number of spikes detected averaging water concentration (present only if spike.treatment— is `set.na` and water is present).

c.spike.count Number of spikes detected averaging carbon dioxide concentration (present only if spike.treatment— is `set.na` and carbon dioxide is present).

a.spike.count Number of spikes detected averaging ammonia concentration (present only if spike.treatment— is `set.na` and ammonia is present).

m.spike.count Number of spikes detected averaging methane concentration (present only if spike.treatment— is `set.na` and methane is present).

temp.spike.count Number of spikes detected averaging temperature concentration (present only if spike.treatment— is `set.na` and temperature is present).

hrel.spike.count Number of spikes detected averaging relative humidity concentration (present only if spike.treatment— is `set.na` and relative humidity is present).

theta Angle of first rotation (around z axis) (degrees).

phi Angle of second rotation (around transformed y axis) (degrees).

psi Angle of third rotation (around transformed axis z) (degrees).

Dir Wind provenance direction (degrees from North, increasing clockwise).

u.star Friction velocity (m/s).

T Turbulence intensity (adimensional).

ang.Phi Wind vector angle to the horizontal (degrees).

mean.power.coef Mean power coefficient, computed on horizontal data after rotation has been performed (m^3/s^3).

mean.power.coef.3 Mean power coefficient, computed on the whole three-dimensional vector (m^3/s^3).

tot.power.coef Total power coefficient, computed on horizontal data after rotation has been performed (m^3/s^3).

tot.power.coef.3 Total power coefficient, computed on the whole three-dimensional vector (m^3/s^3).

3.6 The object type "dir.stat"

Objects of type `dir.stat` are returned as result of the omonymous function `dir.stat` upon successful completion, and are implemented as lists, with the following components:

dir Vector of original data directions (degrees).

data Vector of original data values (unit depends on which value has been selected).

n.class Number of direction classes (no unit).

c.dir Vector of center directions corresponding to the direction classes (degrees).

conf String, containing the name of the algorithm used to compute confidence limits (to date "boot" is the only supported option).

mean Vector containing the mean values within each sector (unit depends on which value has been selected).

std.dev Vector containing the standard deviation of values within each class (unit depends on which value has been selected).

min Vector containing sector minima (unit depends on which value has been selected).

p.25 Vector containing sector 25-th percentiles (unit depends on which value has been selected).

median Vector containing sector median (unit depends on which value has been selected).

p.75 Vector containing sector 75-th percentiles (unit depends on which value has been selected).

max Vector containing sector maxima (unit depends on which value has been selected).

conf.min Vector containing confidence interval minima for each sector (unit depends on which value has been selected).

conf.max Vector Containing confidence interval maxima for each sector (unit depends on which value has been selected).

Chapter 4

Data Sets

4.1 CRA.01

4.1.1 Data collection site

4.1.2 Meteorological context of data set period

4.1.3 Instrument configuration

Data in CRA.01 set have been collected using a Metek USA-1 ultrasonic anemometer with analog input option and a LiCor LI-7500 open-path water and carbon dioxide sensor.

The two output channels of the LI-7500 have been so configured:

Channel 1 Water. Low voltage (0V) corresponding to 0 mmol/m^3, and high (5V) to 1000 mmol/m^3.

Channel 2 Carbon dioxide. Low voltage (0V) corresponding to 10 mmol/m^3, and high (5V) to 20 mmol/m^3.

Channels 1 and 2 of the LI-7500 have then been connected to the USA-1 input channels 7 and 8 respectively.

The specific USA-1 used in data collection is equipped with a 16 bit analog-to-digital converter. Input voltages, in range -10 to +10V, are encoded to count values -32768 and 32767, reported as integer values. Range 0 to 5V encodes then to range 0 to 16384 counts.

The correspondence between count values and gas concentrations is expressed through a linear relation taking the form

$$g = m \cdot C + q \tag{4.1}$$

where m designates the multiplier, q the offset, C count readout from USA-1, and g gas molar concentration.

Computing the values of m and q for water and carbon dioxide can be made using known information and assuming linear conversion as from (4.1). In particular:

Water q [mmol/m^3] $= C \cdot (32767/10) \cdot (1000/5) = 0.0610370C$. That is, $m = 0.0610370$ and $q = 0$,

Carbon dioxide $c\,[\mathrm{mmol/m^3}] = C\cdot(32767/10)\cdot(20-10)/5+10 = 0.000610370C+10$. That is, $m = 0.000610370$ and $q = 10$,

LI-7500 analog output delay can be programmed, and users did select in this case the value 0.309s for each channel. The USA-1 has been operated at 20Hz, with automatic averaging every two samples, with an output dara rate hence equal to 10Hz. The delay selected then rounds effectively to 0.3s, that is, a three data packets.

(We notice incidentally that programming the LI-7500 analog outputs to tighten output band is common practice, since adopting blindly the instrument's factory defaults would resule in excessively large conversion interval and, as a consequence, in massive information loss. At many sites this loss is sufficient to devoid eddy covariance calculations of water and carbon dioxide fluxes of any physical meanings.)

4.1.4 Data preparation

Original data have been converted to SonicLib standard format in a two-stage process,

4.1.5 Where to find prepared data

4.2 UNIMI.Short.Mast and UNIMI.Long.Mast

4.2.1 Data collection site

4.2.2 Meteorological context

4.2.3 Instrument configuration

4.2.4 Data preparation

4.2.5 Where to find prepared data

4.3 AMERIFLUX gold standard files, open-path part

4.3.1 Data collection site

4.3.2 Meteorological context

4.3.3 Instrument configuration

4.3.4 Data preparation

4.3.5 Where to find prepared data

Chapter 5

Converters

5.1 Converting from SLT form

SLT is a commonly used format used in `EddySoft` program and its `eddymeas` data acquisition system. Actually two variants of SLT exist, depending on whether data have been gathered by direct acquisition (`eddymeas`) or by conversion from other formats.

SLT is a *binary* format, with data encoded as integer numbers. The actual details of encoding is slightly different in the two variants, as shown by prof. W. Eugster in his `readslt.R` program.

The SLT to SonicLib converter follows very closely prof. Eugster's implementation, with output file production instead of generation of an R data frame. The decision to use Fortran instead of R is mainly due to efficiency considerations, in the prospective of converting large bulks of existing data.

The converter is named `readslt` in UNIX and Linux (or `readslt.exe` under Windows). To use it you should open a terminal session under your operating system. Under UNIX and Linux the command line is

```
./readslt <InputPrefix> <DateTime> <Mode> <Convert> <OutputPath>
```

where `<InputPrefix>` is the name of the path containing data, followed by the first letters (the alphabetic, not numerical, ones) in data files you're interested in (this is necessary, as the first letters in file name are to some extent freely assigned).

`<DateTime>`, as its name suggests, designates the date and time of the data you like to convert, in ISO form ("`YYYY-MM-DD hh:00:00`"), with minute and seconds forced to zero. This, because we're interested in a specific *hour* in SonicLib, while SLT files are often splitted in fractions of a hour. This argument must be enclused in double quotes, as it contains a space: without quotes, the command line interpreter would split this single argument in two.

`<Mode>` is an alphanumeric string, whose values can only be `eddysoft` and `eddymeas`, specifying any of the two available forms.

Parameter `<Convert>` is the name of a file containing conversion specifications for all data in columns 5 to last, containing analog values. Something more further on.

The last parameter, `<OutputPath>`, contains the name of the directory where the SonicLib file resulting from conversion will be written to.

5.1.1 The SLT conversion specification file

One hard problem of SLT files is they encode data in integer form, and this includes analog data. SonicLib files, on the contrary, contain floating point data.

5.2 Converting from Meteoflux Core V1.0

5.3 Converting from Meteoflux Core V2.0

5.4 Converting from old Meteoflux V3.x

5.5 Converting from LabVIEW-made data acquisition systems

5.5.1 Little- and high-endianness

5.5.2 Endianness in LabVIEW is fixed

5.5.3 Long (32 bit) and short (16 bit) data forms from Servizi Territorio

5.6 Converting from GRIFO form

Chapter 6

Data Gathering and Preparation

This chapter presents some functions whose use allows accessing data in standard format, and their preparation for further processing.

Individual functions are described in sections of this chapter. Each function-specific description follows the same form,

6.1 Read Single Standard Ultrasonic Anemometer Raw Data File

6.1.1 Function Use

To read a single standard ultrasonic anemometer data file use function

```
get.raw.data(file.name, sampling.rate=10,
     threshold=0.005, verbose=FALSE)
```

where

file.name is the name of file to read, according to the specification given in section 3.1.

sampling.rate is an integer number indicating the sampling rate f_s in samples per second adopted when collecting all data in file, according to user knowledge. This value is used to compute the expected number of data in file as $n_s = 3600 \cdot f_s$; the expected number of data is compared with the number of data actually present in file, n and if the relative difference $(n - n_s)/n_s$ is smaller than a threshold value (see next parameter) the data set is refused. Default = 10.

threshold Value of the relative difference $(n - n_s)/n_s$ between actual and expected number of data below which data set is refused. Default = 0.005.

verbose Boolean flag. If TRUE, error and progress message are printed to screen (most useful in interactive use). If FALSE, no message is printed and the success can be assessed only by inspecting output value.

The routine yields a result which on error is NULL (an error message is printed in case only if “verbose” parameter is TRUE). If data read has been completed successfully result is an object of type “sonic.raw.data”, described in 3.1.

6.1.2 Examples

In our first example a data file is read and its w component estimated density plotted to graphic window. A normal density of equal parameters is also plotted as a dashed line, and the result presented in figure 6.1.

```
> d <- get.raw.data("../TestData/20100516.10.csv", verbose=TRUE)
>
> plot(density(d$data$w, na.rm=TRUE),
      xlab="w (m/s)", ylab="Density", main="")
>
> # Build a normal density with same mean
> # and standard deviation and add to plot
> mu <- mean(d$data$w, na.rm=TRUE)
> sigma <- sqrt(var(d$data$w, na.rm=TRUE))
> w.min <- min(d$data$w, na.rm=TRUE)
> w.max <- max(d$data$w, na.rm=TRUE)
> x <- seq(from=w.min, to=w.max, length.out=128)
> lines(x,dnorm(x,mu,sigma),lty="dashed")
```

Quantifying data availability in a single file is an often-performed task, to which no SonicLib function has been devoted. The reason is, deciding whether a data item is valid (and then contributing to availability) or not includes a bit of subjectivity. And more important, testing availability within a hourly file is usually very simple, and may be accomplished by directly using R capabilities.

In the next example availability is checked by counting how many data have been labeled “valid” by the ultrasonic anemometer, and comparing the result with the total number of (valid and invalid) data lines. The result is expressed in percent form.

```
> d <- get.raw.data("../TestData/20100516.10.csv", verbose=TRUE)
> 100 * sum(!is.na(d$data$w)) / length(d$data$w)
  [1] 100
```

100% of data availability within a single hour is quite common with ultrasonic anemometers, thanks to their immunity to wind calms.

6.1.3 Notes

- If optional (e.g. water or carbon dioxide) columns are present they are read with no applied time shift, so that it may be estimated a posteriori if necessary.
- All data read are not scaled, all changes necessary to transform from original instrument files and sonic standard raw files having been made previously by adapter procedures, fed with appropriate conversion parameters.

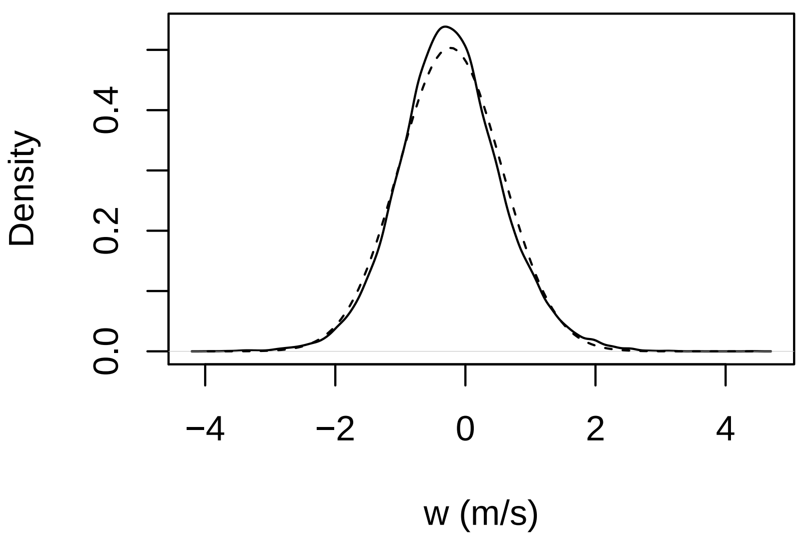


Figure 6.1: Result from first example; normal density (dashed line) shown for reference

6.2 Read Multiple Standard Ultrasonic Anemometer Raw Data Files

6.2.1 Function Use

To read a single standard ultrasonic anemometer data file use function

```
get.multi.raw.data(file.name, n.hours=1, sampling.rate=10,
     threshold=0.005, average.by="none", verbose=FALSE)
```

where

file.name is the name of first file to read, according to the specification given in section 3.1.

n.hours is the number of hours to stich together (default: 1).

sampling.rate is an integer number indicating the sampling rate f_s in samples per second adopted when collecting all data in file, according to user knowledge. This value is used to compute the expected number of data in file as $n_s = 3600 \cdot f_s$; the expected number of data is compared with the number of data actually present in file, n and if the relative difference $(n - n_s)/n_s$ is smaller than a threshold value (see next parameter) the data set is refused. Default = 10.

threshold Value of the relative difference $(n - n_s)/n_s$ between actual and expected number of data below which data set is refused. Default = 0.005.

average.by String, whose allowed values are “none” (default), “seconds” and “minutes”; if “none”, data are gathered as they are, without any averaging; if “seconds”, data are averaged on one second basis; if “minutes”, data are averaged on 1 minute basis.

verbose Boolean flag. If TRUE, error and progress message are printed to screen (most useful in interactive use). If FALSE, no message is printed and the success can be assessed only by inspecting output value.

The routine yields a result which on error is NULL (an error message is printed in case only if “verbose” parameter is TRUE). If data read has been completed successfully result is an object of type “sonic.raw.data”, described in 3.1.

6.2.2 Examples

Function `get.multi.raw.data` is typically used in preparation of spectral analysis, when multi-hour files are to be packed together in order to gain band on the left side of log-log diagrams customarily used in micro-meteorology.

The following example shows an example of such a preparation (without the actual spectral part).

```
> d <- get.multi.raw.data("../TestData/20100516.00.csv", n.hours=4, verbose=TRUE)
```

Proceding this way, and accepting all default, we requested in fact to get four data files and stitch them one after the other, without any averaging (the default value of `average.by` is `"none"`). Operating at 10 Hz this will produce a data set whose length will be around $36000 \cdot 4 = 144000$, quite huge for even the fastest FFT (which if data size is not a power of 2 may degrade performance from order $n \cdot \log n$ to n^2. More important, the result will be very noisy as information from many frequency scales is all mixed together.

A sensible approach might be to average data every seconds, improving the overall appearance of the low frequency part of spectral diagram, but losing an order of magnitude on the Nyquist frequency (from 5 to 0.5 Hz). The call with per-second averaging is:

```
> d <- get.multi.raw.data("../TestData/20100516.00.csv", n.hours=4, average.by="seconds", verbo
```

6.2.3 Notes

Function `get.multi.raw.data` is designed to stitch together a *reasonable* number of data files, say from 1 to 8. Averaging is done after all raw data are read and stitched together, even if per-second or per-minute averaging is made. This will result in significant memory occupation if the number of hours to get is high.

For this function to complete successfully it is required that all individual data files exist, and contain at least $100 \cdot (1 - \text{threshold})$ percent of data. This may seem restrictive (at least if the small default threshold value is retained), but it helps spectral calculations, for which the function has mostly have been written: allowing a high number of gaps would in case force to employ costly periodogram-based algorithms instead of lightweight FFTs. Of course, the authors opinion is just an opinion, and you, the user, are entirely free of choosing your sensible threshold value: you go, gal - at your own risk.

6.3 Extract Subset from a Standard Ultrasonic Anemometer Data Set

6.4 Average Data from a Single Raw Data File

6.4.1 Function Use

To average data from a single raw data file use this function:

```
average.sonic.data(
  d,
  initial.stamp,
  averaging.time=30,
  delay=0.3,
  trend.removal="none",
  spike.detection.threshold=3,
  spike.treatment="set.na",
  verbose=FALSE
)
```

where

d is an object of type `sonic.raw.data` on which averages are taken.

initial.stamp is a POSIXct date and time associated to the beginning instant the data in file have been gathered; this date and time is not arbitrary, and may be generated from file name *in normal cases.* In some circumstances, however, it may be useful to shift time forward or backward by some fixed time, for example to adjust a systematic error of data acquisition system clock. By the way: the POSIXct date and time corresponding to file name may be obtained through function `time.stamp.from.name`, whose use is described later on as an example.

averaging.time is a positive integer t_a, whose admissible values are 10, 15, 20, 30, 60, indicating the averaging time to be used. Default: 30.

delay A real number indicating a unique delay value to applied to any optional columns simultaneously, or an object of class "delay.spec" indicating the delay used for any possible optional column. Objects of class "delay.spec" are created using function `set.delay`, whose use is described in the examples. Default: 0.3 s.

trend.removal is a string specifying the trend removal algorithm to use. Possible values are "none" (no trend removal is performed), "linear" (trend is estimated by linear least squares regression and removed in mean-preserving fashion). The default is "none".

spike.detection.threshold A real number indicating a unique spike detection threshold to be used for all channels, or a structure of type `spike.detection.threshold` containing the individual thresholds for each channel. All spike detection threshold values are positive numbers meaning a iven number of standard deviations. For example, a value of 2 means "two standard deviations". Default: 3.

spike.treatment String, meaning the type of processing to be made on spikes. The possible choices are "none" (no spike processing made, in which case parameter `spike.detection.threshold` is ignored), and "set.na" (spikes are replaced with NA values). Default: "set.na".

verbose A boolean flag indicating whether messages are to be printed (TRUE) or not (FALSE).

Function `average.sonic.data` returns on exit a NULL value in case of failure, or an object of class `sonic.avg.data`, described in section 3.3.

6.4.2 Time stamps: needed or not in data files?

Section 3.3 lists `time.stamp` among the "mandatory" columns. But a word of caution is necessary here, as things are somewhat less dramatic.

In fact: section 3.3 deals with the SonicLib raw data files *as they look after they have been read into memory.* This is not exactly the same as how the very same data appear *on disk.*

To date the only difference between SonicLib raw data files on disk and their memory representation is the time stamp, which may be missing from disk files.

In this case, the time stamp is inferred from data using the rule

$$t = 3600 \cdot \frac{i-1}{n} \tag{6.1}$$

where i is the data item index and n the total number of data items in file. You may notice from 6.1 that time stamp is stored as a number in interval $0 \leq t < 3600$.

6.4.3 Wind speed: many definitions

If the concept of wind velocity vector is quite intuitive and inherently unambiguous, the same can not be said of wind *speed.* The reason is, no unique definition exists: a good reason to make a bit of order.

The *three-dimensional speed* is the length of instantaneous wind vector, $U_3 = \sqrt{u^2+v^2+w^2}$. Its mean value, $\overline{U_3}$, represents the full three-dimensional wind speed.

The *horizontal* wind speed is defined in a similar manner than three-dimensional wind, with the two horizontal dimensions only, $U = \sqrt{u^2+v^2}$. Its mean represents the *scalar velocity*, or wind speed *tout-court.* And here comes the first ambiguity: what horizontal exactly means? In principle the horizontal plane should be the plane perpendicular to local vertical. But in practice it coincides with the *instrumental* horizontal. Instrumental and local horizontal should ideally coincide, but inevitable small errors of positioning prevent this identification to be absolute.

A third type of speed is in common use: *resultant speed.* This is defined as $U_r = \sqrt{\overline{u}^2+\overline{v}^2}$, that is, the length of average horizontal vector. Because of this definition resultant speed can not be larger than scalar speed. The ratio $\frac{U_r}{U}$ is always between 0 and 1, the former value being reached when wind sweeps all possible direction with even intensities, and the latter corresponding to a perfectly uni-directional wind.

6.4.4 Standard deviation of the wind direction

The problem of computing the standard deviation of wind direction is not completely defined, as measuring the deviation of an instantaneous direction angle to a mean direction is inherently ambiguous given the modular nature of angles.

Then, instead of computing the standard deviation directly, in literature many estimators have been proposed for estimating it. Among them, Yamartino method is widely acknowledged to provide the better results in a variety of conditions.

Yamartino estimate starts from the instantaneous horizontal wind versors, defined ad

$$\mathbf{j} = [u_j, v_j] \tag{6.2}$$

where

$$u_j = \frac{u}{\sqrt{u^2+v^2}} \tag{6.3}$$

and

$$v_j = \frac{v}{\sqrt{u^2 + v^2}} \tag{6.4}$$

with u and v the two horizontal wind components along directions x and y.

Once the instantaneous versors have been computed, their components average is taken, $\overline{u_j}$ and $\overline{v_j}$, and the quantity

$$\varepsilon = \sqrt{1 - \left(\overline{u_j}^2 + \overline{v_j}^2\right)} \tag{6.5}$$

is then computed. The standard deviation of wind direction is finally estimated as

$$\hat{\sigma_A} = \left(1 + \alpha\varepsilon^3\right) \cdot \arcsin\varepsilon \tag{6.6}$$

where $\alpha = \left(\frac{2}{\sqrt{3}} - 1\right)$.

6.4.5 Trend removal

One common manifestation on non-stationarity in data is the occurrence of a slow variation in data, as if their mean value drifts following some smooth function of time.

This slow variation is visible within any averaging interval, and may be approximated by a linear function.

As we're using time averages and covariances within each averaging block, if this slow variation is not removed its value will leak into covariances, and appear as a fictive turbulent flux.

It is worth noticing the adverse effects of linear trend are a direct consequence of the way we compute the means and covariances as basically time averages. For them to make sense in the context of data processing of sonic anemometer measurements it is necessary that time averages and ensemble means coincide, which is possible only if we assume stationarity in the process underlying data, and in data themselves. The presence of a linear trend directly invalidates this assumption and, with it, any further serious consideration.

The problem then arises, on how to remove linear trends from any averaging block in sonic data series.

One solution assumes data are positioned in time with sufficient precision. Then, we may take all couples (t_i, s_i), where s indicates signal values and t_i the corresponding time instant, and treat them as experimental points. Them given, we want to identify the corresponding linear model

$$s = a \cdot t + b + \varepsilon \tag{6.7}$$

where ε is a model error.

One popular way to identify model (6.7) is by minimizing the squared difference between experimental points and model values at the corresponding instant. This linear least squares regression yields the equation of a line $s(t) = a \cdot t + b$ which is an estimate of our trend.

Now, if we limit to subtract the trend estimate from the original signal, we obtain

$$S_i = s_i - a \cdot t_i - b \tag{6.8}$$

noindent whose average is

$$\overline{S} = \frac{1}{n}\sum_{i=1}^{n} S_i = \frac{1}{n}\sum_{i=1}^{n} s_i - a \cdot \frac{1}{n}\sum_{i=1}^{n} t_i - b = \overline{s} - a \cdot \overline{t} - b \tag{6.9}$$

which is not the original $\overline{s}$, nor in general has some desirable property as being zero. So, within SonicLib liner trend removal is *mean-preserving*: after the linear trend has been removed using (6.8) the new signal mean is computed and subtracted from the transformed signal. Then, the mean of original signal is added back. In other terms:

$$S_i = s_i - a \cdot t_i - b - \overline{S} + \overline{s} \tag{6.10}$$

Changing the trend-freed signal so that its mean is preserved has no effect on covariances, meanwhile allowing to perform average-depending computations (for example finding rotation angles in eddy covariance).

For all this to be possible, however, we need each data item is accompanied by a non-ambiguous time stamp t_i In section 3.1.3 we have seen this is not necessarily the case: all we may be reasonably sure is time stamps are “regular”.

Time stamp regularity is just fine if all we need to do is partitioning a signal collected in an hourly file among various averaging blocks. But regrettably it is not sufficient to compose experimental points suitable to linear regression. As a workaround, and only for trend removal, an equally-spaced time stamp is synthetized from index by applying the following rule:

$$t_i = \delta t \cdot (i - 1) \tag{6.11}$$

where $\delta t = 1/f_s$.

It is worth mentioning this trend removal strategy is effective only when trend originates from a smooth change as temperature’s normal daily variation. It is completely worthless in cases where the trend takes for example the form of a sudden change, like when a cold or warm front breaks in, or at beginning of Foehn episode.

Another more subtle point is that trend removal as done in SonicLib is entirely “statistical”: it is always possible to perform it, but it has no “real” physical meanings. The trend a human eye and brain can detect, however, are of entirely physical origin. An ideal trend removal strategy should take this into proper account. In SonicLib, at least in the moment, we don’t.

6.4.6 Spike detection and processing

In ultrasonic anemometry lingo, the term *spike* is used as a synonym of outlier in a data series. So the question arises on agreeing on what an ”outlier” actually is - which also answers, at least in part, to the question on what is the most appropriate way of detecting them.

In the practice of statistics outliers are often intended as individual data whose values are not compatible with the distribution underlying the whole data set (outliers excluded). In other terms, it is assumed a data set always contains a subset which is drawn from some well known distribution (not necessarily normal), plus a few number of data whose value has been perturbed during the collection process.

The nature of this perturbation is left unspecified. But of course, its occurrence is sufficient to make that individual reading "something else" than the real data. Think, for instance, to a glitch in an analog to digital converter due to an electric power outage (maybe a brown-out": it's a rare event (which indeed *may* occur), whose most likely event is to alter the original reading imparting it a random deviation.

In strict principle the random deviation imparted by the measurement glitch is not guaranteed to be large: in many cases it is indeed small enough that its effect remains undetected. But on occasions it may result in a change so macroscopic that we can detect it easily. In this case, we have an outlier.

Assuming outliers exist is prudential in ultrasonic anemometry, even for sonics performing plausibility checking on individual data. Reasons for them abound, from electrical noise to alterations (e.g. rain drops) along the measurement path. Although rare, it is highly unlikely all of these events will be detected without error by an automatic method which is (at least with current electronic technology) performed on individual data without taking into account the past values history.

This, indeed, lifts the veil on what outliers are in practical sense: their nature is highly contextual. If you plot your data set in the time domain, outliers emerge as spikes on the plot. But if your sight is restricted to only one data item, then you have little to base your detection on, apart from some quite indirect indicators like for example an imbalance in the attenuation between direct and opposite sound pulse propagation paths.

And, yes: its very name reveals that spikes are a highly *visual* concept. Usually, translating visually obvious concepts into mathematical concepts is a hard nightmare. The human eye took billion years of random variation and natural selection to perfection. And the neural circuitry within of it and beyond, in the brain, is way more complex and powerful than any computer. It is very hard to simulate what we gut-feel as "visually obvious" using a piece of software, when simulating the behavior of a *single* neuron with sufficient accuracy requires a multiprocessor supercomputer (and the simulation proceeds slower than a normal neuron can operate). Really, there is no hope to reproduce visual feeling.

But, we can often find approximate methods. This is what happens with outliers. The most popular way of detecting them is by looking for data values which do *not* satisfy the condition

$$q_\epsilon \le x \le q_{1-\epsilon} \tag{6.12}$$

where q_u is u-th quantile drawn from the distribution underlying our data set. As this distribution is a priori unknown, quantiles from the data set itself are often used in practice. By doing so, we can see how "detecting outliers" actually means "discarding the lower and upper tails of the data set": not exactly the same thing, but practically sufficient in most cases.

Computing quantiles, unfortunately, demands much more work than calculating a mean and standard deviation. In practice you have to sort data, if you want to locate the smaller and the larger, and this is an $O(n \log n)$ operation, quite heavier than the $O(n)$ order of mean and standard deviation computations. This may not be a real challenge with today's computers, but it was decidedly not when ultrasonic anemometry began in the late seventies of past century.

The desire to locate spikes in real time then suggested to adopt a spike detection criterion of quite simplified nature, like this:

$$\mu - K\sigma \leq x \leq \mu + K\sigma \tag{6.13}$$

where μ and σ are mean and standard deviation of data set respectively, and K a real positive constant, one of whose most popular values is 3. Still today, by ingrained habit, this is the way spikes are detected in ultrasonic anemometry, and SonicLib adopts it instead of quantile-based method.

We notice one macroscopic thing: the fact of not satisfying (**??**) is equivalent to (6.12) *only if the underlying distribution is normal.* But this assumption is not necessarily true for experimental data! (This is a plain understatement: for sonic anemometer data sets it is plainly false). So, what to say? Fortunately, as often happens with practical statistics, the approximation we get is not that bad, so detecting outliers the wrong ways indeed works (with an accuracy sufficient to prevent losing our sleep).

There is another thing to say about spike detection, and it is a macroscopic one: *of course*, for it to make sense, the data set must be *stationary.* Would it not be, for example because of a linear trend, then standard deviation (or quantiles, for what matters) would inflate and finding just one outlier would become really a challenge. Think looking for the classical needle in a haystack ten times the size of a normal one.

That is to say, spike detection must be performed *after* trend removal!

Of course, we have to take into account a practical (unavoidable) difficulty: trend removal is based on finding a regression line, a process which is notoriously sensitive to outliers (which in the moment of trend identification are still there). The trend may then be located with some error, whose occurrence would likely have the effect of widening our fishing net somewhat.

To overcome this difficulty we could proceed iteratively, by re-fitting trend after spike have been treated in some way. In SonicLib, as in most practical implementations, this is *not* made, mainly to save computing time.

Now: what to do of the sppikes we have detected? In principle we could do many things. One method we can use is to say "OK, my ultrasonic anemometer performs a lovely plausibility checking on individual data, and I have no specific reason to not believe it". This head-in-the-sand approach may look dangerously akin to "managing by absence", but on some occasions may not be entirely without any sense.

Another possible approach is to shave spikes off in some way. A popular method is replacing their values by the linear interpolant of data immediately before and after. But: who guarantees that the real data would have been this way? Indeed, this method is very attractive as it does not increase the number of "invalid" data. This could make computations a bit easier to implement for programmers, at the price of introducing into data some degree of poetical invention. The same would indeed happen if we adopt any other, less crude, data imputation method.

Last, we may just suppress spikes by replacing them with NAs. In `R` this is no problem at all, as the language already includes the ability to get rid of invalid data when computing means and (co)variances (for example by `is.na=TRUE` key). This is the approach followed in SonicLib.

It is worth mentioning, in the interest of preventing a next little religion war, that replacing spikes with valid, although more domestic, data or killing them does not alter significantly the end results: by definition, spikes are "few"! One consequence of the "killing" approach is, we will not be able to use FFT based spectrum calculation if even one spike is found. But this is not a big deal for SonicLib: we may adopt in this case power spectrum estimation instead. Or, if we know our anemometer plausibility checks are acceptably sound, even choose to adopt the "head in the sand" approach and hope for the best.

6.4.7 Examples

Simple data averaging

In many cases all what done is just averaging data the simple way, accepting all defaults. This is accomplished by issuing a sequence like this:

```
> file.name <- "../TestData/20100516.10.csv"
> d <- get.raw.data(file.name, verbose=TRUE)
> d.stat <- average.sonic.data(d, time.stamp.from.name(file.name))
```

Result, stored in variable `d.stat`, contains one record per each of the two 30 minutes block within hour.

Using different delays for different variables

When more than one analog data set is in raw data file it may happen their data acquisition delays to differ. If this is the case, a unique real valued delay (like the default value) cannot be used, and function `set.delay` is to be used.

The following example shows how different delays are used, assuming a LiCor 7500 water and carbon dioxide detector and a Vaisala HMP45 thermo-hygrometer are connected to ultrasonic anemometer's analog input channels (and then, available in files):

```
> dly <- set.delay(q=0.3, c=0.3, temp=0.0, hrel=0.0)
> file.name <- "../TestData/20120401.03.csv"
> d <- get.raw.data(file.name, verbose=TRUE)
> d.stat <- average.sonic.data(d, time.stamp.from.name(file.name),
                 delay=dly)
```

6.4.8 Notes

Function `average.sonic.data` is seldom used directly, as most data processing occurs on entire campaigns more than a unique file. It may however happen access to data in a single hour is needed (maybe for diagnostic purposes), and then it makes sense to use `average.sonic.data` on its own.

Averages and covariances are computed according to the usual definitions, that is blockwise. No weighting nor pre-filtering strategy is employed during averaging (although signal preparation may have been made priori of this step).

The relative non-stationarity of wind component x is defined as

$$RN_x \doteq \frac{\delta x}{\overline{x}} \tag{6.14}$$

as defined in [18]. Non-steadiness of covariance $\overline{x'y'}$ is defined according to [5] as

$$NS_{\overline{x'y'}} \doteq \frac{\hat{x'y'} - \overline{x'y'}}{\overline{x'y'}} \tag{6.15}$$

where

$$\hat{x'y'} \doteq \frac{1}{n}\sum_{i=1}^{n} \overline{x'y'}_i \tag{6.16}$$

The symbol $\overline{x'y'}_i$ designates the i-th 5 minutes block within averaging time. n, the number of blocks, may be computed readily as $\frac{t_a}{5}$, where t_a represents the averaging time in minutes. Note n is an integer, because of the restrictions imposed on allowed values of t_a.

6.5 Average Data from Multiple Raw Data Files

6.5.1 Function Use

Computing averages is most often done on the result of whole measurement campaigns. If these are stored in a single file, then averaging may be performed using function

```
average.sonic.file.set(
  dir.name=".",
  time.zone="UTC",
  shift=0,
  averaging.time=30,
  sampling.rate=10,
  threshold=0.1,
  delay=0.3,
  trend.removal="none",
  spike.detection.threshold=3,
  spike.treatment="set.na",
  verbose=FALSE
)
```

with the following argument meaning:

dir.name Name of directry where data are looked for (default: current directory).

time.zone String containing the standard name of the time zone to which all time stamps will be attributed (default: "UTC").

shift Number of seconds to be added to time stamps to obtain "true" time, used when mass-adjusting a data set (default: 0).

averaging.time Value of averaging time to be used when processing data in set, in minutes (default: 30; allowed values: 10, 15, 20, 30, 60).

sampling.rate Sampling rate assumed for data in set, in Hertz (default: 10).

threshold Fraction of missing or invalid data above which an averages record is considered statistically invalid, and assigned the NA value (default: 0.1).

delay A real number indicating a unique delay value to applied to any optional columns simultaneously, or an object of class "delay.spec" indicating the delay used for any possible optional column. Objects of class "delay.spec" are created using function `set.delay`, whose use is described in an examples of section 6.4. Default: 0.3 s.

trend.removal is a string specifying the trend removal algorithm to use. Possible values are "none" (no trend removal is performed), "linear" (trend is estimated by linear least squares regression and removed in mean-preserving fashion). The default is "none".

spike.detection.threshold A real number indicating a unique spike detection threshold to be used for all channels, or a structure of type `spike.detection.threshold` containing the individual thresholds for each channel. All spike detection threshold values are positive numbers meaning a iven number of standard deviations. For example, a value of 2 means "two standard deviations". Default: 3.

spike.treatment String, meaning the type of processing to be made on spikes. The possible choices are "none" (no spike processing made, in which case parameter `spike.detection.threshold` is ignored), and "set.na" (spikes are replaced with NA values). Default: "set.na".

verbose Boolean flag indicating whether output messages should be printed (TRUE) or not (FALSE).

In case of failure a NULL value is returned.

If averaging terminated correctly then an object of class `sonic.avg.data` is returned (see section 3.3).

6.5.2 Troubleshooting hints

Function `average.sonic.file.set` is most likely your first contact with SonicLib: most of processing aimed at "distilling information from sonic data" starts from averaging out a data set, and this in turn involves one or more calls to `average.sonic.file.set` as the very first step.

Unfortunately, `average.sonic.file.set` is also quite a delicate step. In itself is not so dramatically difficult from, say, a mathematical or computer science standpoint. But on the other side, it interfaces with "standard SonicLib raw data files", the product of a heterogeneous set of converters in quick evolution.

Chances are non-zero that the last minute version of that new shiny converter does not really produce so "standard" SonicLib raw data files. This is something the kernel SonicLib and its developers can not directly control, converters often coming from end users.

In cases like these `average.sonic.file.set` stops on error. If parameter `verbose` is set to `TRUE` messages are printed in case of troubles: they may be illuminating but understanding them is much quicker with a little bit of a-priori information.

In the subsections following each of the important error messages is described in its meanings. Before to proceed, we note messages may come from `averaging.sonic.file.set`, `averaging.sonic.file` and `get_raw_data`, that is the function described here and the functions it calls in turn: in the first case the error is likely to reside in something pertaining the file set as a whole, while in the others some specific error on a single data file has occurred (in fact, all messages but one come from the single file command `averaging.sonic.file`).

Message average.sonic.file.set:: Error: No CSV files in directory

SonicLib raw data files are CSV file, and their extension must be ".csv". If the data directory contains no file with ".csv" extension then there is no point for `averaging.sonic.file.set` to do anything.

The most likely cause of this message is you have specified a wrong directory name. This may be because the directory does not exist, or it exists but really contains no CSV file.

Notice that if at least a CSV file exists as far as `averaging.sonic.file.set` is concerned it's all right. The message is not emitted and, as soon as `averaging.sonic.file.set` will try reading the wrong CSV file some other error will occur.

Message average.sonic.data:: error: Argument 'delay' is not a member of an instance of class "delay.spec"

This message means you have specified a delay of the "wrong type". Delays, used to cope with the small retard analog data sampled from sensors connected to ultrasonic anemometers often exhibit, can be only numbers (in which case the same delay value is applied uniformly to all channels), or lists of class "delay.spec").

Anything else would make `averaging.sonic.file.set` quite hysterical; to avoid this possibility a type check is made and in case of failure the message is printed.

If you see this message just check parameter `delay`.

Messages average.sonic.data:: error: Averaging time is not a multiple of 5 minutes, average.sonic.data:: error: Averaging time is not a divisor of 60 minutes, Message average.sonic.data:: error: Averaging time is not between 10 and 60 minutes

All these messages are self-explaining: averaging time must be a multiple of 5 minutes dividing exactly one hour and not smaller than 10 minutes.

In practice, the values allowed for the averaging time are 10, 15, 20, 30 and 60 minutes.

If the value of the averaging time is set to a number not satisfying any of these conditions one of the messages is emitted.

Message get.raw.data:: Error: No columns in input data

This message basically tells SonicLib did not understand what the content of a file is.

The most common reason is the file is not a SonicLib raw data file, although its type is CSV. From a purely technical standpoint this may mean the file has been prepared using a spreadsheet and exporting to CSV, but the original file did not contain tabular data.

In practice, the meanings is the directory assumed to contain only SonicLib raw data also contains something else, and of a very strange nature.

An important thing to notice is that despite of the "error“ nature of this message the function `averaging.sonic.file.set` does not stop upon encountering it: the offending file is just skipped and nothing bad happens (apart from the message).

Anyway, if this error annoys you the best you can do is moving the wrong file somewhere else. Locating it is quite simple: before printing the error message SonicLib dumps the name of the current file to the R concole.

It is not unworthy to stress that SonicLib expects the directory containing raw data files is "clean“, that is, contains nothing different than SonicLib raw data files.

Message get.raw.data:: Error: Number of data inconsistent with the expected number

This message in practice tells SonicLib was able to find something tabular in a data file, but the number of rows does not match with the expected number.

This latter is assumed equal to $n_s \doteq f_s \cdot 3600$, where f_s is the sonic's data acquisition frequency employed when collecting current data file. Given n, the number of data records actually present in file, the following indicator is formed:

$$F = \frac{|n - n_s|}{n_s} - 1 \tag{6.17}$$

If F is smaller than the threshold, the corresponding data file is rejected. F represents the deviation of total data in file from the number of expected data, expressed in fraction form. Because of the absolute value in 6.17 the check operates both when the number of actual data is smaller than expected, and when it's *larger*. This choice allows to trigger attention on data sets whose acquisition parameters differ from the ones assumed, which in many cases are just left to their default values: if this message occurs on any file in the data set, then you may be sure the assumed sampling frequency is wrong.

Assigning a sensible value to the threshold may involve contextual data. As a general rule, however, a tighter threshold might make sense when averaging and trend removal are made, or when an averaging period of less than an hour is adopted (regardless on trend removal state). The default in `averaging.sonic.file.set`, 0.05, reflects this.

Message get.raw.data:: Error: No 'u', 'v', 'w' or 't' data in file

This message tells that at least one of the mandatory columns expected in a SonicLib file is not present, and as a consequence the file is rejected.

A likely cause of this error is the supposed SonicLib raw data file is not really a SonicLib file. Or, the column names within of it are wrong (this *may* happen: SonicLib raw data file columns are spelled in lowercase, and maybe the converter wrote them in uppercase).

It's interesting to note the time stamp is not mentioned in this message. The reason is, time stamp is generated automatically if missing from data files. But: for this approach to make sense the file must be reasonably free from gaps, that is, F must be smaller than the (tight) threshold.

6.5.3 Examples

Averaging data "acritically"

Function `average.sonic.file.set` may be used to just load data and average them on the fly, maybe within an automatic offline data processing script like in this example:

```
d <- average.sonic.file.set(
  "../TestData",
  delay=0.3, verbose=FALSE
)
```

Preliminary assessment of data quality

The result of preceding call may then be used directly to feed data processing functions like `eddy.covariance`, as will commonly be made by many users (and automatic procedures).

But if we desire, we can do some more: use averaging and its associated calculation of relative nonstationarity and non-steadiness indices as a direct exploratory tool to preliminarily assess data quality.

In the next example we use averaging to check which hourly files are affected by significant non-stationarity on wind component w and sonic temperature *theta*.

This makes sense if we desire to compute turbulent heat flux from sonic anemometer data reliably, and we can afford the risk of rejecting low-quality data sets.

Of course things are not everytimes this lucky way. For example, if our objective is to feed a lagrangian dispersion model in real time then it could make better sense to be less finicky with data, and accept hours characterized by a sub-standard quality: in cases like this it may be better to use available measurements, accepting the risk some of them are not "scientific paper strength", instead of resorting to data imputation strategies (that is, invention of fictive but "plausible" data to fill gaps). As almost always happens with data processing, the "best" way depends very much on the context.

So, let's assume we want to collect the subset of data on which e.g. eddy covariance will likely get best results, maybe in view of a scientific publication. Because of the assumptions underlying eddy covariance, results tend to be better when data are "stationary".

In a real data set, however, stationarity will never be really fulfilled. The requirement of stationarity is simply too restrictive, and no natural data set will lack its part of trends, changes (often abrupt) of variance or covariance, and infinite more.

So pragmatically we define a hourly file to be "stationary enough" if relative non-stationarity of w is, say, less than 0.5 and the non-steadiness of $\overline{w'\theta'}$ less than 0.3.

Our first step is averaging data. But as we're interested in assessing quality of *hourly* files we avoid using the default of `averaging.time`, 30 minutes, and specify 60 minutes instead:

```
d <- average.sonic.file.set(
  "../TestData", averaging.time=60,
  delay=0.3, verbose=FALSE
)
```

Actual plotting may then be made with standard R commands, like this:

```
plot(
  d$data$t.stamp,
  d$data$w.avg,
  xlab="",
  ylab="Non-rotated W (m/s)",
  main="",
  type="l"
)
```

Here incidentally we see how convenient it is having adopted R for developing SonicLib: in addition to SonicLib specific function (in fact, a tiny set), we gain access to the immense graphic and data analysis facilities already built into R.

Taking this for granted, we may then plot the time evolution of relative non-stationarity and non-steadiness we're interested in, along with the corresponding limiting values we've selected so far. The result is shown in figures 6.3 and 6.4.

But before of this we may do a simpler check, directly on available data. In our case we may learn quite a bit just plotting the hourly averages of vertical wind velocity. The importance of this and other similar "simple" checks can not be overlooked: in our specific case, finding a significantly non-zero vertical mean before applying axis rotation may indicate some positioning problem, and more so if all values have the same sign, thus indicating some systematic trouble.

In our data set, "CRA.01", this is precisely what happens, as we may see in figure 6.2 where we see all non-rotated hourly means of vertical speed to be negative, and indeed quite large. Why does this happen? Is this related to some specific provenance direction, or happens in all cases? Answering these and other related questions is fundamental when documenting micro-meteorological campaigns, and often involves both describing site conditions and performing exploratory data processing before embarking in the huge processing work.

As we may see from figures 6.3 and 6.4, most hours are "good" (in the strict sense that both relative non-stationarity of w and non-steadiness of $\overline{w'\theta'}$ are below their respective thresholds. But some cases exist when one or both of these diagnostic values exceed the threshold.

What to do in these cases?

As mentioned before, no best a priori method can be devised, and a sensible answer must take the overall context into proper account. If, as we imagined,

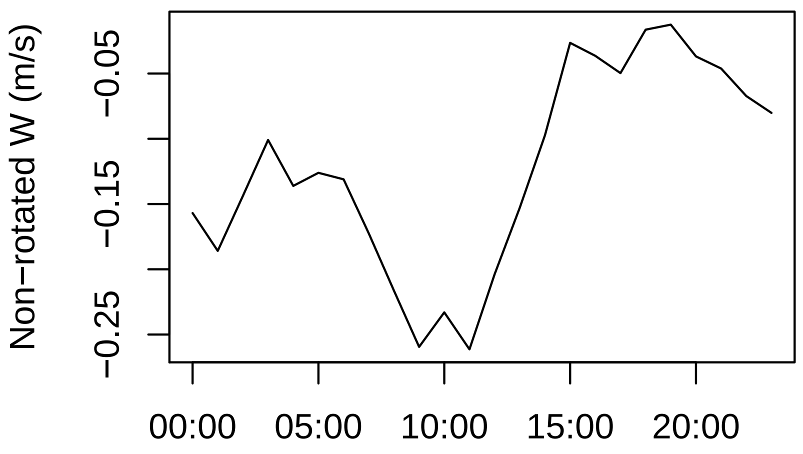


Figure 6.2: Time evolution of hourly-averaged wind component w in data set "CRA.01"

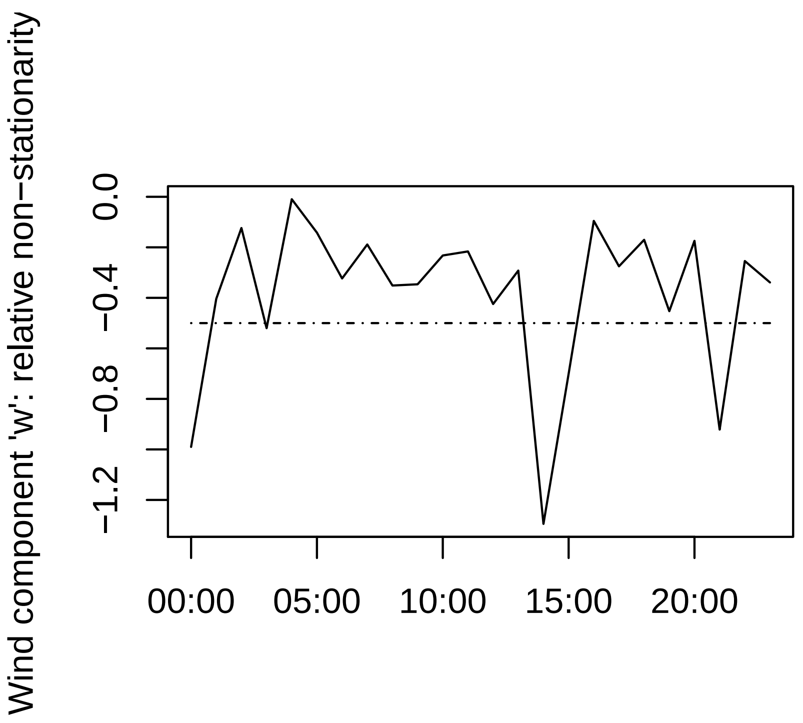


Figure 6.3: Time evolution of relative non-stationarity of wind component w in data set “CRA.01”; note the threshold value is negative, as all averages of w are lower than 0 in the test data set considered

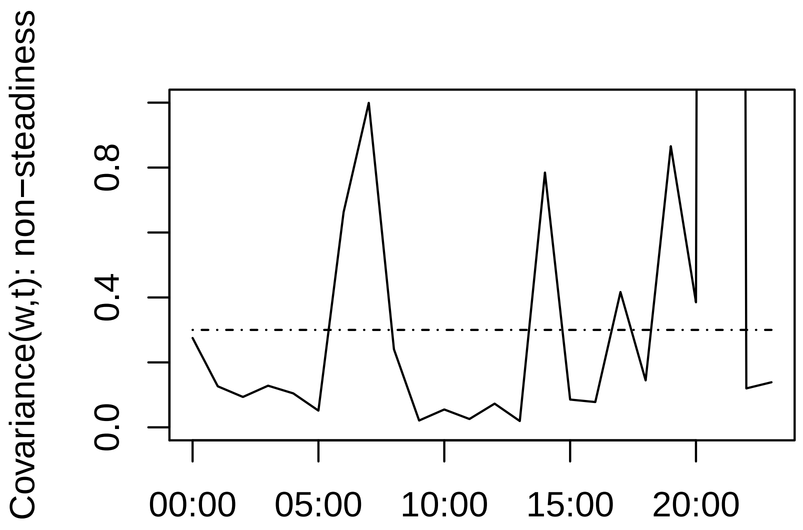


Figure 6.4: Time evolution of non steadiness of covariance $\overline{w'\theta'}$ in data set "CRA.01"

the purpose of the measurement campaign is to write a scientific paper stating some new truth, then it may make sense to *exclude* from any further use the hours whose non stationarity or non steadiness exceed their thrsholds.

Space restrictions do not allow us to proceed further, but an analysis might be worth on rejected data to see whether they really *are* non-stationary. And if so, *why*. Exceptions in data are very often revealing of something deep and unexpected, and it would be a little pity having to just not use them. These detailed analyses, in any case, are highly exploratory and visual, and here R really shines as a life-saving helper.

6.5.4 Notes

Function `average.sonic.file.set` assumes argument `dir.name` to specify a directory in which data files in standard SonicLib format are found, but this does not mean they are required to be unique, nor that some of them are completely missing.

To avoid confusion however it is highly recommended to avoid placing in the data directory anything different from data, and to separate data from different campaigns in different directories. Function `average.sonic.file.set` continues to operate normally in these cases, but results are homogeneous and without strange surprises - at least, strangeness stems from data themselves.

Chapter 7

Analyzing Data

7.1 Spectral Analysis

7.2 Multiresolution Analysis

Multi-resolution (or "multi-scale" - in this Manual we consider the two terms as synonyms) analysis is performed invoking the following function:

```
multires(t.stamp, val, steps)
```

where

t.stamp Vector, containing time stamps expressed in seconds, as in SonicLib raw data files

val Vector, the same length of `t.stamp`, containing the values to be analyzed

steps Vector containing a sequence of strictly decreasing numbers with the property that its i-th term is an exact multiple of $i+1$-th.

7.2.1 Background

The idea of applying multi-scale analysis to data sets is far from new. An example can be found in [4], which investigates the used of wavelets in statistics. Numerous works have been published on the related subject of multiscale modeling (see [1], [14], and in the specific context of ultrasonic anemometry [19]).

Three-axial ultrasonic anemometers capture instantaneous samples of the wind vector $\mathbf{v} = [u, v, w]$, temperature T and zero or more analog input values at rates exceeding 40Hz. In principle samples can be positioned precisely in time by means of a time stamp t. We may then treat sequences of consecutive samples as time series.

Let's now consider samples taken on a given hour. We may imagine their time stamps to form the sequence

$$\{t_k\}_{k=1}^{n} \tag{7.1}$$

where

$$t_k = t_0 + (k-1) \cdot \delta t \tag{7.2}$$

with t_0 the hour's initial instant and $\delta t = \frac{1}{f_s}$ the sampling period, reciprocal of sampling rate. Similarly we may introduce the sequences

$$\{\mathbf{v}_k\}_{k=1}^{n} = \{[u_k, v_k, w_k]\}_{k=1}^{n} \tag{7.3}$$

of wind vector values,

$$\{T_k\}_{k=1}^{n} \tag{7.4}$$

of temperature and of analog values, if any.

7.2.2 Multi-scale decomposition of a time-stamped data sequence

Timed sequences

Forgive me in advance for the nitty-gritty microscopic details which follow. My only justification is, they are quite unavoidable: in what is about to follow, precision is essential to get not lost. (If you still find me quite pedantic, OK, we'll make both a reason! By the way, this is not how I speak normally.)

So let's first take a generic *timed sequence*, that is a collection of generic data values, each associated to its time stamp,

$$X = \{(t_k, x_k)\}_{k=1}^{n} \tag{7.5}$$

under condition $t_k < t_{k+1}$. The t_k represent *time stamps*, and x_k the *values.* A *timed subsequance* of a timed sequence X is a timed sequence

$$Y = \left\{(t_{k_j}, x_{k_j})\right\}_{j=1}^{h} \tag{7.6}$$

with the ordered couples (t_{k_j}, x_{k_j}) chosen in X, and such that $t_{k_j} < t_{k_{j+1}}$. When Y is a timed subsequence of X we write $Y \dashv X$.

Time stamps of a timed subsequences need not a priori being consecutive, but when they are we have the important case of *proper timed subsequences.* A proper timed subsequence of X is a timed subsequence Y in which $k_j + 1 = k_{j+1}$. If Y is a proper timed subsequence of X we write $Y \prec X$. Clearly, $Y \prec X \rightarrow Y \dashv X$.

If $X = \{(t_k, x_k)\}_{k=1}^{n}$ is a timed sequence, by its *time sequence* we intend the sequence

$$T(X) = \{t_k\}_{k=1}^{n} \tag{7.7}$$

We may figure $T()$ as an operator acting on timed sequences and extracting the time part from them. In a very sense, operator $T()$ is a projection from the "space" (?) of timed sequences to that of ordinary finite sequences in the real field. But OK, this is not really necessary to our (after all very simple) needs, so let this mention stay just as a curiosity. We'll use it anytime useful, without overcommitment.

Two timed sequences X_1, X_2 are *time-disjoint* (or simply *disjoint*) if

$$T(X_1) \cap T(X_{21}) = \emptyset \tag{7.8}$$

that is, if their respective time sequences have no time instants in common. When two timed sequences X_1, X_2 are disjoint it makes sense to define their *merge* $X_1 \oplus X_2$ as the time sequence consisting of all couples (t, x) from either timed sequences, ordered with respect to time the usual way.

If $X = X_1 \oplus X_2$, then $X_1 \dashv X$ and $X_2 \dashv X$. Nothing dramatic: just another way to say the same thing. It is much more interesting to our case when $X = X_1 \oplus X_2$ with $X_1 \prec X$ and $X_2 \prec X$, that is when the two timed sequences contributing to the merge are each proper timed subsequences of X. In this case, by necessity, either all time values of X_1 precede all time values of X_2, or all time values of X_2 precede all time values of X_1. When this happens we can say of a *proper* merge.

This leads us to the important concept of *partition* of a timed sequence X. We may define it as a set $\{X_1, X_2, \ldots, X_p\}$ such that $X = X_1 \oplus X_2 \oplus \ldots \oplus X_p$. The partition is *proper* if $X_i \prec X$ for $i = 1, 2, \ldots, p$.

Last, it is worth introducing a symbol to say all times in a timed sequence X_1 precede all times in sequence X_2: we may say X_1 *precedes* X_2, or $X_1 < X_2$. This concept of precedence allows us to establish a convention which simplifies all further steps: whenever we have a proper partition of X, $\{X_1, X_2, \ldots, X_p\}$, we assume $X_1 < X_2 < \ldots < X_p$.

Interval induced partitions

Imagine E is a subset of the real line such that $T(X) \subset E$ If we chose wisely, we may get an E of finite length (sure we could chose an infinite length interval E: but it would give us no special advantage). Now, let $F \subset E$. We can say the timed subsequence Y of X is *induced by* F if all the elements of $T(Y)$ are "inside" of F and not elsewhere in E, that is, $T(Y) \subset F$ and $T(Y) \cap E \setminus F = \emptyset$. When this happens we can write $Y = X_F$.

If we don't specify how F is made, then we leave the door open to it having a lot of holes, so that all we can say is $Y \dashv X$. But if F is an interval, then by necessity $Y \prec X$: this, for us, is the interesting case.

Now, let's consider a partition $\{F_1, F_2, \ldots, F_p\}$ of the interval E in intervals F_i (that is, a collection of intervals which are mutually disjoint and whose union coincides with E). Its corresponding *induced proper partition* of X is $\{X_{F_1}, X_{F_2}, \ldots, X_{F_p}\}$.

Partitioning the time axis in equal length intervals and using them to slice a timed sequence is, as we'll see in the next section, the key idea of the kind of multiscale analysis we'll see in this work.

Temporal scales and temporal scales sequences

From now on we begin being practical. Consistently with SonicLib way of storing data file, we can imagine E to span exactly an hour starting at instant t_0, that is, reasoning in seconds,

$$E = \{t \,|t_0 \leq t < t_0 + 3600\} \tag{7.9}$$

The interval E is open on its upper limit, according to the convention that the initial instant of an hour belongs to the hour.

A *temporal scale* is here considered a synonym of a time interval length. In our case not all time scales are interesting, but only those which divide an hour exactly. That is, we only consider integer time scales Δt with the property that $\Delta t \mid 3600$.

Given an hour E and a timed sequence X, a temporal scale Δt identifies a (proper) partition on X defined as

$$\Xi(X, \Delta t) = \left\{ X_{E_1}, X_{E_2}, \ldots, X_{E_p} \right\} \tag{7.10}$$

where $E_i = \left\{ t \left| t_i \leq t < t_i + \frac{3600}{\Delta t} \right. \right\}$ and $t_i = t_0 + (i-1) \cdot \Delta t$.

A *temporal scale sequence* is an indexed collection $\{\Delta_i t\}_{i=1}^{m}$ of distinct values, with $\Delta_{i+1} t \mid \Delta_i t$ for $i = 1, 2, \ldots, m-1$, and $\Delta 1 t \mid 3600$. An example of temporal scale sequence (actually one which is most used) is

$$\Delta = \{3600, 600, 60, 12, 1\} \tag{7.11}$$

In general, we may think of a temporal scale sequence having form

$$\Delta = \{\Delta_1 t, \Delta_2 t, \ldots, \Delta_q t\} \tag{7.12}$$

Because of the way the various $\Delta_j t$ have been chosen (each one is a non-trivial multiple of the following one), their corresponding induced partitions $\Xi(X, \Delta_i t)$ become progressively finer and finer, in the sense their intervals become progressively smaller, and with the nice property that intervals at any level can only be subsets of intervals of the preceding level. That is, no new level interval can split among two intervals at preceding level. This geometric property is the reason for the divisibility rule in temporal scale sequences.

Partition statistics of a timed sequence

Let $\Xi(X, \Delta t)$ be the timed sequence partition induced on X by time scale Δt. Each element X_i of the partition is itself a timed sequence, and so has a value part, v_{ij} with $j = 1, \ldots, k_j$. These values can be used to form statistics, in particular mean and variance,

$$\mu_i = \frac{1}{k_i} \sum_{j=1}^{k_i} v_{ij} \tag{7.13}$$

and

$$\sigma_i^2 = \frac{1}{k_i} \sum_{j=1}^{k_i} (v_{ij} - \mu_i)^2 \tag{7.14}$$

The collections

$$\{\mu_i \,|i = 1, 2, \ldots, k_{i*} \} \tag{7.15}$$

and

$$\left\{ \sigma_i^2 \,|i = 1, 2, \ldots, k_i \right\} \tag{7.16}$$

can be termed respectively *partition means* and *partition variances.* In the same way we may form partition minima, maxima, quantiles, skewness, and any other statistic we may build on a finite set of real numbers.

Scale approximations to a timed sequence

Let X be a timed sequence, Δt a time scale, and $\Xi(X, \Delta t)$ the corresponding partition. As we have seen already we can construct partition means μ_i.

Now, we form a new timed sequence $M(X, \Delta t)$ approximating X at time scale Δt. According to definition ()7.10), $\Xi(X, \Delta t) = \left\{X_{E_1}, X_{E_2}, \ldots, X_{E_p}\right\}$.

For each $i = 1, 2, \ldots, p$ let's consider the timed sequence X_{E_i}, with time sequence $\{t_{ij}\,|j = 1, 2, \ldots, k_i\}$. As usual we can get the corresponding partition mean, μ_i, and form the new timed sequence

$$M\left(X_{E_i}\right) \doteq \{(t_{ij}, \mu_i)\,|j = 1, 2, \ldots, k_i\} \tag{7.17}$$

Finally we form

$$M(X) = M\left(X_{E_1}\right) \oplus M\left(X_{E_2}\right) \oplus \ldots \oplus M\left(X_{E_p}\right) \tag{7.18}$$

$E(X)$ represents a "scale approximation" in two ways. First, because it has been made according to a given time scale. And second, because when its graph is plotted it closely resembles a discrete version of "scale function" as found in integration theory. Forgive me this pun - it was completely involuntary!

The vector space of timed sequences

Let's consider the set $\mathcal{S} = \{X\,|T(X) = T\}$ of all timed sequences X sharing the same time sequence T. This set can be mapped on $\mathbb{R}^n$, where n is the number of elements of T, using the following definition:

$$s(\{(t_i, v_i)\,|i = 1, 2, \ldots, n\}) = \sum_{i=1}^{n} v_i \cdot \mathbf{j}_i \tag{7.19}$$

As $T = \{t_i\}_{i=1}^{n}$ is fixed, mapping s is clearly a bijection. We may use this fact to *identify* the set $\mathcal{S}$ with $\mathbb{R}^n$ and, in particular its vector space structure.

In practice, by adopting this view we gain the ability of multiplying a timed sequence by a scalar, and summing two timed sequences. It also means we can speak of a "zero" timed sequence (one, all whose values are 0).

The residual of the scale approximation to a timed sequence

Remembering (7.17) and (7.18), and taking into account the vector space nature of the space of timed sequences on same T, it makes sense to introduce the *residual* of a scale approximation as

$$R(X) \doteq X - M(X) \tag{7.20}$$

We can say the same, stating that a signal (represented by a timed sequence) may be decomposed in the sum of its scale approximation and residual. If you are accustomed to micro-meteorological lingo you may get the similarity between this "scale decomposition" and the Reynolds decomposition of a turbulent signal in mean and fluctuating parts. Indeed, this similarity is more than casual, at least from the mathematical standpoint.

In fact, it is easy to verify that scale decomposition satisfies all Reynolds postulates:

1. $M(M(X)) = M(X)$
2. If α is a scalar, then $M(\alpha X) = \alpha M(X)$
3. $M(X + Y) = M(X) + M(Y)$
4. $M(X \cdot M(Y)) = M(X) \cdot M(Y)$

All these four properties are immediate consequence of the definition of the arithmetic mean and the way we propagated it to all value points of any time block. For reasons of lack of space I'll not give a mathematical proof, but building it is really straightforward (forgive me this statement).

We may draw from this an important consequence: scale decomposition is a special case of Reynolds decomposition. In practical terms, it is akin to the very popular block averaging scheme, with the difference that the underlying time scale Δt may be very small.

The fact of satisfying Reynolds postulates may be interpreted stating scale approximation is a "true averaging" operation. The practical side of it is that if we meet equations involving sums and products of timed sequences we may always manipulate them to partition them in a "mean" and "fluctuating" parts, as we can do on a single timed sequence.

Multi-scale decomposition of a timed sequence

We may now consider the case of a temporal scale sequence. As their definition demands, its elements $\Delta_j t$ are subsequent divisors, so their corresponding scale approximations to a timed sequence X are finer and finer. But more important, they allow a very interesting construction.

We begin with X itself, and take the scale approximation corresponding to the partition induced by $\Delta_1 t$, which we may denote by $M_1(X)$. Once this is computed we may take the residual, $R_1(X)$ as we've seen in the previous section. We don't stop here, however.

On the second step, we build the scale approximation of $R_1(X)$ corresponding to the time scale $\Delta_2 t$, $M_2(X)$, and from this the corresponding residual, $R_2(X) = R_1(X) - M_2(X)$.

On third step we redo all these operations, but this time on residual found at end of second step, and with time scale $\Delta_3 t$, obtaining the scale approximation $M_3(X)$ and its corresponding residual $R_3(X)$.

Iterations continue until the last time scale, Δ_q, when we get $M_q(X)$ and $R_q(X)$. At this point we may look back to what we have found. First, we have a sequence

$$M_1(X), M_2(X), \ldots, M_q(X) \tag{7.21}$$

of scale approximations, and a sequence of residuals

$$R_1(X), R_2(X), \ldots, R_q(X) \tag{7.22}$$

Because of the way we have defined scale approximations and residual, and because of time scale divisibility, we can say a very important thing:

$$X = M_1(X) + M_2(X) + \ldots + M_q(X) + R_q(X) \tag{7.23}$$

Equation (7.23) expresses the multi-scale decomposition of a timed sequence. The multi-scale decomposition exists, each time we can guarantee any of the time intervals induced by $\Delta_i t$ and t_0 is non-empty (as we always have assumed implicitly; we could relax this requirement, for example imposing the value of μ_i corresponding to an empty set is 0, or using some special symbol to denote missing values, like the `NA` in R; things would become very complicate however, at no significant gain: so, we stick to our original assumption). As the mean is unique, so scale approximations are, and so it is for the multi-scale decomposition.

Basic properties of multi-scale decomposition

Orthogonality Turning attention to the actual meaning of the scale decomposition, we can start by noticing that each scale approximation $M_i(X)$ provides information about variation of original timed sequence X at time scale $\Delta_i t$ only, very much alike to Discrete Fourier Transforms at a given sequence.

Indeed, an orthogonality property follows for multi-scale decompositions as well. To see this, let's take the dot product $M_i(X) \cdot M_j(X)$, defined as the dot product of values in their time order. If $i = j$, it will be in general greater than zero (unless the very exceptional case in which one or two of the tributaries are identically 0).

But if $i \neq j$, then by necessity $M_i(X) \cdot M_j(X) = 0$. In fact: let's assume $i < j$ (if this is not, we can just switch i and j in the following "proof", and things go well anyway). The divisibility of time scales ensure that any time block induced by the i-th time scale contains some sub-blocks at j-th time scales. If we now compute the mean value of all j-th scale sub-blocks of any i-th scale time block, we get 0: all eventual non-zero part of the average has already been accounted for by the i-th scale approximations, and subtracted from the residual.

Now, we can acknowledge two things. First, if the average of j-th sub-blocks is zero the same must be for the average multiplied by any scalar, that is, in particular, the sum of all j-th sub-blocks of an i-th block. And second, as far as j-th sub-blocks are concerned, they "see" the i-th block as constant. So the part of dot product pertaining to i-th block can be simplified to the product of a constant by the sum of j-th sub-block values, which we know being zero. That is, it is zero. But we chose the i-th block arbitrarily, so this must happen to *any* i-th block, and then to the whole dot product.

The property of orthogonality expresses the fact that once we build the scale approximation at level i acting on level $i-1$ residual, we are guaranteed that all information at levels $i-1$ and less has already been "stripped" by all preceding level scale approximations. In full analogy to Fourier analysis of a discrete time series.

Independence from time spacing I've not mentioned this explicitly until now, but I've always assumed the time spacing of a timed sequence X needs *not* being constant.

This fact, which would give a lot of trouble to discrete Fourier transforms, is completely harmless to multi-scale decomposition, by the very way we have constructed it.

Multi-scale decomposition independence from time spacing makes it suitable to treat classes of timed data which are not, strictly speaking, "time series". In this a priori very large class we can identify two subclasses of great importance in the real world:

- Time series with (not too much) gaps, as happens for example when data are collected on a fixed rate by an instrument performing data plausibility checking (as for example Metek USA-1 anemometers).
- Time series collected on an uneven rate, for example data collected from a high-latency wireless network like ZigBee.

Convergence of decomposition to the original signal Let's consider the *k-th cumuate scale approximation*

$$S_k(X) = \sum_{i=1} kM_i(X) \tag{7.24}$$

Clearly (7.24) is a scale function (in integration theory sense), and it "converges" to X in the sense that $S_p(X) = X - R_q(X)$ according to (7.23): numerically, $R_q(X)$ tends to have a "small" magnitude.

Sequential compatibility with Reynolds postulates We had seen already how scale approximation is compatible with Reynolds postulates ([33]), but we've done it informally. In this section we'll see a proof, out of the many possible, dealing with multiple time scales combined. My proof is straightforward, but unfortunately full of many nitty-gritty details, so I feel nice to give a short outline of it.

Our first step will be to develop a bit of formalism allowing us to express $M_k(M_l(X))$, where $l < k$. That is, to write the k-th scale approximation of M_l.

This done, we will see a single scale decomposition is involutory. We'll then prove that scale decomposition is compatible with the vector space structure of the set of timed sequence sharing their times set with X. This proves at once postulates 2 and 3.

As this is done, involutory character is extended (in a specific way) to cumulate scale approximations.

Finally we'll prove the fourth postulate, exploiting the regularities highlighted by the formalism developed on first step and the vector space nature proven on previous step.

Some auxiliary formalism If we plot $S_k(X)$ we would easily find it is a scale function, consistent with the intervals induced on hour E bytime scale $\Delta_k t$.

This may be said more precisely. Given $\Delta_k t$, let τ_{ki} be the subset of $T(X)$ contained into the i-th interval of the partition induced by $\Delta_k t$; index i range from 1 to $\frac{3600}{\Delta_k t}$. Now, let's consider index l with $1 \leq l \leq k$, and let τ_{lj} be the subset of $T(X)$ contained into the j-th interval of the partition contained in $\Delta_l t$, with $j = 1, 2, \ldots, \frac{3600}{\Delta_l t}$. By our assumption on time scales sequences, we know $\Delta_k t$ divides $\Delta_{k-1} t$, which in turn divides $\Delta_{k-2} t$, …, which divides $\Delta_l t$. But relation "divides" is a partial order on integers, so by transitivity we have that $\Delta_k t$ divides $\Delta_l t$. So, let $u = \frac{\Delta_l}{\Delta_k}$. By divisibility we know u is an integer. Moreover, τ_{lj} may be partitioned in u blocks τ_{ljv} with $v = 1, 2, \ldots, u$.

Reasoning in terms of values of $M_l(X)$, we know that on τ_{lj} the value is constant and equal to, say, μ_{lj}. This must also be on all the τ_{ljv} in which τ_{lj} partitions when looking for the k-th scale approximation of $M_l(X)$.

We conclude the current section by noting that $M_l(M_k(X)) = 0$ whenever lk. This happens because of the way successive scale approximations are built on the residuals. More specifically, $M_{l+1}(X)$ is built applying $l+1$ scale approximation on the residual $R_l(X)$, whose l-th scale approximation is 0 by construction. But this is also true of the l-th scale approximation of $R_{l+2}(X)$, and so on until k-th scale.

More precisely, let's consider l-th and $l+1$-th scales, and let's consider $M_l(X)$ and $R_l(X)$. On all sets τ_{lj} the value μ_{lj} is constant in $M_l(X)$. When we take the residual $R_l(X)$, its values r_{ljq} on τ_{lj} are $v_{ljq} - \mu_{lj}$. By applying scale approximation $l+1$ to $R_l(X)$ set τ_{lj} splits by division in subsets τ_{ljv}, on each of which the average is taken, of the form $\mu_{ljv} = \frac{1}{|\tau_{ljv}|}\sum_{p\in\tau_{ljv}} v_{ljvp} - \mu_{lj}$.

Now, let's consider the l-th scale approximation of the newly computed averages. It is $\tau_{lj} = \cup_v \tau_{ljv}$ and average on τ_{lj} is

$$\mu'_{lj} = 1/\frac{\Delta_l t}{\Delta_{l+1} t} \sum_{v=1}^{\frac{\Delta_l t}{\Delta_{l+1} t}} \frac{3600}{\Delta_{l+1} t}\mu_{ljv} = \frac{3600}{\Delta_l t} \sum_{v=1}^{\frac{\Delta_l t}{\Delta_{l+1} t}} \mu_{ljv} \tag{7.25}$$

The last member of equation (7.25) may be rewritten as

$$\frac{3600}{\Delta_l t} \sum_{v=1}^{\frac{\Delta_l t}{\Delta_{l+1} t}} \frac{1}{|\tau_{ljv}|} \sum_{p\in\tau_{ljv}} (v_{ljvp} - \mu_{lj}) \tag{7.26}$$

We can see expression (7.26) is 0. Backtracking, we see the same must be for (7.25), and in conclusion, $M_l(M_{l+1}(X)) = 0$. The same argument may be developed with $l+2$ in place of $l+1$, and so on, until k. Our arbitrary choice of l then proves $M_l(M_k(X)) = 0$ for $l = 1, 2, \ldots, k-1$.

Involutory character on a time scale and all time scales preceding it We have just seen the value of $M_l(X)$ on each τ_{ljv} is equal to μ_{lj}, then constant. As the mean of a constant is that same constant, we can say the value of $M_k(M_l(X))$ on τ_{ki} is also μ_{lj}. This means that $M_k(M_l(X)) = M_l(X)$ for any $l = 1, \ldots, k-1$.

If $l = k$, then the value of $M_k(X)$ on τ_{ki} is μ_{ki}, that is once again constant. Its average on τ_{ki} is then again τ_{ki} so that $M_k(M_k(X)) = M_k(X)$.

Vector space compatibility of scale approximation Let X and Y be two timed sequences with $T(X) = T(Y)$, and α, β two scalars. Then we may form the timed sequence $\alpha X + \beta Y$.

Now, imagine applying k-th scale approximation $M_k(\alpha X + \beta Y)$. If we restrict our attention to subset τ_{ki}, where X values are $x_1, x_2, \ldots, x_{n_{ki}}$ and Y values $y_1, y_2, \ldots, y_{n_{ki}}$, we have that values of $\alpha X + \beta Y$ are $\alpha x_r + \beta y_r$ for $r = 1, \ldots, n_{ki}$.

If we take the average, necessary to form k-th scale approximation, we get it is

$$\frac{1}{n_{ki}}\sum_{r=1}^{n_{ki}}[\alpha x_r + \beta yr] = \alpha\frac{1}{n_{ki}}\sum_{r=1}^{n_{ki}} x_r + \beta\frac{1}{n_{ki}}\sum_{r=1}^{n_{ki}} y_r = \alpha\mu_{ki}(X) + \beta\mu_{ki}(Y) \quad (7.27)$$

As we chosen the τ_{ki} arbitrarily among the available subsets, the preceding property must be true for all subsets induced by $\Delta_k t$. So we may say

$$M_k(\alpha X + \beta Y) = \alpha M_k(X) + \beta M_k(Y) \quad (7.28)$$

Now, let's consider the sum

$$S_k(\alpha X + \beta Y) = \sum_{l=1}^{k} M_l(\alpha X + \beta Y) \quad (7.29)$$

By applying vector compatibility on each term we can write

$$S_k(\alpha X + \beta Y) = \sum_{l=1}^{k} M_l(\beta Y) + \sum_{l=1}^{k} M_l(\beta Y) = \alpha S_k(X) + \beta S_k(Y) \quad (7.30)$$

Involutory property to cumulant scale approximations Note we may apply this property to any timed sequences, so in particular scale approximations induced by different scales. More specifically, we can write

$$M_k(M_1(X) + M_2(X) + \ldots + M_k(X)) = M_1(X) + M_2(X) + \ldots + M_k(X) \quad (7.31)$$

that is

$$M_k(S_k(X)) = S_k(X) \quad (7.32)$$

In addition, from $M_l(M_k(X)) = 0$ whenever $l < k$ we have

$$M_1(M_k(X)) + M_2(M_k(X)) + \ldots + M_k(M_k(X)) = S_k(M_k(X)) = M_k(X) \quad (7.33)$$

Combining the last expression with (7.32) we get finally

$$S_k(S_k(X)) = S_k(X) \quad (7.34)$$

The fourth Reynolds postulate Let X and Y be two timed sequences with $T(X) = T(Y)$. We can then form the expression $M_k(XM_k(Y))$. As $M_k(Y)$ is constant over all the same index sets τ_{ki} on which averages are computed to build the $M_k(XM_k(Y))$ we may "move $M_k(Y)$ out of summation and then, by applying vector compatibility, we can say

$$M_k(XM_k(Y)) = M_k(X)M_k(Y) \quad (7.35)$$

Now let's consider $M_k(XS_k(Y)) = M_k(X\sum_{l=1}^{k} M_k(Y))$. We can say

$$\begin{aligned} M_k(X\sum_{l=1}^{k} M_l(Y)) &= M_k(\sum_{l=1}^{k} XM_l(Y)) = \sum_{l=1}^{k} M_k(XM_l(Y)) \\ &= \sum_{l=1}^{k} M_k(X)M_l(Y) = M_k(X)\sum_{l=1}^{k} M_l(Y) \\ &= M_k(X)S_k(Y) \end{aligned} \tag{7.36}$$

We also have $M_l(XM_k(Y)) = M_l(X)M_k(Y)$. To see this, we may substitute $Y' = M_k(Y)$ and apply (7.35). Now let's take

$$\sum_{l=1}^{k} M_l(XS_k(Y)) = \left(\sum_{l=1}^{k} M_l(X)\right) S_k(Y) \tag{7.37}$$

that is,

$$S_k(XS_k(Y)) = S_k(X)S_k(Y) \tag{7.38}$$

This completes the proof thet both operators $M_k(\cdot)$ and $S_k(\cdot)$ satisfy the Reynolds postulates - as promised.

Variance spectrum in multi-scale decomposition

The mean and variance of a timed sequence We first of all define the *mean* and *variance of a timed sequence* as the variance of its values:

$$\mu(X) = \frac{1}{n}\sum_{i=1}^{n} v_i \tag{7.39}$$

and

$$\sigma^2(X) = \frac{1}{n}\sum_{i=1}^{n} [v_i - \mu(X)]^2 \tag{7.40}$$

Mean and variance of a scale approximation Let's now consider the scale approximation $M(X)$ of a timed sequence X induced by scale Δt, and its associated residual $R(X)$.

We may say Δt partitions the hour in $m = \frac{3600}{\Delta t}$ intervals of same length $\tau_j, i = 1, \ldots, m$. Let $n_j = |\tau_j|$, so that $n = \sum_{j=1}^{m} n_j$.

By construction we know $M(X)$ is constant (and equal to μ_j) over any τ_j, so we may write

$$\mu(M(X)) = \frac{1}{n}\sum_{j=1}^{m} n_j\mu_j \tag{7.41}$$

that is, the mean of a scale approximation is a weighted average of its interval means. From basic statistics we know the mean operator is linear, so we may also say

$$\mu(X) = \mu(M(X) + R(X)) = \mu(M(X)) + \mu(R(X)) \tag{7.42}$$

Now, the residual $R(X)$ was obtained by subtracting $M(X)$ from X, that is, the interval mean τ_j to any value v_i with $i \in \tau_j$. So, its mean can be written as

$$\mu(R(X)) = \frac{1}{n}\sum_{j=1}^{m}\sum_{i\in\tau_j}(v_i - \mu_j) = \frac{1}{n}\sum_{j=1}^{m} n_j\,(\mu_j - \mu_j) = 0 \tag{7.43}$$

so replacing in (7.42) we get

$$\mu(M(X)) = \mu(X) \tag{7.44}$$

Turning attention to variance, and remembering its basic properties, we can write

$$\sigma^2(X) = \sigma^2(M(X) + R(X)) = \sigma^2(M(X)) + 2C(M(X), R(X)) + \sigma^2(R(X)) \tag{7.45}$$

where

$$C(M(X), R(X)) = \frac{1}{n}\sum_{i=1}^{n}[m_i - \mu(M(X))]\cdot[r_i - \mu(R(X))] \tag{7.46}$$

Remembering (7.43) and (7.44), equation (7.46) can be put in the following form:

$$C(M(X), R(X)) = \frac{1}{n}\sum_{i=1}^{n}[m_i - \mu(X)]\cdot r_i \tag{7.47}$$

Now, by construction,

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}[m_i - \mu(X)]\cdot r_i &= \frac{1}{n}\sum_{j=1}^{\frac{3600}{\Delta t}}\sum_{k=1}^{n_k}[\mu_j - \mu(X)]\cdot r_{jk} \\
&= \frac{1}{n}\sum_{j=1}^{\frac{3600}{\Delta t}}[\mu_j - \mu(X)]\sum_{k=1}^{n_k} r_{jk} \\
&= \frac{1}{n}\sum_{j=1}^{\frac{3600}{\Delta t}}[\mu_j - \mu(X)]\cdot 0 = 0
\end{aligned} \tag{7.48}$$

In conclusion, $C(M(X), R(X)) = 0$ so that

$$\sigma^2(X) = \sigma^2(M(X) + R(X)) = \sigma^2(M(X)) + \sigma^2(R(X)) \tag{7.49}$$

Multi-scale decomposition of variance The construction in previous section may be applied to a whole time scale sequence yielding the *multi-scale variance decomposition* corresponding to the given scale sequence.

$$\sigma^2(X) = \sum_{k=1}^{p}\sigma^2(M_k(X)) + \sigma^2(R_p(X)) \tag{7.50}$$

This multi-scale variance decomposition is a discrete analogue of the power spectrum (duly integrated with respect to frequency bands).

7.2.3 Examples

Assessing turbulence scales

Wind components and temperature data gathered by an ultrasonic anemometer contain information pertaining phenomena of different qualitative type.

For example, the vertical wind and temperature are likely to bear information pertaining exchanges of thermal and mechanical energy between ground and the lower atmosphere. Horizontal wind components, in addition, are likely to contain information from synoptic scale circulation down to local effects like breeze alternation of meandering past a stack.

Multi-scale analysis allows to easily quantify the amount of energy stored in each time scale: it is the variance associated to each time scale!

The plot of variance as a function of time scale may be expressed in the more familiar terms of frequency, by using the reciprocal of time scales, and adopting a log axis for abscissas.

An example of this representation can be seen in figures 7.1, 7.2, 7.3 and 7.4.

Assessing concentration variability

The study of chemical concentrations and their change with time has huge practical applications to fields ranging from air quality management to defense and civil protection.

Of special concern are chemicals which have immediate effects as soon as their value exceeds a given threshold.

As an example we may consider the case of an odor-carrying chemical, whose presence can be detected by people as soon as its concentration averaged over the time span of a breath exceeds a given odor threshold. Conventionally, a "breath time span" is taken to be 12 seconds.

Now: concentration c can be partitioned as any atmospheric scalar in the sum of a *mean* part, $\overline{c}$, and a *turbulent fluctuation*, c'. The mean part $\overline{c}$ is likely to exhibit an important variation due to the concurrent occurrence of phenomena like lateral meandering of plumes, changes in airflow due to local circulation patterns, and the like. These phenomena interest many temporal scale, so that their separation may be an ideal application of multi-scale analysis.

In this section however we investigate an interesting special case. Suppose you are trying to estimate hourly variability (that is, in particular, variance) of concentration by using a "conventional" model (i.e. Calpuff) on a "short" time frame, say with ten minutes averaging time. If ou do so, and you feed the model with direct reading from an ultrasonic anemometer (that is, 10-minutes means and sigmas of wind components), you find an "hourly" field be the time average of 6 10-minutes fields. Why just ten minutes? Maybe, because your meteorological data are routinely average this way, and not differently (indeed, 10 minutes averaging is common in civil protection and other similar "short-term" applications).

The question then arise, whether the variance of ten-minutes averages gives some reliable information about, say, 12-seconds means (that is, a breath). And

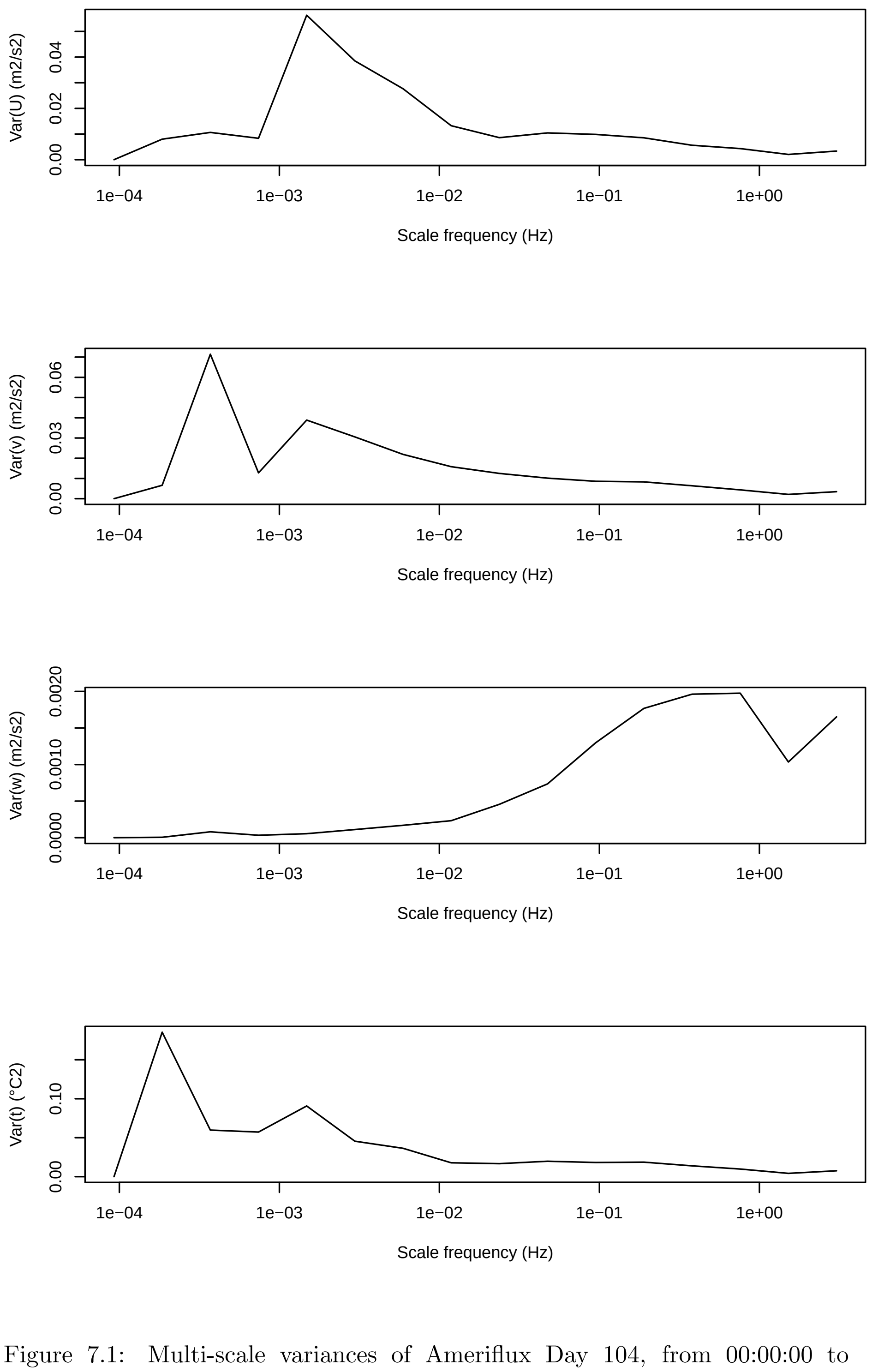


Figure 7.1: Multi-scale variances of Ameriflux Day 104, from 00:00:00 to 03:00:00

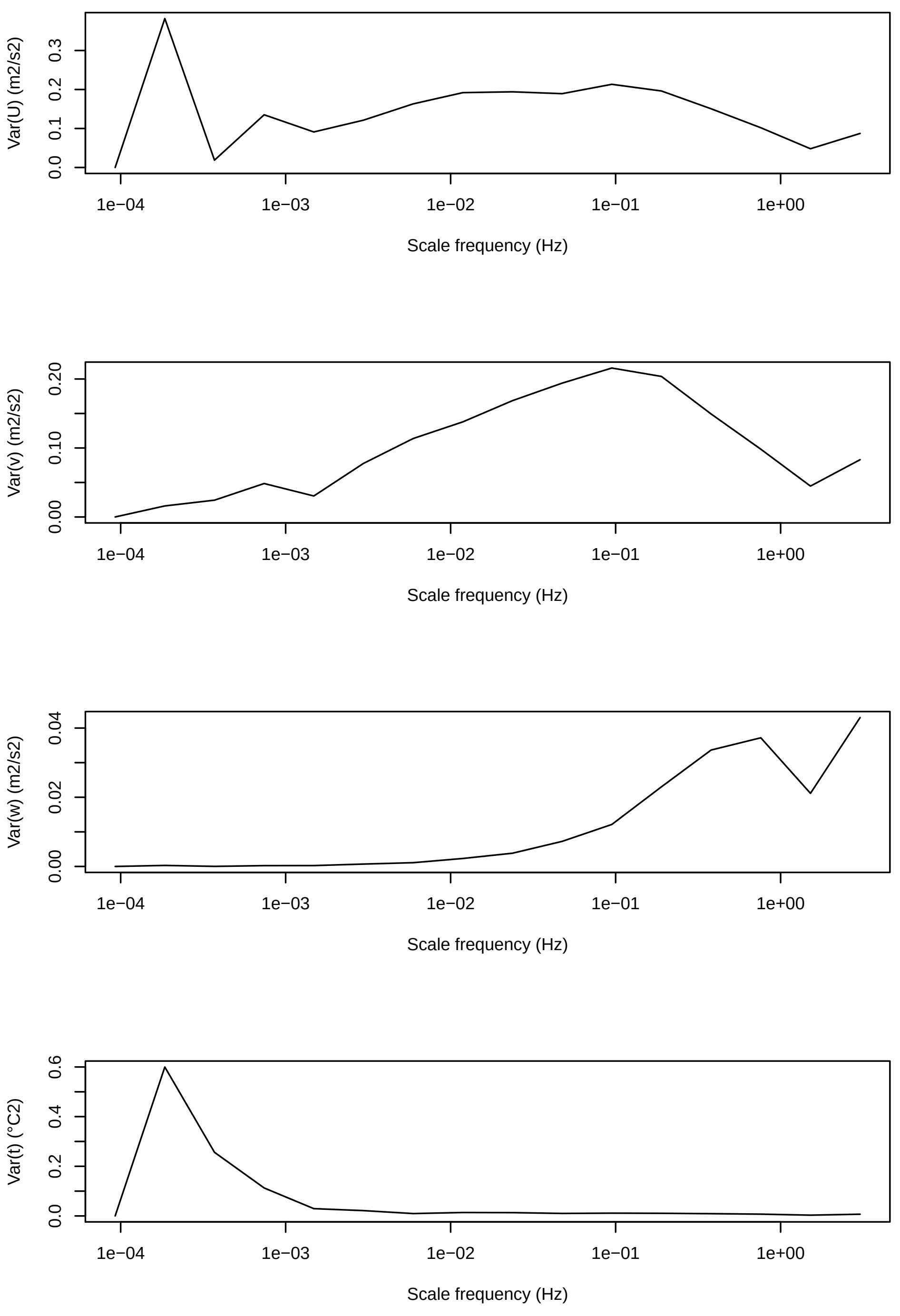


Figure 7.2: Multi-scale variances of Ameriflux Day 104, from 12:00:00 to 15:00:00

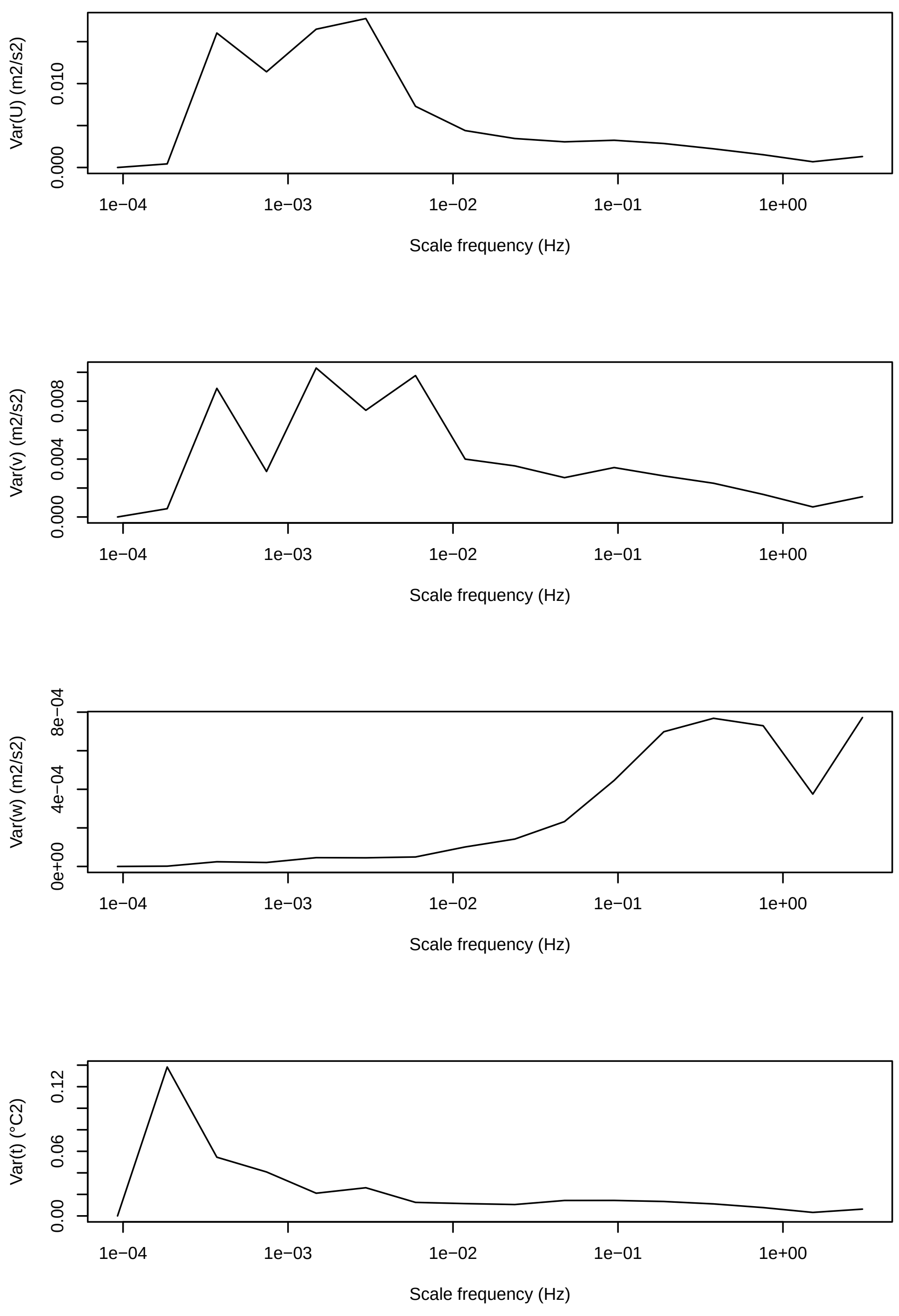


Figure 7.3: Multi-scale variances of Ameriflux Day 181, from 00:00:00 to 03:00:00

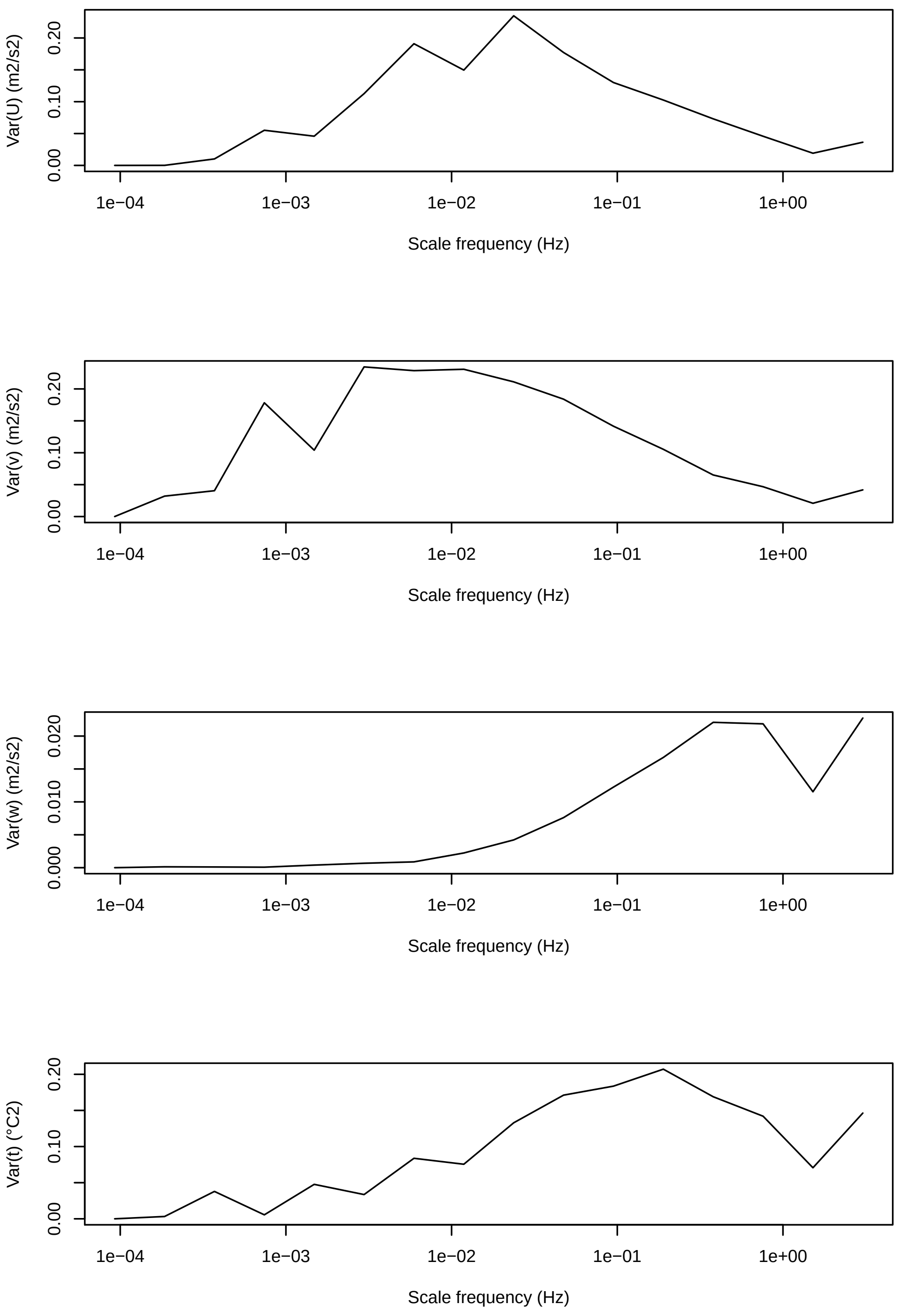


Figure 7.4: Multi-scale variances of Ameriflux Day 181, from 12:00:00 to 15:00:00

if so happens, whether this connection may be captured by a linear model. And if this makes sense, to identify and discuss it.

To frame our problem mathematically, we may take an appropriate multi-scale decomposition of our hourly data, apply it, compute variances, and then check them. But first, what "an appropriate multi-scale decomposition" means in our case? As we're interested in 10 minutes and 12 seconds means, these two time scales should be included in our analysis. This is indeed a simple request: 10 minutes are 600 seconds, and 12 divides 600. For ease of understanding we could use some other scales before, after and in-between. Here's one:

$$3600, 600, 60, 12, 1 \tag{7.51}$$

If we actually apply (7.51) to a hourly data set (for example, to a SonicLib file), then we arrive at a sequence of scale approximations we may nest using their cumulate form. at this point, remembering properties of variance in multi-scale decomposition, we may compare $\sigma^2_{\leq 600} = \sigma^2_{3600} + \sigma^2_{600}$ to $\sigma^2_{\leq 12} = \sigma^2_{3600} + \sigma^2_{600} + \sigma^2_{60} + \sigma^2_{12}$. That is, the variation contained in the scales up to 600 seconds to the variance at scales up to 12 seconds. Do they match?

A general answer can not be found, all situations being different. But, look the scatter plots in figures 7.5, 7.6 and 7.7.

In all the three cses shown (that is, everywhere but on day 104 of Ameriflux), data tend to cluster around a well defined line passing through the origin. So there *is* a linear model fitting the data, or at least a large part of them.

Here we've used water, and not a smely substance like, for example, hydrogen sulfide. This may seem a little contrived, but atmospheric water, along with carbon dioxide, is relatively easy to measure accurately using available (although expensive) technology like for example laser based open-path sensors. In addition water, as a gas, tends to behave as any other gas available in traces, and in case of at least some of the odor cases it acts as a transport agent. So using water is not that illogical as it might seem.

Our serendipitous finding should of course be confirmed on longer scale observations taken at a variety of sites. But its meaning is strikingly attracting: the standard deviation of 12 seconds concentration may be estimated quite accurately from 10 minutes averages, and this vary fact opens the way to using conventional models like Calpuff to draw *probabilistic* conclusions on concentrations. The road is quite long, but offers a method much beyond the actual guidelines used in various European countries and regions, based on applying hourly concentrations a sort of safety limit whose value cannot be defended scientifically. Besides, one in our research group (Stefano Cesco) is testing this idea in his thesis.

Detecting non-stationarity

Many data processing methods used in ultrasonic anemometry (one for all, eddy covariance - see [23]) rely on the assumption their input signals to be stationary. So it is not surprising that many stationarity testing methods have been developed in the past (see for example [18], [23], [5]).

Signal stationarity (see [27] and [20]) is very easy to assess using multi-scale analysis. A signal which is stationary in the weak sense at order two must exhibit constant first and second moments. This means that if we build a scale

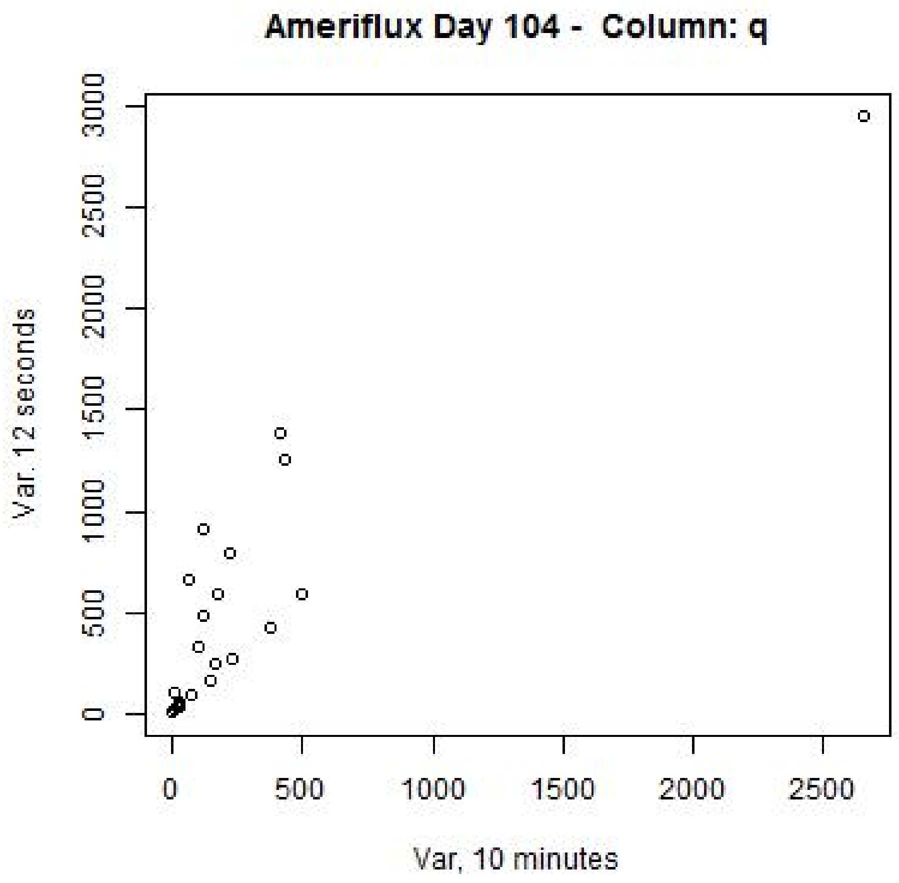


Figure 7.5: Variances up to 10 minutes compared to up to 12seconds of water concentration on Ameriflux Day 104

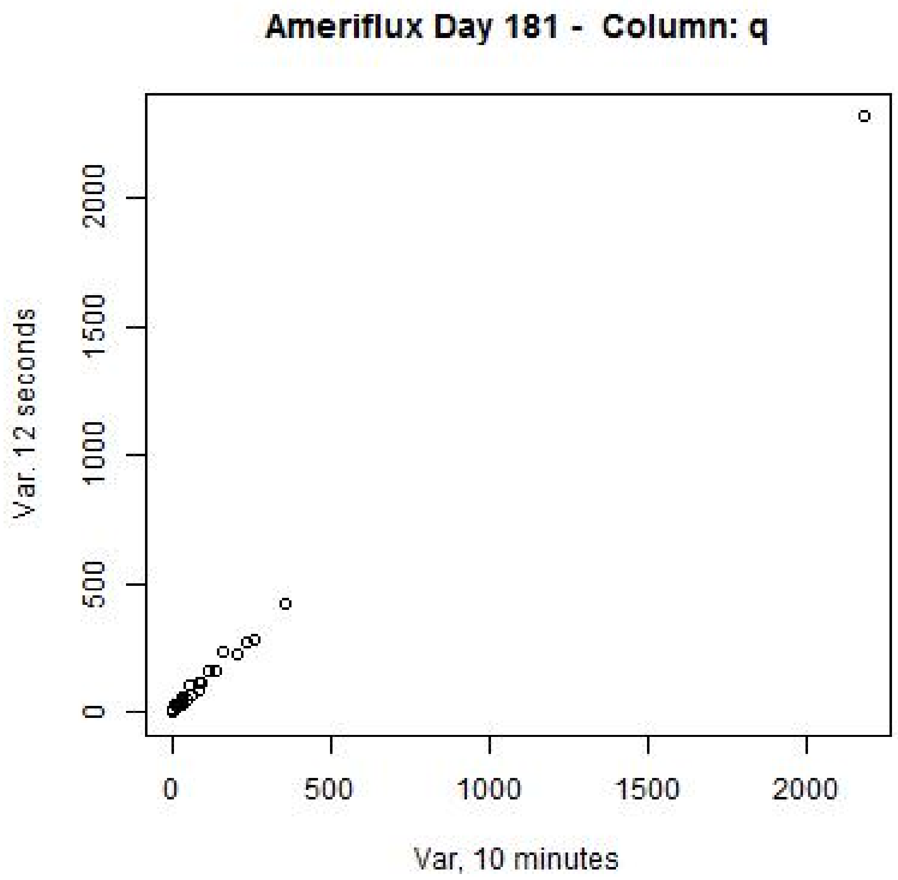


Figure 7.6: Variances up to 10 minutes compared to up to 12seconds of water concentration on Ameriflux Day 181

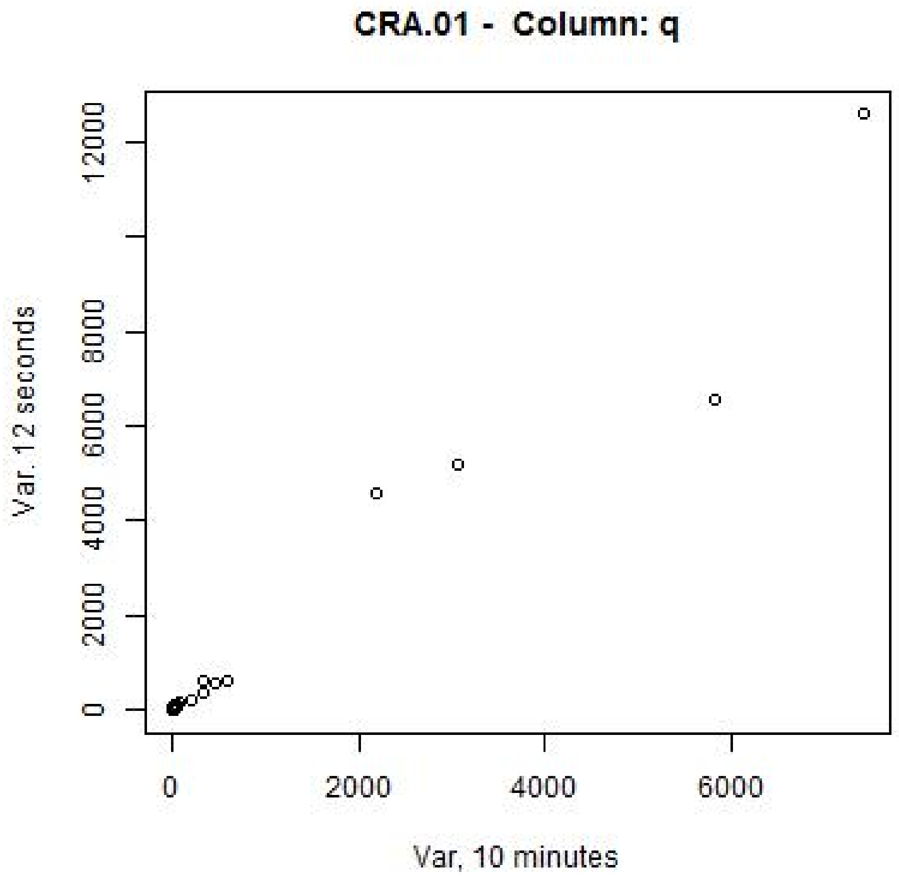


Figure 7.7: Variances up to 10 minutes compared to up to 12seconds of water concentration on CRA.01

approximation of it to a scale time longer than the reciprocal of acquisition rate we'll find it is nearly constant, so that most of the original variance in the signal is concentrated in the residual. As a consequence, the scale approximation's variance will be very small with respect to total variance. This offers the way to define an index, namely the ratio of Δt scale approximation related variance to the total signal variance. This number will be small for reasonably stationary signals, but close to 1 in cases of wild non-stationarity.

We can use multi-scale analysis not only to answer the question whether a signal is non-stationary, but also to find at what time scale does non-stationarity occur most.

Figure 7.8 shows a case of very strong non-stationarity. As time scale variances show, most variation occurs at the 10 minutes scale due to a steady, almost linear-type decrease over all time. Conversely, figure 7.9 presents a case when concentration is quite stationary.

The difference in time scale variance is striking. In the former case, all of it is concentrated in the first scale not encompassing the whole time interval, while in the second most variation is in the residual.

Turbulent kinetic energy

One interestng turbulence indicator is the TKE, shorthand of turbulent kinetic energy. This is defined by relation

$$TKE = \frac{1}{2}\left(\sigma_u^2 + \sigma_v^2 + \sigma_w^2\right) \tag{7.52}$$

The interest of TKE stems from its invariance with respect to rotation, so that it stays unchanged during the critical step of eddy covariance. Rotation invariance is itself an immediate consequence of the TKE being half the trace of covariance matrix, and the fact known from linear algebra that the trace of a matrix is invariant under similarity by an orthogonal matrix of same order (a convoluted mathematical way to say "rotation").

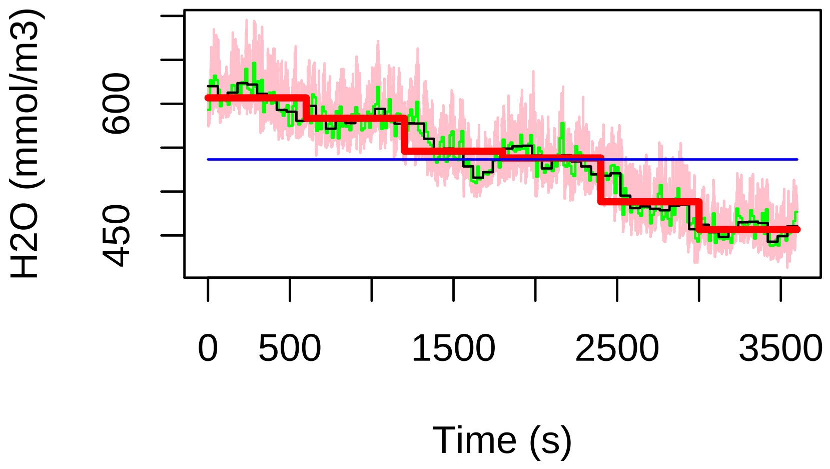
As a sum of variances, the TKE may be easily subjected to multi-scale analysis, using the halving method to generate automatically the time scales of interest. By doing so we may partition TKE in scale components, a nice way to discover at what time scales does most of turbulent kinetic energy occurs.

Figures 7.10 and 7.11, obtained with scale sequence

$$3600, 1200, 400, 200, 100, 50, 25, 12.5, 6.25, 3.125, 1.5625, 0.78125, 0.390625 \tag{7.53}$$

show two different cases, the former stable and the latter convective. Compared, these two situations appear strikingly different, each representative of a distinct extreme pattern. In "pure stable" hours most TKE concentrates on longer time scales, while an important part of TKE resides in the residual time scale in "pure convective" cases, with a large contribution from other scales - and possibly a local maximum on some intermediate scale.

Assuming terrain is reasonably flat, so that the geoid vertical can be assumed to approximate the fluid-mechanical vertical well, an interesting question is whether the contribution to TKE is larger on the horizontal plane or the vertical direction.



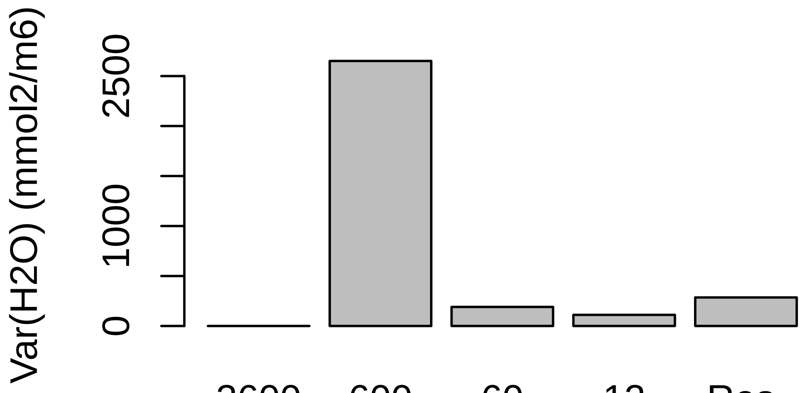
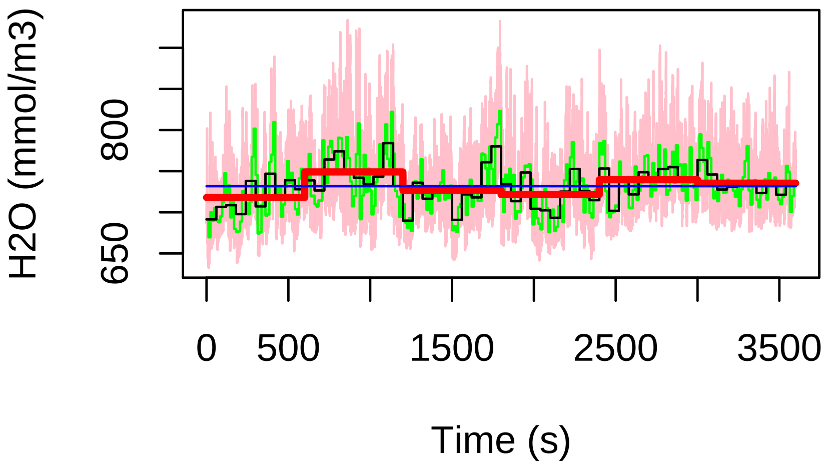


Figure 7.8: Multi-scale analysis of hour 16:00:00 to 17:00:00 in Ameriflux Day 104



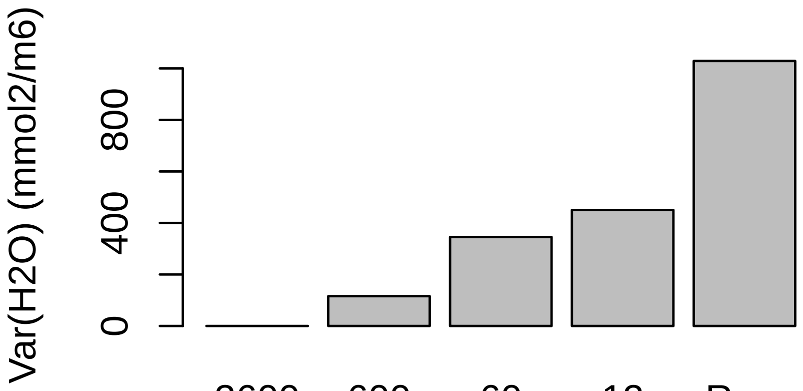


Figure 7.9: Multi-scale analysis of hour 12:00:00 to 13:00:00 in Ameriflux Day 104

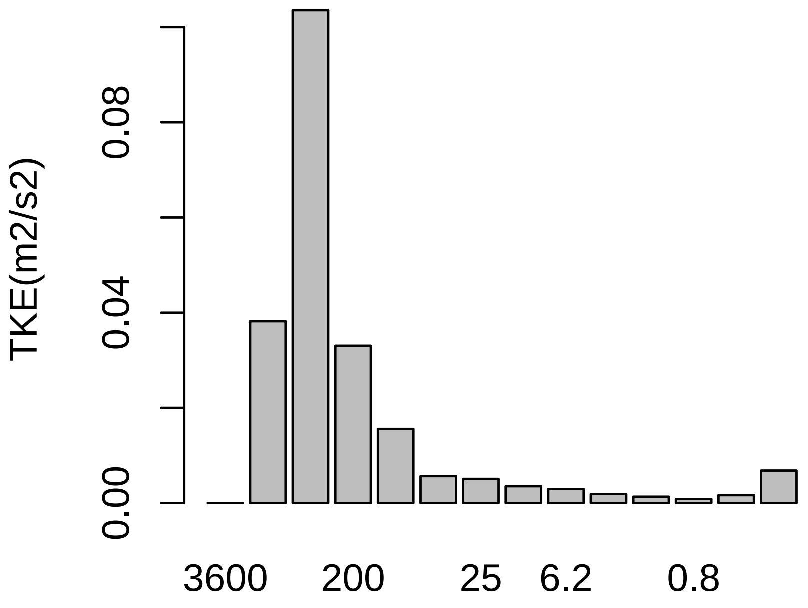


Figure 7.10: Multi-scale analysis of TKE on hour 05:00:00 to 06:00:00 in Ameriflux Day 104

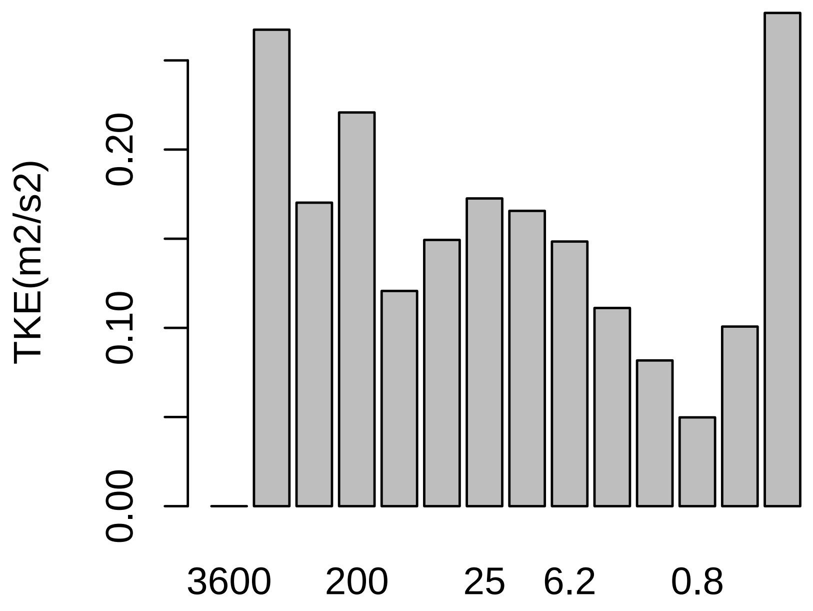


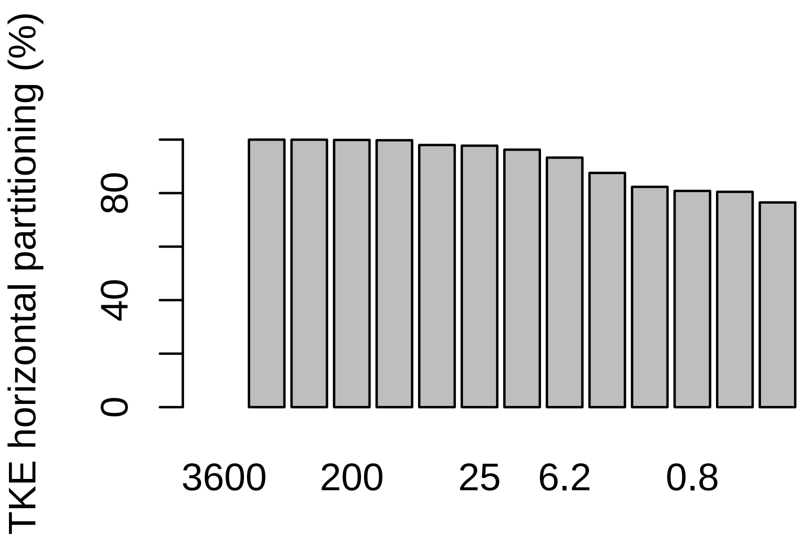
Figure 7.11: Multi-scale analysis of TKE on hour 12:00:00 to 13:00:00 in Ameriflux Day 104

In view of this objective we may introduce a "horizontal" TKE, $TKE_h = 0.5\left(\sigma_u^2 + \sigma_v^2\right)$, and a vertical TKE, $TKE_w = 0.5\sigma_w^2$. Of course $TKE = TKE_h + TKE_w$, that is, the TKE partitions in a horizontal and a vertical component. In non-flat terrain the same argument is valid, but "rotated" sigmas are due. We know already the TKE is a rotation invariant, so its value will not change under rotation, but its partitioning will.

Figures 7.12 and 7.13, obtained with same scale sequence nd at same times as 7.10 and 7.11, show what happens. In both cases a tendency is quite clear for the horizontal quota of turbulent kinetic energy to grow as scale time decreases, consistently with the intuitive feeling the vertical direction being the one where thermal exchanges (naturally pointing upwards or downwards as circumstances demand).

Interestingly, however, TKE along horizontal directions is dominant, however. This is a repeating pattern, with an ample variability if we use as "experimental" data the residual variance.

7.3 Checking Average Plausibility



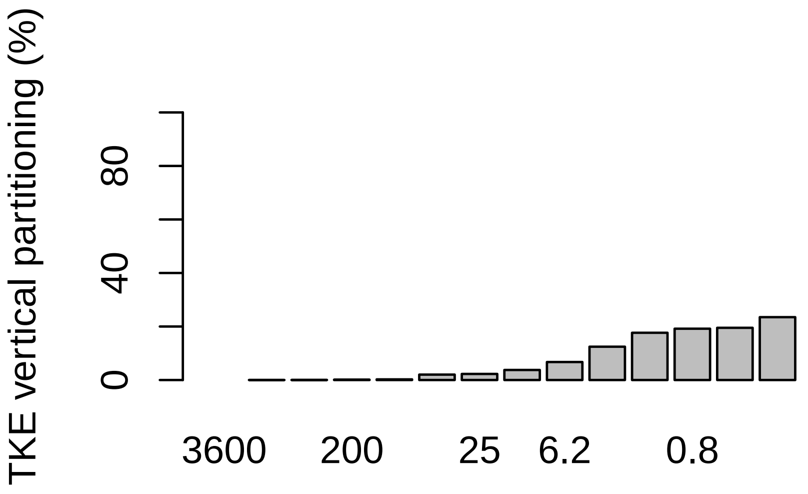
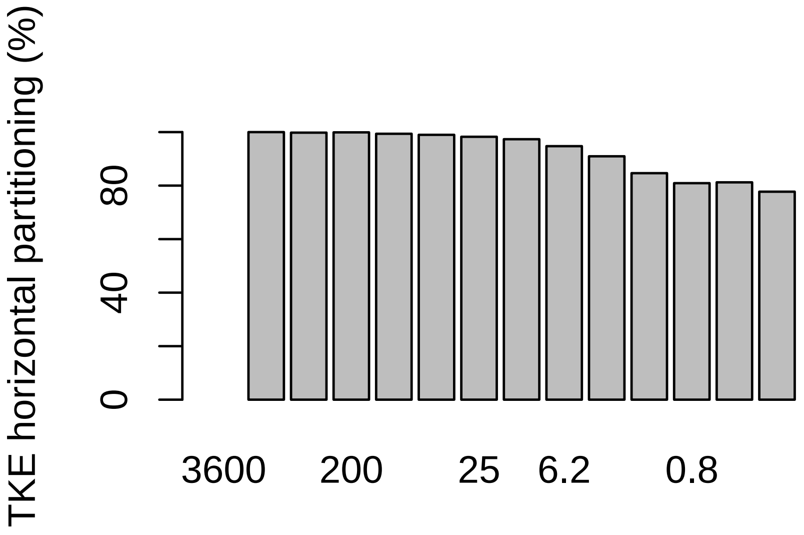


Figure 7.12: Multi-scale horizontal and vertical analysis of TKE on hour 05:00:00 to 06:00:00 in Ameriflux Day 104



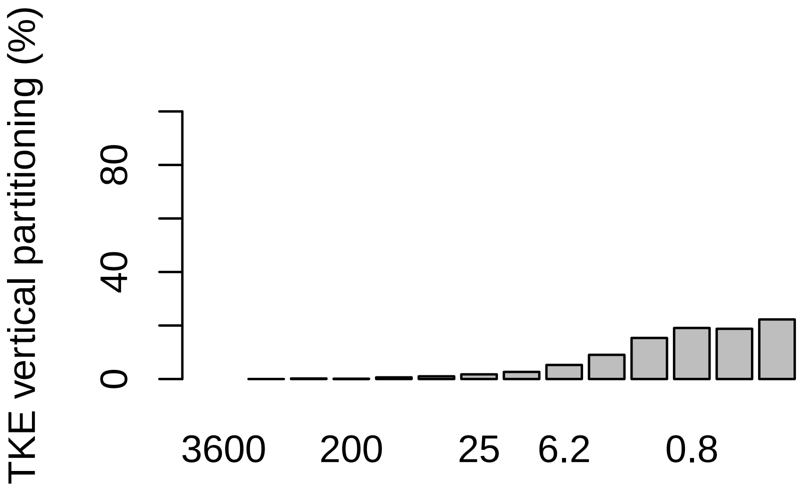


Figure 7.13: Multi-scale horizontal and vertical analysis of TKE on hour 12:00:00 to 13:00:00 in Ameriflux Day 104

Chapter 8

Giving Data a Meaning

8.1 Eddy Covariance

An important part of ultrasonic anemometer data processing has to do with computing fluxes of matter and energy, in contexts as diverse as air quality monitoring and ecosystems ecology. As different as they may be, all these application have one thing in common: they rely on some form of eddy covariance.

The SonicLib function implementing eddy covariance is

```
eddy.covariance(
  d,
  station.altitude,
  anemometer.height,
  mode,
  third.rotation.angular.threshold,
  verbose
)
```

where

d is an object of type `sonic.avg.data` on which eddy covariance calculations are to be made.

station.altitude is the station height above mean sea level, in meters (no default).

anemometer.height is the anemometer height above ground level, in meters (default=10).

mode is a string specifying the axis rotation method to use; possible values are "eddy.covariance" (classical eddy covariance with 2 axis rotations, also the default), "eddy.covariance.3" (eddy covariance with three axis rotations) and "planar.fit".

third.rotation.angular.threshold Positive real number which is compared to third rotation angle. If the absolute value of third rotation angle is found larger than threshold value the third rotation is not performed. Default: 10. Expressed in degrees.

verbose is a boolean flag indicating whether output messages should be printed (TRUE) or not (FALSE).

In case of error a result value of NULL is returned. In case of successful return, an object of type `sonic.eddy.cov.data` is returned.

8.1.1 Preliminary estimates

An important part of "eddy covariance" consists in applying similarity and other relations on mean and covariance values. Some of these formulae require data like air density which, although not measured directly using a sonic anemometer, may be estimated with good accuracy.

These estimations may be computed at any time before their use. In SonicLib we compute them almost immediately.

Pressure and dry air density

Dry air density, necessary to compute thermal and density-related corrections, is computed according to the formula

$$\rho_d = \frac{P_a}{R_s T_a} \tag{8.1}$$

where P_a is site absolute pressure in Pa, T the air temperature in K, and $R_s = 287.058$ is the specific gas constant in J/(kg · K). If more customary units are used for P_a (hPa) and for air mass (g instead of kg), then

$$\rho_d = 348.3617 \frac{P_a}{T} \tag{8.2}$$

As T is expressed in K, its value can be safely replaces by sonic temperature θ, itself approximating virtual temperature as shown further on. This way formula (8.2) simplifies to

$$\rho_d = 348.3617 \frac{P_a}{T_a} \tag{8.3}$$

where $T_a = \theta + 273.15$.

Absolute pressure can not be measured using a sonic anemometer. But, it may be estimated once station height is known using hydrostatic equation

$$P_a = 1013 \exp\left(-0.0342 \frac{z}{T}\right) \tag{8.4}$$

where z is station altitude, and T the absolute temperature in K. Using considerations similar to the ones developed for density, in SonicLib we simplify this definition as

$$P_a = 1013 \exp\left(-0.0342 \frac{z}{T_a}\right) \tag{8.5}$$

where z is the station altitude above mean sea level.

ρC_p

The product $\rho_d C_p$ is approximated in SonicLib using similar consideration on absolute and sonic temperature as used above, yielding the following formula:

$$\rho_d C_p = 350.25 \frac{P_a}{T_a} \tag{8.6}$$

Pressure is assumed estimated using formula (8.5).

8.1.2 Axis rotation

In most real-world site the condition $\overline{w} = 0$ is not satisfied, and appropriate reference system rotation must be accomplished. This is made in SonicLib in two ways: according to the classical eddy covariance scheme (that is, on single averaging time basis) and using planar fit (on a whole campaign, or an important part of it).

Classical eddy covariance

In classical eddy covariance axis rotation is individually accomplished on each average. In most cases two rotations are done. In special circumstances a third may be done.

First two rotations The first two rotations are aimed at changing the reference system so that its new x axis points in the direction of mean wind. The overall rotation matrix, $\mathbf{R}_{02}$, is defined as the product of two separate elementary rotations,

$$\mathbf{R}_{02} = \mathbf{R}_{12} \cdot \mathbf{R}_{01} \tag{8.7}$$

where

$$\mathbf{R}_{01} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.8}$$

is a rotation around the vertical axis with

$$\sin\alpha = \frac{v}{\sqrt{u^2 + v^2}} \tag{8.9}$$

and

$$\cos\alpha = \frac{u}{\sqrt{u^2 + v^2}} \tag{8.10}$$

thus aligning the transformed axis x to point in the direction of horizontal wind.

The second rotation matrix complete this work, by rotating the new reference around the new y axis to align the transformed x axis to full three dimensional wind:

$$\mathbf{R}_{12} = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \tag{8.11}$$

where

$$\sin\beta = \frac{w_1}{\sqrt{u_1^2 + w_1^2}} \tag{8.12}$$

and

$$\cos\beta = \frac{u_1}{\sqrt{u_1^2 + w_1^2}} \tag{8.13}$$

(here we use the symbols u_1, v_1 and w_1 to designate the three wind components in the reference obtained by applying the first rotation).

In SonicLib the two rotations are computed sequentially and applied individually to mean wind and scalar covariances (assimilable to vectors) and wind covariance matrix. The actual formulae used are:

$$\mathbf{v}_1^T = \mathbf{R}_{01} \cdot \mathbf{v}^T \tag{8.14}$$

$$\gamma_1^T = \mathbf{R}_{01} \cdot \gamma^T \tag{8.15}$$

and

$$\mathbf{C}_1 = \mathbf{R}_{01} \cdot \mathbf{C} \cdot \mathbf{R}_{01}^T \tag{8.16}$$

where $\mathbf{v} = [u, v, w]$ represents wind vector, $\gamma = \left[\overline{u'x'}, \overline{v'x'}, \overline{w'x'}\right]$ the "vector" of covariances of wind components with a generic scalar x, and $\mathbf{C}$ the wind covarianve matrix.

Second rotation is then applied on the result of the first, using the following relations.

$$\mathbf{v}_2^T = \mathbf{R}_{12} \cdot \mathbf{v}_1^T \tag{8.17}$$

$$\gamma_2^T = \mathbf{R}_{12} \cdot \gamma_1^T \tag{8.18}$$

and

$$\mathbf{C}_2 = \mathbf{R}_{12} \cdot \mathbf{C}_1 \cdot \mathbf{R}_{12}^T \tag{8.19}$$

with an intuitively extended meanings of symbols.

Third rotation It should be easy to understand that the plane containing axes xz is still perpendiculat to the horizontal plane after two rotations. This means the w_2 axis does still *not* points to the the "fluid-dynamical vertical". For this to happen a third rotation should be made around the new x axis, with matrix

$$\mathbf{R}_{23} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\delta & \sin\delta \\ 0 & -\sin\delta & \cos\delta \end{bmatrix} \tag{8.20}$$

where

$$\delta = \frac{1}{2}\arctan\left(2\frac{\overline{v_2'w_2'}}{\overline{v_2'v_2'} - \overline{w_2'w_2'}}\right) \tag{8.21}$$

Performing third rotation aligns the vertical of transformed reference to the true fluid-dynamical vertical. On the other side of this desirable fact it often happens the third rotation to be affected by error. To prevent excessive impact, thethird rotation is really executed if the rotation angle is lower than a threshold whose default value is 10 degrees, and which may be altered by the programmer as shown when documenting the call.

If third rotation angle absolute value is smaller than the threshold angle, then rotation matrix is computed and applied:

$$\mathbf{v}_3^T = \mathbf{R}_{23} \cdot \mathbf{v}_2^T \tag{8.22}$$

$$\gamma_3^T = \mathbf{R}_{23} \cdot \gamma_2^T \tag{8.23}$$

and

$$\mathbf{C}_3 = \mathbf{R}_{23} \cdot \mathbf{C}_2 \cdot \mathbf{R}_{23}^T \tag{8.24}$$

Planar fit

As an alternative to classical eddy covariance, in which rotations are computed individually on any averaging interval, the *planar fit* method has been proposed which performs an identification of the fluid-dynamical vertical from the global behavior of many averages (typically an entire measurement campaign).

Planar fit also performs three rotations, but in a different order than classical covariance: the first two, around axes x and y, transforms the reference so that its new vertical coincides with the fluid-dynamical vertical. The third rotation, around transformed axis z, aligns axis u with mean wind.

Apart the different order of rotations, the most important difference between planar fit and classical rotation is that first two rotations are computed once forever from all campaign data, while the third is performed on each averaging interval.

To determine the first two rotations model

$$w = b_0 + b_1 u + b_2 v + \varepsilon \tag{8.25}$$

is fitted to experimental data, assuming ε to be a normally distributed random variables with zero mean and a given variance. In SonicLib fitting is determined using the standard `lm` linear least squares model, assuming equal variance for u and v determinations.

With best fitting values b_0, b_1 and b_2 at hand, the following values are computed:

$$\sin(\beta) = -\frac{b_1}{\sqrt{b_1^2 + b_[^2 2] + 1}} \tag{8.26}$$

$$\cos(\beta) = \frac{\sqrt{b_1^2 + 1}}{\sqrt{b_1^2 + b_[^2 2] + 1}} \tag{8.27}$$

$$\sin(\delta) = \frac{b_2}{\sqrt{b_1^2 + 1}} \tag{8.28}$$

and

$$\sin(\delta) = -\frac{1}{\sqrt{b_1^2 + 1}} \tag{8.29}$$

The first two rotation matrices are then formed as

$$\mathbf{R}_{12} = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \tag{8.30}$$

and

$$\mathbf{R}_{23} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\delta & \sin\delta \\ 0 & -\sin\delta & \cos\delta \end{bmatrix} \tag{8.31}$$

The first two rotation combined matrix is then computed as

$$\mathbf{R}_{13} = \mathbf{R}_{23}\mathbf{R}_{12}$$

and applied to non-rotated data in whole set as we've seen in section 8.1.2.

In planar fit, third rotation is computed individually for all data in set, with rotation matrix defined as

$$\mathbf{S} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.32}$$

where $\alpha = \arctan\frac{v_2}{u_2}$, u_2 and v_2 being the horizontal wind mean components after having applied the first two rotations. Thirs rotation is then applied the usual way.

Notice how performing the three rotations in a different order than in classical eddy covariance changes in general the three angles respect to their respective eddy covariance values, but does not modify the final result.

8.1.3 Scalar fluxes and the WPL correction

Because of the way SonicLib standard raw files are defined, all scalar concentrations are expressed as molar concentrations, that is in mmol/m^3. On return, mass densities and molar concentrations are used for fluxes, and this requires different conversions and some non identical details in how Webb-Pearman-Leuning corrections (see [13]) are formulated.

SonicLib implementation follows the path described in chapters 3 and 4 of [23]. Passages are a bit delicate, and their assessment is then made possible by storing in output data all relevant intermediate results. Advanced users may employ these as a quality assurance tool in addition to more traditional stationarity tests.

We start, by writing the expression of the fluxes F_s and $F_{s.mol}$ of a scalar s expressed in density and molar concentration form respectively. Notation is derived from [23], with some minor simplifications. The mass flux is

$$F_s = m_s\overline{w'c'_s} + m_s\frac{\overline{c_s}}{\overline{c_d}}\left[\left(\overline{c_d} + \overline{c_v}\right)\frac{\overline{w'\theta'}}{\overline{\theta}} + \overline{w'c'_v}\right] \tag{8.33}$$

Gas / Element	Molar mass m_s in g/mol
Water	18.0153
Carbon dioxide	44.0100
Ammonia	17.0305
Methane	16.0425
Hydrogen	1.00794
Oxygen	15.9994
Carbon	12.0107
Nitrogen	14.0067

Table 8.1: Molar mass of gases known to SonicLib and their component chemical elements

while the molar flux is

$$F_{s.mol} = \overline{w'c'_s} + \frac{\overline{c_s}}{\overline{c_d}} \left[(\overline{c_d} + \overline{c_v}) \frac{\overline{w'\theta'}}{\overline{\theta}} + \overline{w'c'_v} \right] \tag{8.34}$$

As it turns out by comparing (8.33) with (8.34), a simple relation exists between mass and molar fluxes:

$$F_s = m_s \cdot F_{s.mol} \tag{8.35}$$

where m_s is the molar mass of the gas considered in units of g/mol. Table 8.1 states the molar masses of the gases known to SonicLib.

8.1.4 Sensible heat fluxes H_{0v} and H_0

It is well known that sonic temperature θ is connected to dry bulb temperature T by relation (see [22] and [16])

$$\theta = T \cdot (1 + 0.51q) \tag{8.36}$$

and a very similar relation to hold for virtual temperature,

$$\theta_v = T \cdot (1 + 0.61q) \tag{8.37}$$

where q represents the mass mixing ratio of water to air.

Taking the difference between these temperatures we get

$$\theta_v - \theta = 0.1Tq \tag{8.38}$$

whose value is very small. As a consequence, in sonic anemometry it is commonly assumed that $\theta = \theta_v$.

If this is accepted for true, then the same will happen to any quantity containing sonic temperature, namely the covariance $\overline{w'\theta'}$, which may be safely interpreted as $\overline{w'\theta'_v}$. This covariance, multiplied by ρC_p, yields the *surface buoyancy flux* (see [26]):

$$H_{0v} = \rho_d C_p \overline{w'\theta'_v} \tag{8.39}$$

If all we have is a sonic anemometer we can say no more, and use (8.39) as an approximation to the actual *sensible heat flux*

$$H_0 = \rho C_p \overline{w'T'} \tag{8.40}$$

If we have water in our data set, however, a correction may be used (see [11] and [22]) in the form

$$\overline{w'T'} = \overline{w'\theta'} - 0.51\overline{T} \cdot \overline{w'\chi'} + 2\frac{\overline{T}\overline{u}}{\overline{c^2}}\overline{u'w'} \tag{8.41}$$

where χ is water mass mixing ratio. In equation 8.41 T represents an absolute temperature, whose numerical value may be safely replaced by $T_a = \theta + 273.15$.

As pointed out in [25] the term $2\frac{\overline{u}}{403}\overline{u'w'}$ only applies to vertically directed measuring path, which is not always the case with 3D ultrasonic anemometers. Variations have been proposed for this term taking into account the full geometry of instrument, with different constants used in different sonic anemometers. Most often on these days the revised term is inforporated in sonic anemometer firmware, and needs not to be applied. This given, in SonicLib the following simplified relation is used:

$$\overline{w'T'} = \overline{w'\theta'} - 0.51\overline{\theta} \cdot \overline{w'\chi'} \tag{8.42}$$

8.1.5 Latent heat flux H_e

The latent heat flux is computed accordingly to [22] as

$$H_e = \frac{\lambda}{1000}\overline{w'q'} \tag{8.43}$$

where $\overline{w'q'}$ is expressed in g/m^{2}s and

$$\lambda = 2500.8 - 2.36\theta + 0.0016\theta^2 - 0.00006\theta^3 \tag{8.44}$$

is latent condensation heat (note θ is expressed in Celsius degrees, not K).

8.1.6 Examples

Classical eddy covariance "the normal way"

The first step consists in a quite standard processing, the kind one most often does, aimed at getting results in the shortest possible time.

In this example we use the "CRA.01" data set (4.1). But instead of working on it directly in SonicLib standard raw form, we begin from start: a nice way to demonstrate an entire workflow. As shown in section 4.1 the original data have been collected using a special purpose data acquisition system, written using National Instrument's LabVIEW and operating on an industrial PC.

The resulting files bear very little resemblance to SonicLib files, and must then be converted. This is accomplished outside SonicLib using specially designed "adapter" software.

The actual conversion is quite involved, reflecting the complexity of the original format. This originates from the data acquisition system, which on each read encodes textual data from sonic anemometer to binary form. Data are then saved, but as the system has been written in LabVIEW they are not stored as (say) regular Intel binary numbers. They are stored as LabVIEW binary instead, and this turns out a big difference: LabVIEW binary integers are encoded in big-endian form (most significant byte first in memory) while Intel binary numbers are encoded in little-endian form (least significant byte first in memory).

Converting between the two forms is straightforward (it is sufficient to swap bytes in any number), yet needs to be taken care of, and this in turn demands using a few low-level operations.

The first conversion step is performed using program `decode`, also released as public domain thanks to Servizi Territorio srl, partner of SonicLib team, who wrote it (you may find the source in CRA.01 data set directory).

Use of `decode` is simple, once some basic knowledge of data collection text is known. The program is run from the command line as

```
decode <input> <type> <output>
```

where `<input>` designates the input file, `<output>` the output file name, and `<type>` an integer number specifying the file type. And this is precisely the first thing to know: what the type is.

The types supported by the `decode` utility are:

1 Meteoflux V3.x format (a binary format use on an old data acquisition system made by Servizi Territorio srl).

2 LabVIEW-based data acquisition system, with 32 bit integers used to encode sonic raw data (aka “LabVIEW long”).

3 LabVIEW-based data acquisition system, with 16 bit integers used to encode sonic raw data (aka “LabVIEW short”).

Original data in CRA.01 set are in LabVIEW short form. How do we know it? The ideal answer would have been “Oh, because thanks to a rigorous configuration management process we know exactly which format the data are, for any specific file ever collected.”

But this is not the case! Data acquisition systems have been modified quite often, to follow the quick evolution of sonic anemometers. The result is a huge set of campaign data collected in many different places with different systems. But fortunately, there is a simple method to say with absolute certainty which form files are, by just “looking at file size”. We know each CRA.01 file is 721 kByte long, and that all LabVIEW short files are exactly 721 KByte each. This means CRA.01 original data are in LabVIEW short form.

When `decode` is launched, it produces an output file which still is not in SonicLib form, but in Meteoflux Core text form. This is still another proprietary format by Servizi Territorio srl, but with a decisive advantage over LabVIEW short form: it’s human readable! And, recent enough to devise its own adapter, not in SonicLib library but anyway as part of the SonicLib project: `mfc1`.

Similarly to `decode`, `mfc1` operates as a command-line utility. Torun it, just type

```
mfc1 <input> [<cvt>] <output>
```

where

Value of "Meanings"	Actual meanings, in plain form
q	Water, in mmol/m^3
c	Carbon dioxide, in mmol/m^3
a	Ammonia, in mmol/m^3
m	Methane, in mmol/m^3
temp	Air temperature, in Celsius degrees
hrel	Relative humidity, in %

Table 8.2: Values of "Meanings" field known to SonicLibto date

¡input¿ designates an input file in Meteoflux Core textual form (as produced by `decode`).

¡cvt¿ is a conversion file (its being surrounded by square brackets indicates it's *optional*). More on later: it's a very important file. If omitted, no conversion is performed on analog data, if any.

¡output¿ finally designates an output file, this time in SonicLib form.

All interesting behavior of `mfc1` is specified by the contents of the ¡cvt¿ file. This is a text file containing, for each of the analog columns to convert, the following information:

Channel Channel number to which the instrument provoding the data has been physically connected (1 to 14 for Meteoflux Core using Metek USA-1 anemometer).

Multiplier Coefficient m in conversion relation $m \cdot x + q$.

Offset Term q in conversion relation $m \cdot x + q$.

Meanings A string, containing the short name of the quantity gathered. The possible values are given in table 8.2.

Filling in the details of conversion file demands deep knowledge of the ultrasonic anemometer, the other sensors connected, and collection process. Nothing really difficult: just something to be done with care.

In the specific case CRA.01 data set instrument configuration and conversion data have been described extensively in section 4.1.3. Here we limit to use the values there computed:

H_2O Multiplier = 0.0610370, Offset = 0.

CO_2 Multiplier = 0.000610370, Offset = 10.

Since water and carbon dioxide channels from the LI-7500 have been connected to analog inputs no. 7 and 8 of USA-1, the following concents may be used for the conversion file:

```
7 0.0610370 0.0 "q"
8 0.000610370 10.0 "c"
```

Using `decode` and `mfc1` may be tedious, as both expect users to specify single files. They may be invoked however from suitable scripts (e.g. in Python) to automate file production within a directory. These script may well be ad-hoc and very small, or as complex as full-fledged software packages. One of such automation scripts is given here as just an example - and a sub-optimal one indeed:

```
#!/usr/bin/python

import glob
import sys
import os

files = glob.glob("*.*r")

for file in files:
out = file[0:len(file)-1]+".csv"
os.system("./mfc1 "+file+" cra01.cfg "+out)
```

This script converts automatically all files with names terminating with one character 'r' (that is, a lowercase r) to SonicLib raw data files in the same directory. Once execution has terminated, SonicLib standard files for raw data may be copied manually to data source directory, in our case `<SonicLib_Root>/TestData/CRA.01`.

Up to now, we have performed no more than conversion of data: maybe a little tedious task, yet simple after all. Really interesting things begin here.

The very first step consists in translating the huge mass of SonicLib files to averages and covariances, in preparation to eddy covariance. To perform this step we may use a function we've met already, `average.sonic.file.set`, with a call like this:

```
d <- average.sonic.file.set(
  dir.name="../TestData/CRA.01",
  verbose=TRUE
)
```

As you see, we in practice accepted all defaults. Part of them are already consistent with data as they are (like for example data rate). Others, namely the averaging period, have a default which accidentally concides with the one we want for our case: 30 minutes. A special mention goes to delay, which we left untouched: this is a good thing, the 0.3s default being consistent with how the Li-7500 sensor has been programmed on field.

Specifying `verbose=TRUE` enables printing status and error messages, and may be helpful for an interactive run like the one we're running in this example. Within an automated processing procedure, specifying `verbose=TRUE` may be quite annoying, and leaving the default might be more acceptable. No fixed rule exists, however, and anyone may proceed as their taste suggests.

At end of processing we have in our hands a new object, `d`, of type `sonic.avg.data`, *or* of value `NULL` if something goes awry. From the description given in section 6.5 we know this *may* happen, and a check is then worth.

In line of principle, verifying if a value is NULL without printing the whole of it (which may be unreasonably large) may be done in a very simple way:

```
is.null(d)
```

The result is a boolean value, TRUE or FALSE depending on the answer. Better even, we may give a command like

```
summary(d$data)
```

which in case of a NULL value yields just the NULL, but upon successful return gives a wealth of interesting statistics about the results obtained.

In our case execution has completed successfully, so `d` is really not NULL. As a very simple test we just plot the average wind speed, sonic temperature, water and carbon dioxide. Results, shown in figures reffig:CRA.01 Vel, 8.2, 8.3 and 8.4, are quite heartwarming: as small as the CRA.01 sample is (1 day), plots show some dynamics: whatever we've done, instruments were not completely dead.

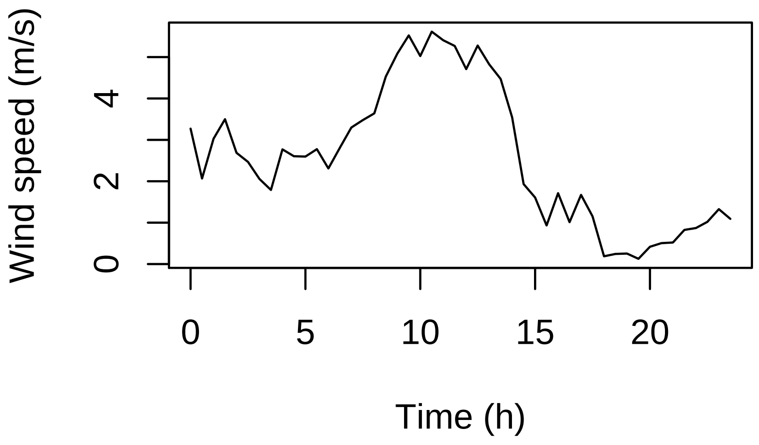


Figure 8.1: Evolution of wind speed in CRA.01 data set

The fact we have seen a dynamics of course does not mean data are correct, nor it implies they are suitable for eddy covariance (maybe because of important non-stationarity). We'll be back on these important issues, but for the moment we proceed contenting ourselves with minimal evidence of quality.

The next step is, finally, eddy covariance. The most important decision we have to take at this point is its *type*. The data set is so short we can immediately rule planar fit out. But a choice is still possible between the two flavors of classical eddy covariance: with two, or three rotations?

To take this choice we may proceed in many ways. Two popular methods are:

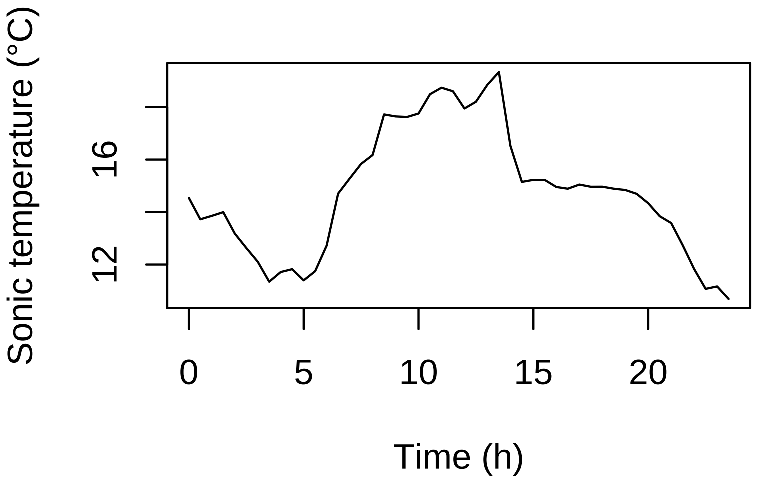


Figure 8.2: Evolution of temperature in CRA.01 data set

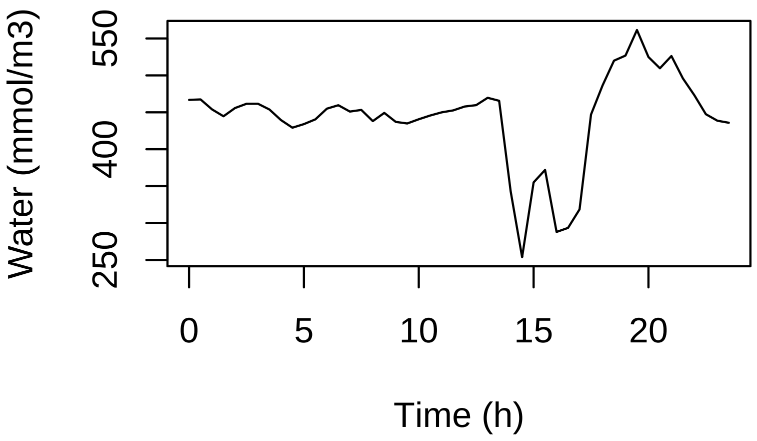


Figure 8.3: Evolution of water molar concentration in CRA.01 data set

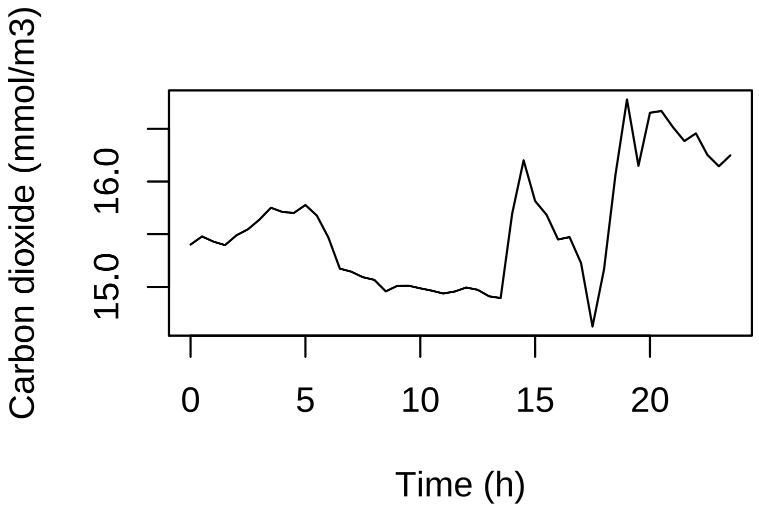


Figure 8.4: Evolution of carbon dioxide molar concentration in CRA.01 data set

- Try 2 and 3 rotations, and use the better looking result set. This way demands a method exists to decide which result set “looks better”, and once again many methods exist. A sensible one is to check key correlations arising from similarity and say the best result set is the one for which they get the best score.
- Use a priori knowledge and select the most appropriate type. For example in CRA.01 data set the flat experimental field suggests limiting to 2 rotations.

Notice some research groups do not recommend using 3 rotations (see for example ([23])).

In this example we adopt two rotations, which incidentally is the default value. This important decision taken, we may finally run eddy covariance:

```
e <- eddy.covariance(
  d,
  station.altitude=20
)
```

We may now have a look to what we obtained.

In figure 8.5 we may see the buoyancy flux. This is a very interesting indicator, yielding a first impression of sensible heat flux even if a fast water sensor is missing. It is negative on first (“leftmost”) night-time, and positive during daytime, as normally happens under daytime convective conditions. On second

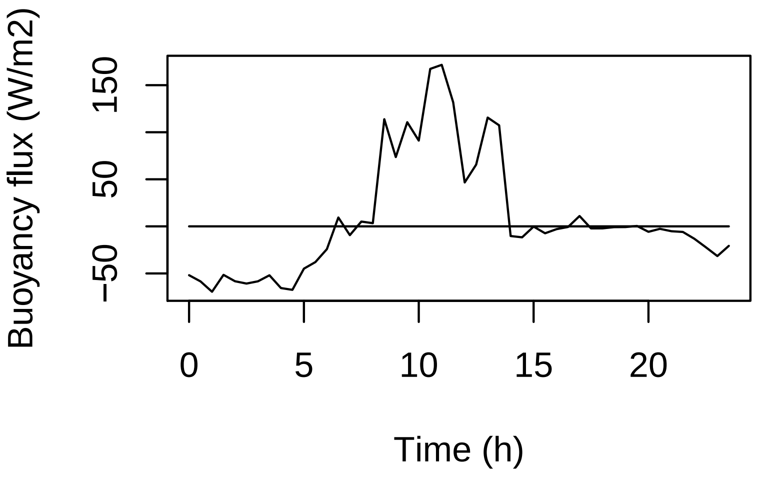


Figure 8.5: Evolution of buoyancy flux (aka "sensible heat flux" when a fast water sensor is missing) in CRA.01 data set

night, however, things change dramatically, with buoyancy flux moving just around the zero reference line. What happened?

Trying to reconstruct past meteorological events from a single variable is not more serious than figuring out an entire animal given a single tooth, 'a la Baron Cuvier. This is the reason field campaigns are accompanied by extensive meteorological observation data. Yet *something* may be conjectured, and results may sometimes be interesting.

In our case, a well known experimental fact is that sensible heat flux (and buoyancy flux) visually "copy" the time evolution of solar net radiation. On a sunny day, at an ideal rural site not affected by spurious shading, the net radiation varies smoothly from a small negative value on nighttime to a larger positive value on daytime, the maximum being reached at local noon.

Under identical conditions the same would happen to buoyancy flux. But figure 8.5 tells us an entirely different story: especially on daytime we have a distinct sawtooth pattern in the graph, which leads us to imagine clouds intercepting a lot of incoming shortwave radiation. If precipitation also occurred, the terrain has poor draining properties (or was already saturated by a preceding rain episode, or is waterlogged, or . . .) and rain was intense enough to saturate the ground, then we'd likely get a very small night-time ground cooling due to the large thermal capacity of water, and consequently the sensible heat flux would drop on nighttime. And the same would happen to buoyancy flux, a variant of it.

Figure 8.6 represents the sensible heat flux after correction of density and water flux effects have been removed. We may see changes from buoyancy flux

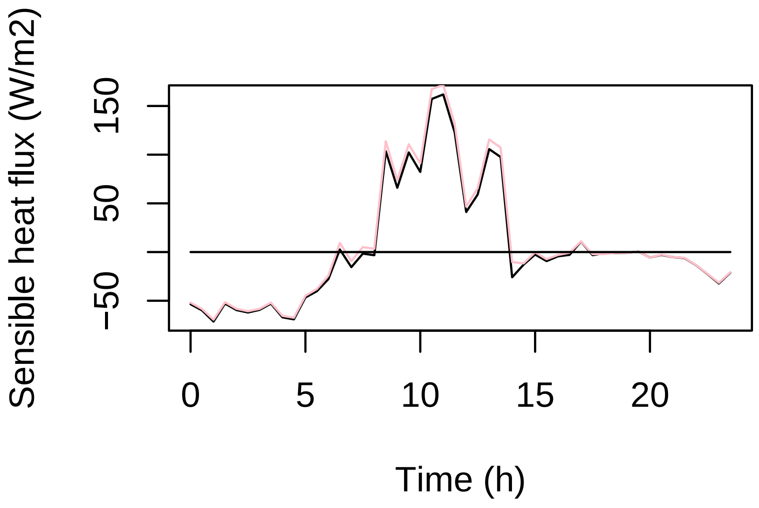


Figure 8.6: Evolution of sensible heat flux (in black) in CRA.01 data set; the pink continuous line shows, for comparison, the buoyancy flux

are quite small, and evident only on daytime. On the "leftmost" nighttime a bit of variation is also visible, but on the rightmost the two fluxes are virtually identical.

Figure 8.7 shows the *latent* heat flux. This graph is very interesting, since what we have already conjectured. On first nighttime and during daytime its value was always positive. On the second, rightmost, nighttime, instead, the sensible heat flux drops to zero.

Sensible heat flux tightly connects with the flux of water vapor, whose time evolution we see in figure 8.8. Both quantities indicate a massive peak afternoon, as if precipitation has fallen on a warm ground, with consequent large evaporation. Has relative humidity then grown to impede further evaporation on the following nighttime? Did ground saturate of water, increasing hugely its thermal capacity?

A bit of sane skepticism

The questions closing the preceding sections may remain unanswered, if no meteorological observations are done during a field campaign. This might be a pity, especially in ecology and agricolture, as it makes quite difficult to separate purely physical effects (e.g. due to regolith warming and cooling in absence of vegetation) and the response of a forest camopy or crop.

But at least we might resort to some detective work the kind we've seen. We can gain conjectures, not experimental proofs, but at least we have something.

The question then arises, whether what we have is really there, or not.

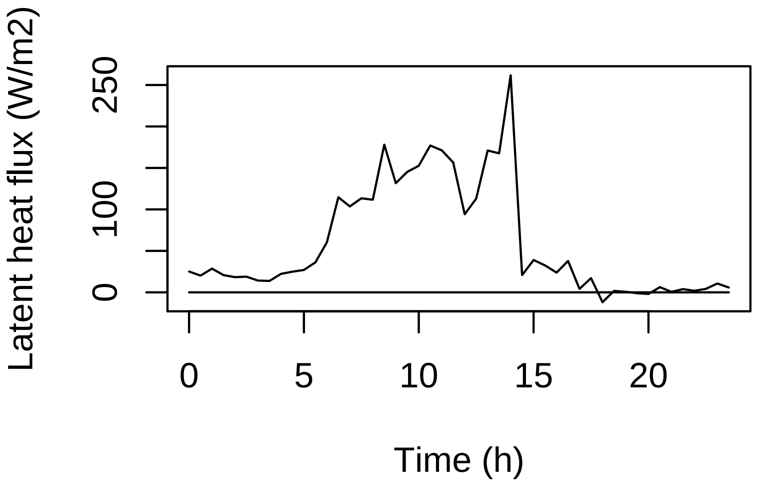


Figure 8.7: Evolution of latent heat flux in CRA.01 data set

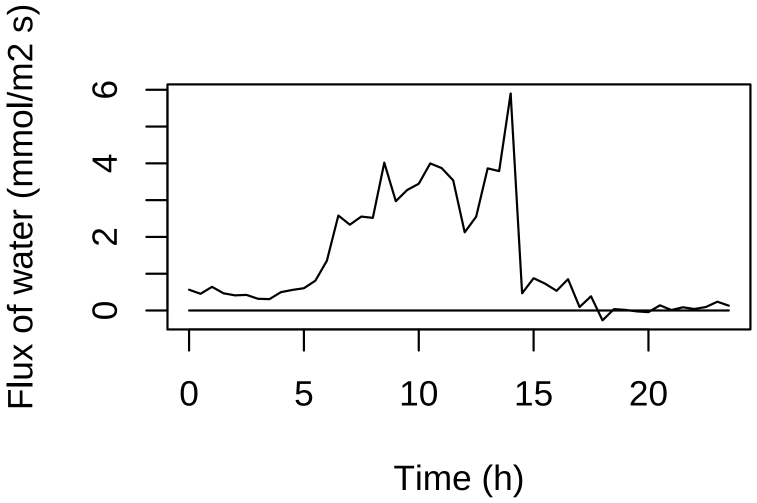


Figure 8.8: Evolution of water molar flux in CRA.01 data set

Eddy covariance in fact is guaranteed to give sensible results only under quite restrictive conditions, stationarity at least up to second order being one. In many cases (as often happens for example with fixed meteorological stations aimed at feeding input data to pollutant dispersion models) we can't be that finicky with data quality, and even values obtained when the assumptions behind eddy covariance are not met are considered better than gaps filled with an act of creativity.

But if a possibility of check is given, it is good practice using it.

So, what may we say of CRA.01 set? Are eddy covariance assumptions met? More importantly, *how* may we check?

As a starter, we check data against non-stationarity - one of the most common cases of eddy covariance error.

In view of this objective we proceed the classical way, by locating data items for which relative non-stationarity of wind components and non-steadiness of covariances fall beyond their associated limits (see [23]).

Within SonicLib, relative non-stationarity and non-steadiness are computed before rotations, so that non-stationarity from any wind component may contribute to non-stationarity along rotated vertical. For sake of brevity, and just for this example, we will restrict however to non-rotated vertical indicators.

We begin with relative non-stationarity of vertical wind, shown in figure 8.9 along with a reference line corresponding to the threshold, 0.50.

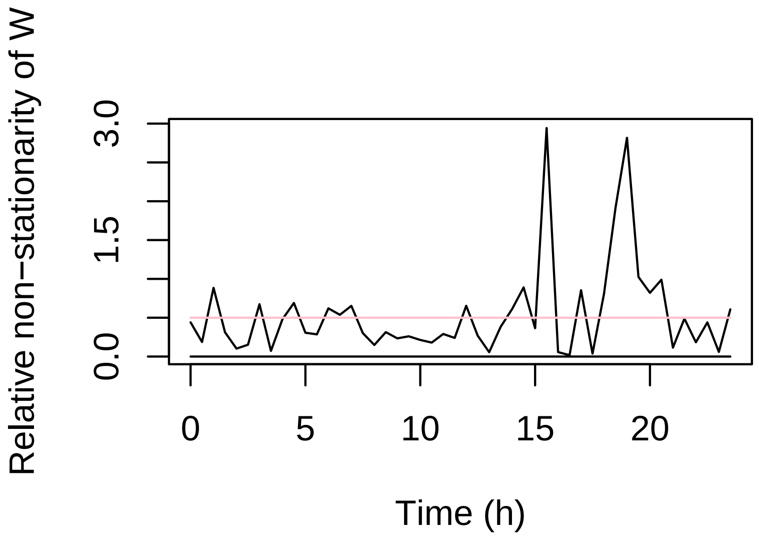


Figure 8.9: Evolution of relative non-stationarity of vertical wind in non-rotated frame in CRA.01 data set

Macroscopic anomalies apart, we may notice that 18 out of 48 data items exceed the threshold for relative non-stationarity of vertical wind. This number, apparently large (37.5 % of the whole data set) is somewhat in the middle of the values observed for ocean surface (15 %) and forests (55 %) (see [23]). Overall, this looks not bad (although the data set is so small we can't honestly draw any statistically sound truth). Yet there are 18 values for which relative non-stationarity is significantly large. What should we do on them?

Before attempting to devise an answer, let's have a look on non-steadinesses. In principle we should chech the ones from all possible covariances, but as before we limit attention to the three we're immediately interested to: $\overline{w'\theta'}$, $\overline{w'q'}$ and $\overline{w'c'}$.

We can see them in figures 8.10, 8.11 and 8.12.

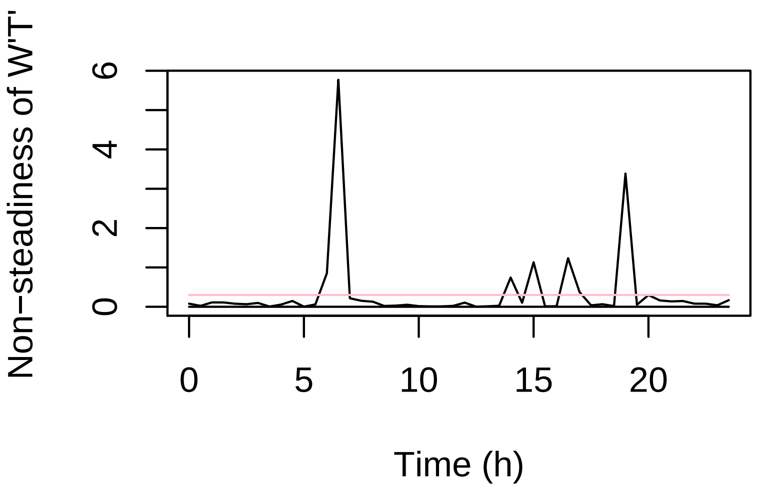


Figure 8.10: Evolution of non-steadiness of $\overline{w'\theta'}$ in CRA.01 data set

The typical threshold for non-steadiness is 0.30 (see [23]). As we had made with vertical wind relative non-stationarity, figures 8.10, 8.11 and 8.12 show a pink reference line at threshold level, allowing to see when exceedances have occurred.

With threshold exceedance rates of 0.146, 0.229 and 0.229 respectively for flux of temperature, water and carbon dioxide, the situation pertaining non-steadiness shows an incidence not far from the 0.375 we've seen for vertical wind relative non-stationarity. Distribution of exceedances in time is, however, quite different. A method for summarizing our findings is then welcome.

A sensible approach is to attribute each average a *quality meter*. Or, if you prefer, a *problem index*, and this is the way we do it.

Suppose each time we find some exceedance (of relative non-stationarity of w, or of any one of the three non-steadiness values) we increment a "trouble counter". The end result will be associating to each averaging block an integer number ranging from 0 (trouble-free, ideal situation) to 4 (all non stationarity indicators exceeded their threshold, the worst possible case).

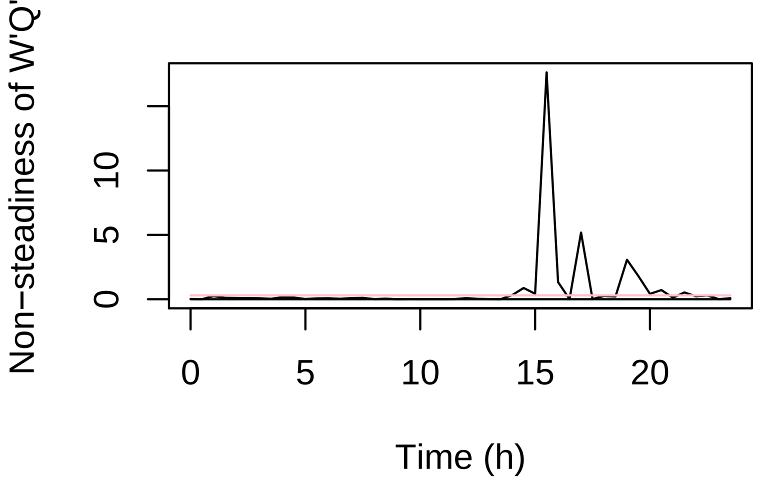


Figure 8.11: Evolution of non-steadiness of $\overline{w'q'}$ in CRA.01 data set

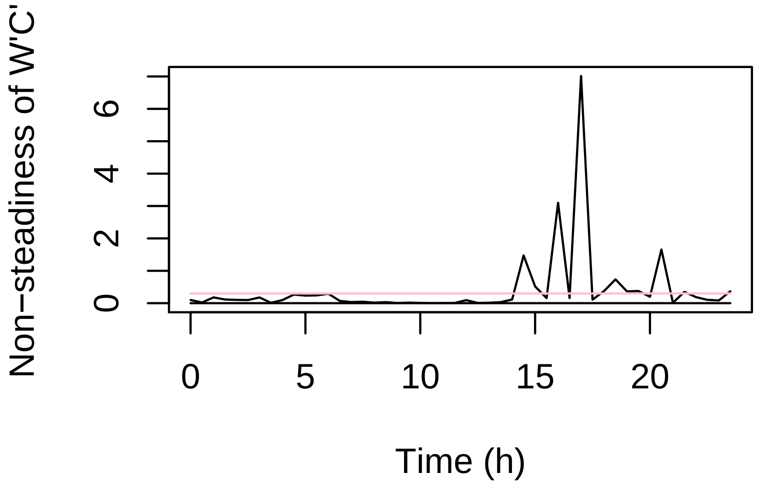


Figure 8.12: Evolution of non-steadiness of $\overline{w'c'}$ in CRA.01 data set

This is very easy to accomplish in R. First of all, we create a vector of numbers of the same length of the number of averaging blocks:

```
W.rns <- e$data$w.rns
trbl <- numeric(length(W.rns))
```

This first line just extracted vertical wind relarive non-stationarity, just to save typing. The important line is in reality the second one, creating a vector of the right length all whose elements have the value 0 (you may test this).

We may then increment the counter for the averaging blocks where vertical wind relative non-stationarity exceeded the threshold 0.5:

```
idx <- abs(W.rns) > 0.5
trbl[idx] <- trbl[idx] + 1
```

The first line creates an index vector `idx` of boolean values, where the TRUEs indicate an exceedance. This index vector is used in the second line to increment all components of trouble vector `trbl` where an exceedance has occurred.

We may then repeat the process using this time the three non-steadinesses, and incrementing exactly where they occur:

```
WT.nst <- e$data$wt.nst
idx <- abs(WT.nst) > 0.3
trbl[idx] <- trbl[idx] + 1
WQ.nst <- e$data$wq.nst
idx <- abs(WQ.nst) > 0.3
trbl[idx] <- trbl[idx] + 1
WC.nst <- e$data$wc.nst
idx <- abs(WC.nst) > 0.3
trbl[idx] <- trbl[idx] + 1
```

If we type `trbl` from the R command prompt we can see how many troubles we have, and when.

```
 [1] 0 0 1 0 0 0 1 0
 [9] 0 1 0 0 2 2 1 0
[17] 0 0 0 0 0 0 0 0
[25] 1 0 0 0 3 3 3 2
[33] 2 1 4 0 2 2 4 3
[41] 2 3 0 2 0 0 0 2
```

It is interesting to note that many averaging blocks are trouble-free. A couple, however, totalized 4 troubles each, indicating possible problems with the averages.

If the troubles score shows such a moved and interesting dynamics, how might we use it? The answer, non unique, depends on the context and the application.

A possible line of action might be to not consider in final evaluations all data whose trouble count exceeds some value. Or, troubled data may be flagged as suspect, and their users warned of possible problems would they use flagged data (discharging in the end any responsibility to their judgement). These are just two example, of the many possible we may find.

On a more practical side we pose another problem: are non-stationarity indicators *enough* to pinpoint problems in data? Maybe not? We will see a possible glimpse in the next section.

Non-stationarity, rotation angles and very strange results

One of the SonicLib developers has witnessed a very strange output from a few fixed eddy covariance installations, where occasionally the buoyancy flux exceeds by half an order of magnitude the incoming shortwave radiation.

This nuisance is not frequent (six cases per year in the worse affected station), and its manifestation is macroscopic enough to not survive validation. Yet it's annoying, and provisions were taken to perform a specific investigation.

The result was very interesting: the strange cases occurred when wind was subject to a sudden change in strength or direction, accompanied by significant change of temperature. These conditions are not impossible in Po Valley, especially associated with Foehn development.

These sudden velocity changes, acting as macroscopic non-stationarities, interfered with the determination of rotation angles. Of these, the first occurs around the vertical axis and its consequences are typically harmless.

But this is not true which the *second* rotation. It may in fact happen a very large tilt is applied, due to the massive non-stationarity, while terrain and good sense would have suggested a near-horizontal wind tangent plane.

When the incorrectly large tilt rotation is applied, the large covariance between horizontal wind components and changing temperature leaks into the vertical direction, and is counted by eddy covariance as an unrealistically huge vertical buoyancy flux.

It is worth mentioning this effect occurs only if non-stationarity on wind velocity is accompanied in synch by a nonstationarity in temperature. If only one of these conditions is met important errors also occur, but the magnitude of the change to buoyancy flux is not sufficient to allow easy detection on visual inspection.

On a deep level this real-world example shows how catastrophic effects may come from a wrong partition between mean and turbulent parts of natural signals. The issue is problematic, however, as the mean and turbulent partitioning schemes in use to date are based in most cases on time averages over a fixed length interval, whose limits are not connected to abrupt changes in the signal. Solution of this problem may challenge even adaptive averaging interval length determination, and maybe will involve some redefinition of partitioning scheme on more physical grounds.

On the more practical side, we maybe have just been given the gift of another data quality indicator: value of second rotation angle!

That this can eventually be can be visualized by plotting the second rotation angle as a function of time. We've made on CRA.01 data set, and results are presented in figure 8.13.

As flat as the CRA.01 site is locally, second rotation angle assumes sometimes veri large values. In addition if we look at sign we see it's always negative, indicating for CRA.01 data a systematic vertical component.

Second rotation angles for CRA.01 set, by the way, are much smaller than observed occasionally at the faulty location, where values in excess of 30 degrees

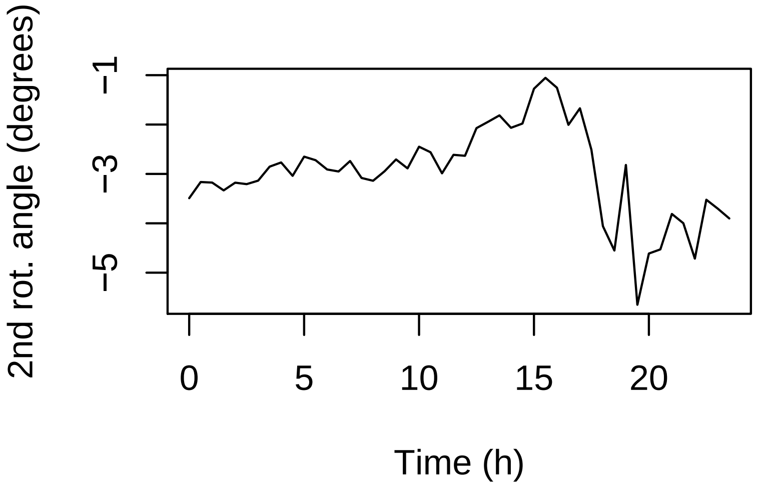


Figure 8.13: Evolution of second rotation angle in CRA.01 data set

have been observed. On the basis of second rotation angle alone we have no apparent reason to label some of the 30 minutes averages in CRA.01 suspect.

Yet, might this check devise some further investigation?

Removing linear trends

One of the many forms nonstationarity may take is a gentle monotonically increasing or decreasing "trend".

In many cases this "trend" may be well approximated by straight segments, which can be approximated from data.

If outliers are not exceedingly frequent, a sensible (in the senses: economical and implementable in one line of code within R) way is using linear regression, as explained in section 3.3.

In a more SonicLib flavor, we may enable trend removal by specifying `trend.removal` to something different from the default `"none"` in calls to `average.sonic.data` and `average.sonic.file.set`.

The following example shows processing of data set CRA.01 with and without trend removal, and producing a plot of buoyancy flux.

```
# Compute control and treatment
d0<-average.sonic.file.set(
  "../TestData/CRA.01",
  trend.removal="none",
  verbose=TRUE);
d1<-average.sonic.file.set(
  "../TestData/CRA.01",
  trend.removal="linear",
  verbose=TRUE);
ec0 <- eddy.covariance(d0, station.altitude=50);
ec1 <- eddy.covariance(d1, station.altitude=50);
t.stamp <- d0$data$t.stamp;
H0.v.trend   <- ec0$data$H0.v;
H0.v.detrend <- ec1$data$H0.v;

# Plot buoyancy fluxes with and without trend removal
pdf(file="../manual/diagrams/H0v_detrend.pdf",
  width = 4.2, height = 3.5);
plot(t.stamp, H0.v.trend,
  type="l", xlab="",
  ylab="Buoyancy flux (W/m2)",
  col="red");
lines(t.stamp, H0.v.detrend,
  col="blue");
lines(t.stamp, rep(0,times=48),
  col="black");
dev.off();
```

(Incidentally, you may note this example may be run as an automatic procedure).

The result of preceding example can be seen in figure 8.14. In this specific case you may see how the two curves are similar, a visual suggestion any trend present in signal was not severe enough to perturb the covariance $\overline{w'\theta'}$.

This is not necessarily the case, however. Cases exist in which important indicators change significantly, at least if we consider visual impact. Linear detrending may then improve final results.

Is this always true? In general the answer is negative: it depends on which shape the actual trend has!

As a counterexample, imagine a data series of length 36000 representing (say) temperature. The series is composed by a perfectly constant block of length 18000, of value 20, followed by 18000 data items whose value increases linearly from 20 to 22. This data item may be generated using the R command

```
tt <- c(rep(0,times=18000),
  seq(from=20,to=22,length.out=18000))
```

We may then compute the regression line,

```
tau<-1:36000
tt.l<-lm(tt~tau)
```

whose corresponding equation is

$$5.556 \cdot 10^{-5} \cdot t + 19.50 \tag{8.45}$$

The problem in this case is we're trying to approximate a piecewise-linear function with a (simpler) straight line. The deviation of trend from function is quite apparent in figure 8.15.

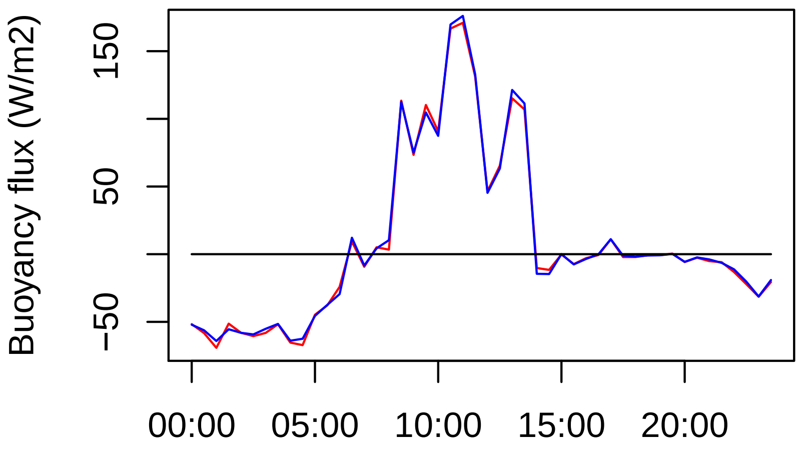


Figure 8.14: Buoyancy flux with (blue) and without (red) linear trend removal

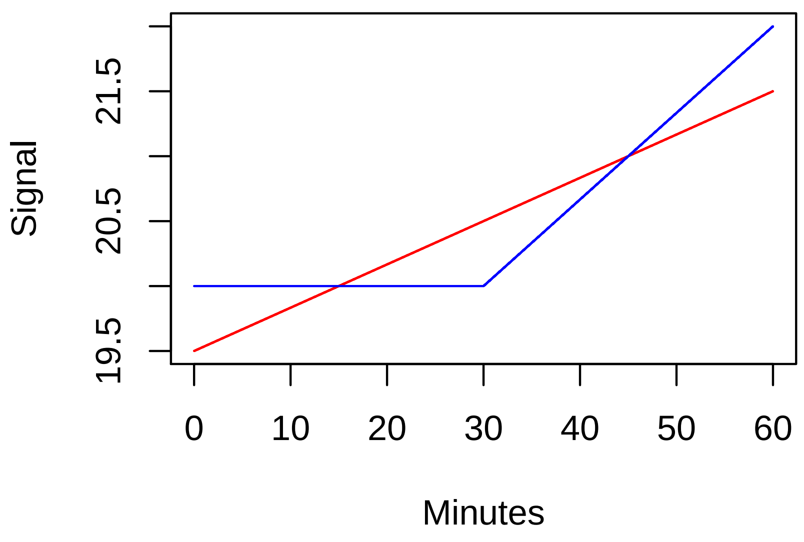


Figure 8.15: Example of an artificial signal (in blue), simulating temperature, and (in red) its fitted linear trend

Residuals from trend in a case as depicted in figure 8.15 are large enough to elicit immediate suspect. But what happens on real-world signals, especially “convective”, where variation may be more apparent than trend?

Checking some more quantitative aspect may prove helpful in these cases. For example we may try to compare relative non stationarity (of w in our case) or non-steadinss (for $\overline{w'\theta'}$).

Figures 8.16 and 8.17 show what happens in CRA.01 case.

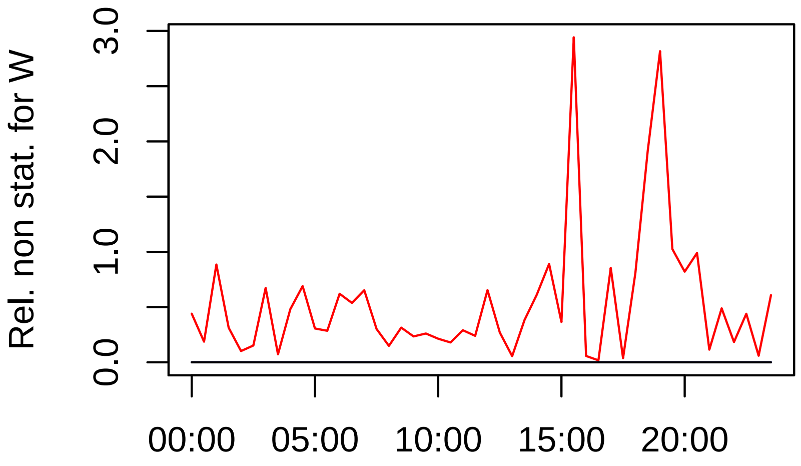


Figure 8.16: Relative non stationarity for wind component w with (blue) and without (red) linear trend removal

In figure 8.16 we can see only one line of the expected two. This is because relative non-stationarity for wind component w is very close to 0 and covered by the reference black line. But this adds very little to our knowledge: by the very way it’s defined (linear regression of experimental data using time as independent variable) relative non-stationarity “must” drop to 0 (within roundoff errors).

More interesting is the behavior of non-steadiness, whose connection with linear trend removal is less strong. As figure 8.17 suggests, some decrease occurs especially in correspondence of the two large peaks at 6 and 19. The smaller peak at 17 o’clock has remained perfectly unchanged however.

The effect of detrending may then be beneficial in many cases, but we should take for granted specific situations will present where this strategy is not ideal. This is not surprising: nature is *non*-linear, and imposing linearity on it is nothing more than a convenient artifice.

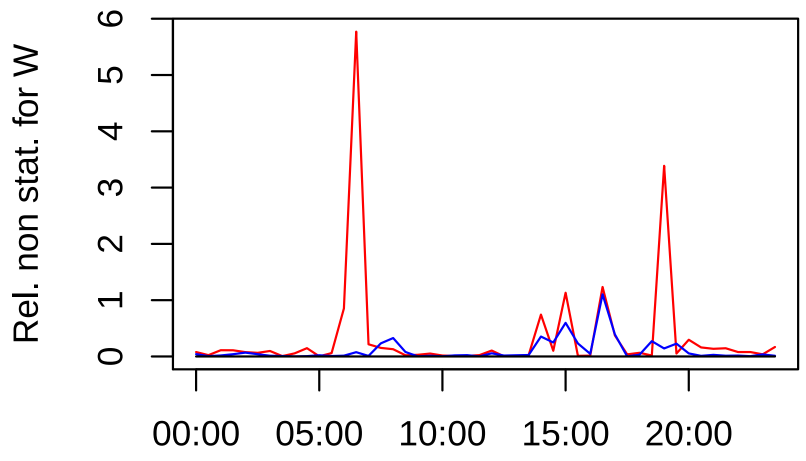


Figure 8.17: Non steadiness of $\overline{w'\theta'}$ with (blue) and without (red) linear trend removal

Comparing the horizontal and vertical sigmas

The standard deviations of horizontal and vertical wind can be used to feed atmospheric dispersion models, and the question arises of "which" standard deviations to use.

In modeling applications it is customary to use the vertical and cross-wind variances, and this suggests *rotated* covariances are of interest. One practical question then is, whether non-rotated variances can be used instead. A positive answer makes sense, as computing non-rotated variances is of course simpler than with rotated variances.

On the other side, this positive answer is not physically impossible to achieve. In fact, the sum $\sigma_u^2 + \sigma_v^2 + \sigma_w^2$ (incidentally, twice the turbulent kinetic energy) is the *trace* of the covariance matrix, and as such is invariant with respect to transformations like

$$\mathbf{SCS}^T \tag{8.46}$$

where $\mathbf{S}$ is non-singular. Rotation of reference, as performed in classical eddy covariance or planar fit, is a special case of this same transform, so we may expect the sum of the three wind component variances to stay unchanged under rotation. On reasonably not-too-complex terrain we may expect that the vertical component will not change too much, so the same must also be true for the sum of the two *horizontal* variances.

That this desirable fact really happens is site-dependent, and connected to the average absolute magnitude of the second rotation angle. The verification is simple however.

In figure 8.18 we see the effect of reference rotation in CRA.01 set, by applying classical eddy covariance with 2 rotations.

As we can see, rotated and non rotated horizontal variances are practically the same in this case. By the rotation invariance of turbulent kinetic energy the same must be for vertical standard deviation. But skepticism is a virtue in science, so let's check directly in figure 8.19.

The similarity of rotated and non-rotated variances is not the full story. Another interesting question is, whether some relation exists between horizontal and vertical component variance. In some dispersion models a linear relation between horizontal and vertical variances is assumed.

The scattering we can see in figure 8.20 is visually significant, yet the zero-passing linear model with its 0.1431 horizontal-to-vertical sigma suggests a much stronger horizontal standard deviation.

The existence of a clean relation between vertical and horizontal sigma allows site specific parameterizations of the sigma, which may be inserted in an existing atmospheric dispersion model, possibly after data have been classified in stable and convective; better even, measured standard deviations may be used directly in the model.

8.1.7 A caveat: intrinsic limitations of eddy covariance

Eddy covariance is a very powerful technique, and as such it is often used in many applications of immediate value, as for example quantifying fluxes of water or nutrients at the interface between an ecosystem and the Earth atmosphere.

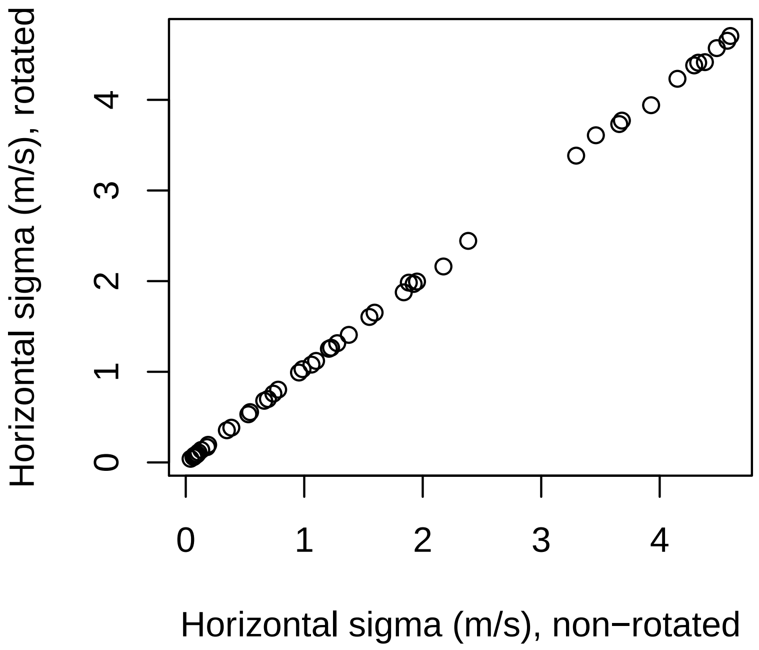


Figure 8.18: Horizontal standard deviation $\sqrt{\sigma_u^2 + \sigma_v^2}$ in non rotated vs rotated frame (data set: CRA.01)

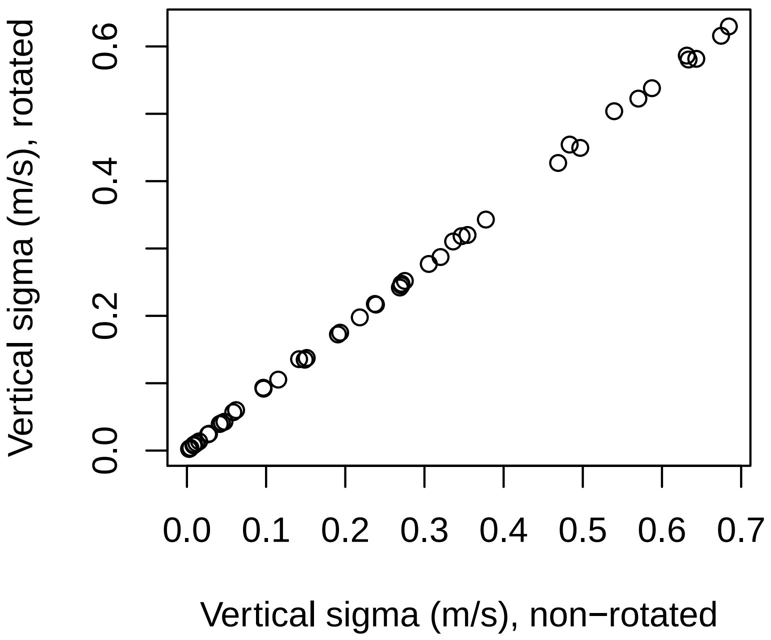


Figure 8.19: Vertical standard deviation σ_w^2 in non rotated vs rotated frame (data set: CRA.01)

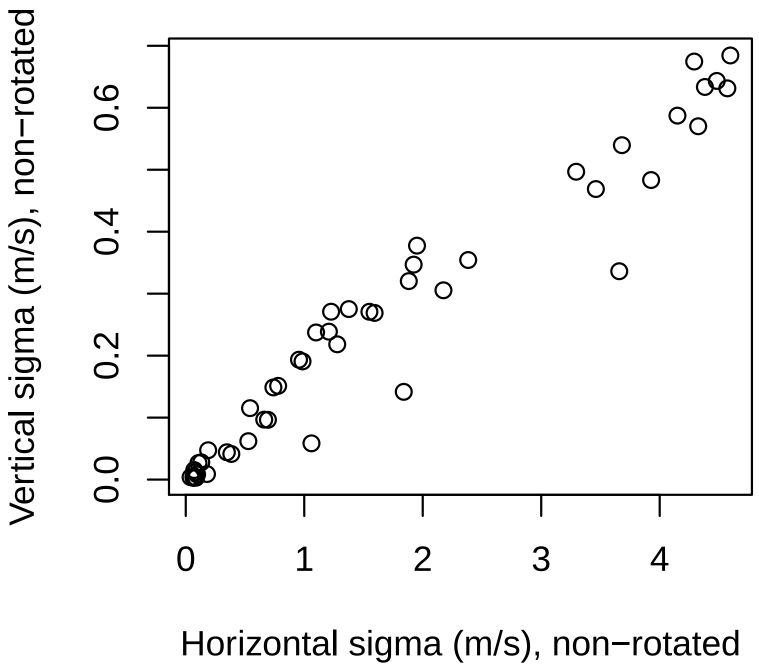


Figure 8.20: Horizontal and vertical standard deviations compared (data set: CRA.01)

In so doing many implicit assumptions may be taken for true, and a verification is often worth.

One assumption is terrain and canopy to be flat and uniform, and it often is self-evident whether this condition is met by a field experiment. Other assumptions may be much less intuitive, and related to the complex physics involved.

Experience shows eddy covariance exhibits a tendency to underestimate fluxes of carbon dioxide on night-time, under stable conditions (see e.g. [23]). Some recent work (see in particular [8]) highlight the possibility that divergence of horizontal flux contribute significantly. This is unfortunate, as the measurements needed are expensive and difficult (more towers, equipped with sonics and gas sensors, operating as in sync as possible). And "unfortunately" this effect may not explain all, or most, of the vertical exchanges of carbon dioxide, other factors (as flux from ground) having a very important impact on observed data.

Things may be even tougher than said. In cases where the "forgotten terms" are negligible other factors may also occur inducing eddy covariance to underestimate fluxes.

An interesting possible effect has been identified on rough terrain (see [9]), where low-frequency eddies may give an important contribution to fluxes. Long-frequency eddies might easily be washed out as "trend" by standard eddy covariance performed aggressively enough.

Possible solutions may come from including the non-intuitive aspects in models, and using them to validate measurements or devise new experimental methodologies. Two examples of this approach are shown in [10] and [12].

Anyway an appreciation of which is the (scientific and practical) context of measurements is fundamental to gain knowledge from measurements, and not just huge piles of data.

8.2 Ultrasonic anemometry for wind power plants

8.2.1 Introduction

Wind power plant economic feasibility at a given site depends critically on whether enough wind is available, and if this wind can be used without harm (that is, at acceptable maintenance cost).

Traditionally the problem of *plant siting* is addressed in stages, beginning with a CFD simulation performing the screening of comparativelky large regions (some kilometers) to locate the most promising sites for turbines. Once these prospective sites are defined, measurements are taken for limited periods (typically two months) to confirm the CFD predictions.

Measurements can be taken in many ways. Perhaps the most common (and supported by current technical standards) is by using mechanical anemometers installed at different heights above ground on a meteorological tower. Technical alternatives exist, and are gaining momentum: for example, using small SODAR system for remote sensing wind speed at many altitudes. And, of course, ultrasonic anemometers. This is the application we'll cover in this section.

Ultrasonic anemometers promise many improvements over mechanical anemometers. Two of these may be of special concerns to wind power plant designers: the ability of ultrasonic anemometers to see also the vertical wind (to which conventional anemometers are blind) and being able to measure turbulence directly. Both turbulence and vertical wind affect negatively the long term reliability of turbines. Said differently: they are connected to financial losses.

Just as a remind as we are dealing with money issues: large turbines cost a monstruous lot. I've heard orders of magnitude of 2 millions Euros per 1.6 MW (to date a medium size turbine, tending towards the small side). Such big investments are typically acted by banks, who are willing to give their money if and only if they foresee solid prospectives of yield.

This means banks wish to predict the future financial yield (or loss) associated to a prospective wind power plant the most accurately they can. As this is not always possible (banks often lack the accurate knowledge of wind) they tend to rely on the predictions made by designers, by ensuring these people follow sound modeling and measurement practice.

How do they do? The "usual" way in engineering: by verifying the plant designers and proposers have followed a rigid protocol. The protocol is quite detailed, and reflects the existing technical standards and the measurement technology available when they were first formulated. Currently, banks ask for "certified" measurements - and for measurements to be certified they must have been taken using "certified instruments" (to date, some mechanical anemometers). SODARs and ultrasonic anemometers are superior, and provide information banks would like very much to have, but are often not "certified". Or if they are, certificates are valid only in some countries and their automatic acceptance in other countries is not granted (neither impossible, however: as far as I know, for example, certificates emitted in Germany are accepted without discussion in Italy). To specify in precise legal terms what a "certificate" is lays much beyond the purpose of this manual, and we'll not pursue it any longer.

Rather, we focus on what ultrasonic anemometers may yield of interest to bankers, plant designers and other stakeholders, that conventional anemometers can't provide. Maybe will this contribute a little bit in improving the obsolete technical standards in current use?

8.2.2 The function "wind.power"

Function `wind.power` yields wind power and maintenance related data, both as sets of time-stamped vectors and overall indicators. Analyzed by skilled people, these numbers give detailed information about wind from a power production standpoint, to a level much refined than using mechanical instruments.

Before to proceed with the description, an important detail is worth mentioning: focus is on *wind*, not turbines. Any specific turbine has its own power curve, which may be vastly different than power curves for other models. Electing one turbine as reference would be unfair to all others, so we'll not do. On the other side: is knowledge of wind is detailed enough, then computing a yield prediction is quite simple if the turbine power curve is known.

To invoke the function you may type this command:

```
wind.power(
  d,
  mode,
  cut.speed,
  lower.cut.speed,
  upper.cut.speed,
  verbose
)
```

where

d is an object of class `sonic.avg.data` containing the averaged data set from which to synthesize wind power related information.

mode is an alphanumeric string containing the type of reference rotation to be made, with two possible values ("2.rotations" and "3.rotations", with obvious meanings; default is "2.rotations").

cut.speed is a boolean variable specifying whether processing should be limited to wind speed between specified limits (TRUE) or not (FALSE) (default: TRUE).

lower.cut.speed is a wind speed signifying the lower limit to be considered if `cut.speed` is TRUE (m/s; default: 6; ignored if `cut.speed` is FALSE).

upper.cut.speed is a wind speed meaning the upper limit to be considered if `cut.speed` is TRUE (m/s; default: 6; ignored if `cut.speed` is FALSE).

verbose is a boolean flag specifying, if TRUE, that error and informational messages are to be printed, or not if FALSE (default: FALSE).

On successful completion function `wind.power` returns an object of class `sonic.wind.power.data`. In case of error a NULL value is returned, and if `verbose` was seto to TRUE an error message is printed explaining what happened.

8.2.3 Wind power specific quantities, and how they are computed in SonicLib

Data requirements in wind power engineering do not coincide with eddy covariance and flux-related computations as we've seen in 8.1, although some overlap exists.

Two important problems in wind power engineering are how fast energy is produced, and whether wind conditions are likely to cause maintenance problems. The answer to both questions has a deep economical impact, determining in fact whether the plant will ever repay itself, and possibly produce some wealth.

Both questions are surprisingly tough, rooted in statistics as they are. Sensible answers based on sound rationality must start from measurements, and sonic anemometers give an important help as we'll see in the next sub-sections. It is worth mentioning that intuition is not as useful in this regard: we humans are prone to overestimate the impact of unusual conditions, which at many sites include a wind sufficiently strong to start turbine rotation.

Mean and total power

Maybe the most apparent indicator of wind quality at a site is its mechanical power. This is known to be proportional to the cube of wind speed,

$$P = k \cdot U^3 \tag{8.47}$$

The constant k depends on turbine size, geometry and to some extent context, but in fact can be considered fixed provided a given turbine is selected. We may then concentrate on wind speed alone, introducing the corresponding wind factor,

$$F_P = U^3 \tag{8.48}$$

Expression (8.48) is entirely abstract, as it involves wind speed as it is. We may rewrite it in more practical terms if we care to apply Reynolds decomposition to the total wind speed, $U = \overline{U} + U'$. By applying the binomial theorem we first get

$$U^3 = \left(\overline{U} + U'\right)^3 = \overline{U}^3 + 3\overline{U}^2 U' + 3\overline{U}U'^2 + U'^3 \tag{8.49}$$

What we obtained is *instantaneous* wind power, and we get very little from it. Average power is decidedly more interesting:

$$\overline{U^3} = \overline{\overline{U}^3 + 3\overline{U}^2 U' + 3\overline{U}U'^2 + U'^3} \tag{8.50}$$

The right member in 8.50 may be rewritten taking into account $\overline{x+y} = \overline{x} + \overline{y}$ as

$$\overline{U^3} = \overline{\overline{U}^3} + 3\overline{\overline{U}^2 U'} + 3\overline{\overline{U}U'^2} + \overline{U'^3} \tag{8.51}$$

In 8.51 we have $\overline{\overline{U}^2 U'} = \overline{\overline{U}^2}\,\overline{U'} = \overline{U}^2 0 = 0$, $\overline{\overline{U}U'^2} = \overline{\overline{U}}\,\overline{U'^2} = \overline{U}\,\overline{U'^2} = \overline{U}\sigma_U^2$ and $\overline{\overline{U}^3} = \overline{U}^3$ so that the formula can be rewritten as

$$\overline{U^3} = \overline{U}^3 + 3\overline{U}\sigma_U^2 + \overline{U'^3} \tag{8.52}$$

The terms on right of 8.52 may be interpreted according to their physical meanings. $\overline{U}^3$ represents the average power, that is the part likely to be extracted from wind (at some given rendement) by a turbine.

The remaining part, $3\overline{U}\sigma_U^2 + \overline{U'^3}$, indicates the amount of power stored in turbulence. This power, not organized in some non-zero mean wind, is likely off the possibilities of use by wind turbines, at least directly, and the interesting question arises of large is it.

From a more practical side, computing the effective part of wind energy is straightformward: it is sufficient to get the period time average of wind speed, and raise to the cube. This is the meaning of `mean.power.coef` and `mean.power.coef.3`, the difference between the two coming from the first using only the horizontal components, and the second the whole vector.

Whether or not the turbulent quota of wind energy power affects turbines adversely might be an interesting research question, lying beyond this work' scope however, and we'll not follow it any seriously.

Turbulence intensity

An important indicator of wind quality (or better its lack) is *turbulence intensity* defined as

$$T = \frac{\sigma_U}{\overline{U}} \tag{8.53}$$

where U represents instantaneous wind horizontal speed, and both mean and standard deviation have been computed on the same time interval.

Turbulence coefficient in SonicLib is computed in two ways. In the first, a turbulence intensity value is calculated for each time average (element `T` in `data`) using mean and variance of wind speed as computed when averaging the raw data set,

$$T\left(t\right) = \frac{\sigma_U\left(t\right)}{\overline{U}\left(t\right)} \tag{8.54}$$

In the second, a global turbulence coefficient is introduced

$$\overline{T} = \frac{\sum_{t \in \tau} \sigma_U\left(t\right)}{\sum_{t \in \tau} \overline{U}\left(t\right)} \tag{8.55}$$

Of these, the first 8.54 is the one to really consider. The second, 8.55, stored in data element `turb.coefficient`, is a sort of indication of the residual turbulence intensity due to the choice of averaging time. The possibility this indicator is nonzero (and possibly significantly large) depends on the fact that averaging time is standardized to 10 minutes, well below the mean position of the spectral gap, leading in many cases to an underestimation of time-wise wind speed standard deviation.

Angle of wind to the horizontal plane

Under ideal conditions the mean wind vector is characterized by a condition like $\overline{w} = 0$, that is, it belongs to the horizontal plane.

Ideal conditions are very seldom found in nature, and it very often happen to the mean wind vector to form a non-zero angle to the horizontal. The value of this angle, a signed quantity, is computed as

$$\Phi\left(T\right) = \tan^{-1}\left(\frac{\overline{w}\left(T\right)}{\overleftarrow{\overline{U}}\left(T\right)}\right) \tag{8.56}$$

Friction velocity

Friction velocity is an important indicator in wind power engineering, its value being often used to estimate wind profiles as

$$U\left(z\right) = \frac{u_*}{k} \log \frac{z}{z_0} \tag{8.57}$$

Relation 8.57 is strictly speaking valid only under neutral (adiabatic) conditions, that is when potential temperature stays constant with height above ground. Under stable and convective conditions an additional similarity term should be added to the logarithm, yielding a much more complex formula. On the other hand, existing wind turbines are able to produce energy in sensible amounts only at wind speeds in excess of, say, 6 to 8 m/s. By then, as shown in [31], the strong mixing caused by wind friction with ground surface is able to destroy any stable stratification or unstable convective structure, thus leveling anything towards the adiabatic.

So, we can assume 8.57 makes some theoretical sense. SODAR profiles most often show a much more complex situation, sometimes with various layers of air overlapping with different properties. Even considering this, relation 8.57 still tends to hold *in the mean*, when many consecutive SODAR profiles are averaged together.

In lack of precise measurements of it, friction velocity is estimated by wind power engineers using the fixed value 0.1 m/s (if one only measurement level is available), or by an expression like $U(z_1) - U(z_2) = \frac{u_*}{k} \log \frac{z_1}{z_2}$ when two measurement levels z_1 and z_2 exist. In both cases, the von Karman constant k does not bother as its value is agreed to be practically equal to 0.4. The other quantity, the aerodynamic roughness length z_0, also does no harm as it is not considered with one measurement level, and it cancels out with two levels.

These simplified formulas often bear with them a large error whose extent is difficult to quantify merely relying on theory and a-priori information. Direct measurement by sonic anemometer sometimes shows quite different values, sometimes up to 0.3 m/s.

Because of this, friction velocity is computed by SonicLib, using the very same approach introduced in section 8.1:

$$u_* = \left(\overline{u'w'}^2 + \overline{v'w'}^2\right)^{1/4} \tag{8.58}$$

As with eddy covariance friction velocity is computed on every averaging step. Statistics, both overall and directional, may then be computed using standard R commands.

Wind direction

Wind provenance direction is an important tool: in conjunction with `dir.mean` function it allows to detect spatial dis-uniformities which in case of wind power plants may be revealing.

Provenance direction is computed as

$$D = \tan^{-1}\left(\frac{\overline{u}}{\overline{v}}\right) \tag{8.59}$$

using the actual R command like

```
dir <- atan2(-u,-v) * 180/pi
```

so that result is expressed according to the conventions of common use in the Earth sciences, North being designated as 0 and angles increasing clockwise. The `atan2` function yields it results in the interval $-\pi \leq \theta \leq \pi$, so negative angles are produced. Converting them to positive form is easily accomplished by setting

```
dir[dir < 0] <- dir[dir < 0] + 360.0
```

Rotation angles

Rotation angles θ, ϕ, ψ are used to compute the streamline reference system necessary for compiting other quantities - in particular the friction velocity.

Storing them allows to detect pointwise anomalies in data time series, for example by highlighting data records whose second rotation angle exceeds a given safety band.

8.2.4 Preparing data for "wind.power"

We mentioned banks and bankers, and the standards they enforce as a condition to supply money. One standard does directly apply to our case: time averaging, which in wind power community is almost universally accepted to be 10 minutes.

This requirement closely connects to data being certifiable, an attribute allowing banks to reduce analysis costs, and very seldom might be changed: it's widely accepted by wind turbine community.

Preparing data for analysis by function `wind.power` then demands data to be averaged using a call like this:

```
d <- average.sonic.file.set(
  dir.name,
  averaging.time=10
)
```

where I've dropped all non-mandatory data.

8.2.5 Examples

Economical feasibility

Maybe the very first problem turbine designers (and banks) face is whether the plant to be will give some economical yield.

Indeed, investments for large plants are significant, in the order of 2 million Euro per 1.6 MW (to date a medium size machine: the really very large ones have reached rated power in the order of 6 MW, and dimensions tend to increase further). But as soon as the wind farm is connected to the electrical network any energy produced by it will generate financial yield.

For a plant to be economically feasible yield must exceed cost. Let's for the moment assume the plant owner is big enough to be able sustaining the whole installation and startup cost in a single large tranche, so that we may imagine the whole exit cash flow to occur just on the beginning of plant life. Of course this assumption is unrealistic, and here is where banks enter the overall picture, by providing loans so that the plant owner can pay bit by bit. But this is a detail which is not relevant to our example: all we have to know is that banks do not offer loans for free, so we should remember to increase the plant cost taking into account all financial loads.

As I mentioned, the plant produces some income. But this will occur slowly and, what matters maybe more, erratically: as much as wind power, more or less.

Here enters another factor: wind turbines do not endure forever, so it is common to consider a wind turbine lifetime of 20 years. After this time has elapsed the inevitable wear-out will have rendered maintenance cost prohibitive, not taking into account that by 20 years any current machine will be obsolete if technological development rate in wind turbines will remain as fast as it is now.

The morale is: a turbine must repay itself in *less* than 20 years. All time in the 20 years past the repay period will generate a net income.

For the turbine to be let's say economically reasonable, the net income should be larger than the interest a bank would pay if you put all the money used to build, install and start the turbine to an account. Would this not be, then it would be more convenient to store the money in a bank and live more or less happily as financial speculators. But this means that the turbine must not only repay itself: it must generate real money.

Of course this analysis is strictly unilateral. There are other costs associated to wind power plants which to date do not enter the economic equation. To name one, the massive environmental impact of turbine themselves and the construction works associated to their deployment (and it is not excluded these costs sooner or later will be translated into some form of taxation). As we'll see in later examples there are still other costs, this time immediately relevant to turbines, which have not yet been considered.

But for now, let's assume our simplistic engineering-style analysis is exhaustive. Then we face a problem which looks hard enough: to predict in advance how much power will a given turbine generate if installed precisely at the desired site.

The fine details depend on the actual machine, so let's simplify in the extreme.

In general, a wind turbine will not extract a fixed fraction of the effective power stored in mean wind. At slow wind speed the turbine blades will stall and the rotor remains stuck. If wind increases above a limit depending on the specific turbine geometry and control policy the rotor will start its rotation, but will produce only a fraction of its nominal power. This ratio will progressively grow until wind reaches the minimum nominal production speed. If wind increases further the control policy of the wind turbine will intervene more and more energetically to allow the turbine to continue producing exactly the nominal power, while dissipating the excess (usually in mechanical form, fo example by reducing the blades' attack angle). Eventually, would wind exceed the safety speed of the turbine, the control system will reduce the blades attack angle to zero and act a powerful brake, thus stopping the machine to limit structural damages.

To place all these subjective considerations in perspective it may be worth specifying in some more detail what is meant as a slow of fast wind. Many large turbine start their rotation with mean wind in excess of 4 m/s. But the turbine will produce an infinitesimal amount of energy until, say, a 6 m/s, the nominal power being met at around 8 m/s. Turbine shutdown will occur would wind blow at more than 20 m/s. These figures are indeed *large*. At current electricity remuneration, for a turbine to produce enough energy to repay it and generate some net income demands something around 70% of time with wind speed between 6 and 20 m/s. This condition is quite infrequent in the areas where most of our data sets have been collected, but do indeed occur in many places of the World. Offshore and mountain crest sites are not rare.

What of small turbines instead? To date machines rated at 1 kW or even less are not uncommon, and in many European sites is indeed likely to see one or two in town blocks, installed over roofs. These small-sied machines are lightweight constructed and because of this operate at a somewhat smaller wind speed. Rotation very often begins at 2 m/s, and full productivity is attained between 6 and 8 m/s. Of course the tiny nacelles of these amateur power producer plants are too small to host an emergency brake, and no blade rotation is possible. Then, would wind blow at an excessive speed these small-sized turbines are likely to disintegrate. The logics behind all that is fairly simple: the owner wait until tempest has gone and buys just another.

As I mentioned, in order to avoid excessive complications we'll stay simple. In practice, we imagine the turbine to operate ideally (that is, at full nominal power) within a fixed speed range, say 6 to 20 m/s. Stated this way, our problem reduces to determining how much time does wind speed stays within these limits.

We need to draw this information from wind measurements taken for a limited time at the prospective installation site. Wind is infamous for its erratic behavior however, and so we can see a trouble: the measurement period might be too short, or chosen so to not represent the climatological site reality.

But, let's begin with the how. The method commonly used to solve the problem assumes wind speed can be modeled as a random variable following a Weibull distribution. The trick then is to fit an instance of the Weibull distribution from data, write it as probability density function $p(x; \alpha, \beta)$, then derive the probability that wind speed is between the desired limit as

$$\mathbf{P}\left\{U_{min} \le U \le U_{max}\right\} = F(U_{max}; \alpha, \beta) - F(U_{min}; \alpha, \beta) \tag{8.60}$$

where $F(U_{max}; \alpha, \beta) - F(U_{min}; \alpha, \beta) = \int_{U_{min}}^{U_{max}} p(x; \alpha, \beta)dx$ is the difference between the cumulative distribution functions computed at interval endpoints.

The problem with Weibull distribution is its parameters α and β are not simple to estimate. The Weibull density is

$$p(x; \alpha, \beta) = \begin{cases} \frac{\alpha}{\beta} \exp -(\frac{x}{\beta})^{\alpha}(\frac{x}{\beta})^{(\alpha-1)} & x > 0 \\ 0 & x \le 0 \end{cases} \tag{8.61}$$

and its first two moments are

$$\mu = \beta\Gamma\left(1 + \frac{1}{\alpha}\right) \tag{8.62}$$

and

$$\sigma = \beta\sqrt{\Gamma\left(1 + \frac{2}{\alpha}\right) - \Gamma\left(1 + \frac{1}{\alpha}\right)^2} \tag{8.63}$$

Taken together, equations 8.62 and 8.63 constitute a system in the two unknowns α, β. This non-linear system is obviously closed, but a solution is not easy to find in explicit terms. To solve it we need then to use a numerical method, and we ought to build it in R.

Before to start solving the non-linear system as it is, we may do some manipulations in order to make the problem simpler. Indeed, if we divide 8.63 by 8.62 we get an expression whose value depends on α only:

$$\frac{\sigma}{\mu} = \sqrt{\frac{\Gamma\left(1 + \frac{2}{\alpha}\right)}{\Gamma\left(1 + \frac{1}{\alpha}\right)^2} - 1} \tag{8.64}$$

This expression may be seen as an equation allowing to identify the value of α, given $\frac{\sigma}{\mu}$. But unfortunately an analytic solution can not be found - at least simply. A numerical solution may be found easily however, the function

$$f(\alpha) = \sqrt{\frac{\Gamma\left(1+\frac{2}{\alpha}\right)}{\Gamma\left(1+\frac{1}{\alpha}\right)^2} - 1} \tag{8.65}$$

appearing to the right of 8.64 is continuous in the interval $(0,\infty]$ with $\lim_{x\to 0^+} f(x) = +\infty$ and $\lim_{x\to\infty} f(x) = 0$.

For any (necessarily positive) value of the ratio $\frac{\sigma}{\mu}$ equation 8.64 has exactly one solution. Once this solution α^* has been found, estimating the parameter β is straightforward:

$$\beta^* = \frac{\mu}{\Gamma\left(1+\frac{1}{\alpha^*}\right)} \tag{8.66}$$

If we use the whole data set CRA.01 using a call like

```
w <- wind.power(d, cut.speed=FALSE)
```

we get Weibull values $\alpha = 1.59$ and $\beta = 3.04$. The general shape of the Weibull density corresponding to these two parameters is shown in figure 8.21 for reference.

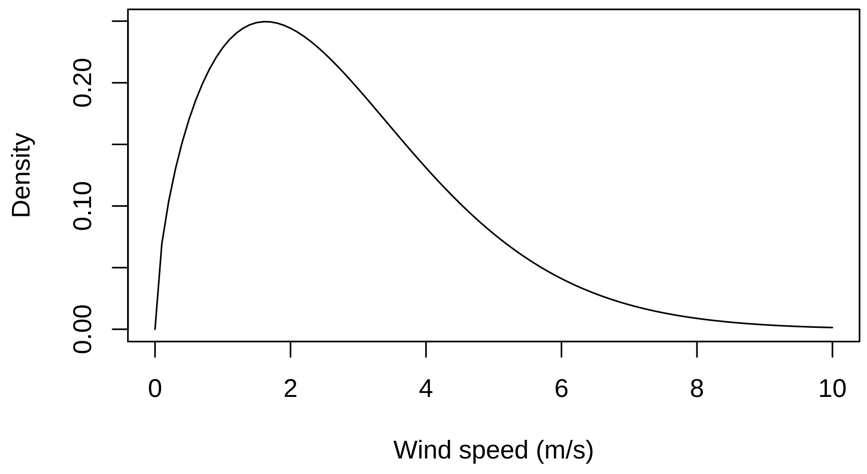


Figure 8.21: Density of estimated Weibull wind speed distribution (data set: CRA.01)

Weibull probability density says quite a little visually but an important thing: wind speed extends to quite large maximum values. Indeed we'd like a more intuitive economical indicator, and this is the cumulative probability distribution, or better even its one complement, showing the probability a given wind speed is exceeded. The value of this probability gives a rough indication of how often will the turbine produce valuable energy.

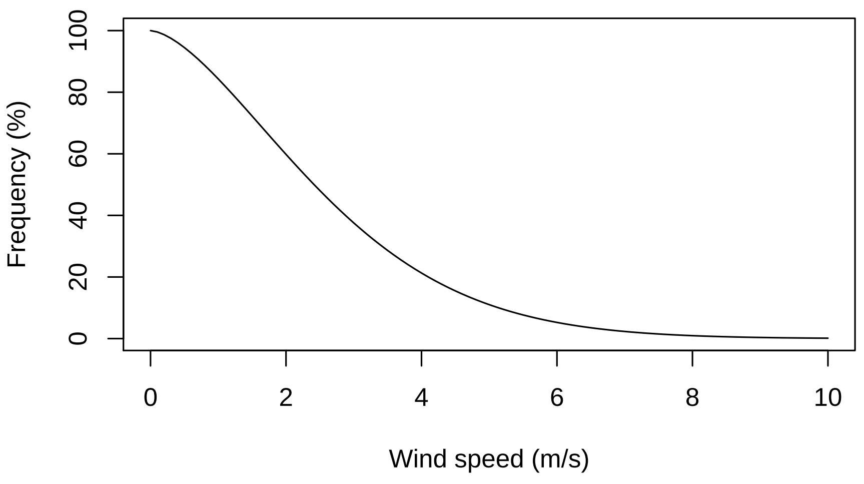


Figure 8.22: One-complement of the CDF corresponding to estimated Weibull wind speed distribution (data set: CRA.01)

Figure 8.22 just shows this indicator. Here the likelihood of production is clearly visible. Would for example the turbine start producing at 4 m/s, we could expect something in the order of 20 % of productive time (and an 80 % of idle time: not a very good thing, in financial terms). The precise value is 21.3 %.

When dealing with economical problem, the bare estimation of how many hours will be productive is a good starting point (especially to prove a site is *not* suitable for a wind power plant) but is usually not sufficient. A more interesting subject is having an indication of how much power will be actually produced and, optimistically assuming a one-to-one sell ratio, how much money will be earned at a specified tariff level.

Answering this advanced question demands however much more information, namely the turbine response curve. Response curves are typically given in form of tables, with power produced at some given speeds. Some of these tables also contain air density as a second input, lighter air containing less momentum at the same wind speed. An example of table is given for an existing small (10 kW) turbine:

Wind speed (m/s)	Power (kW)
1	0
2	0
2.5	0.4
4	0.9
5	2.3
6	3.3
7	4.9
8	6.7
9	8.1
10	9.3
11	9.8
11.5-25	9.8

We conclude this section with a caveat. Not all wind turbine plants are justified in economical terms. That is, there may be circumstances where the analysis just depicted makes no sense. An example may be the case of a wind turbine used to lift water on a small island where no power supply is available. In such a case the good is not some energy which can be sold, but rather the benefit of having water lifted, and then available to human use. If so, any output from the wind turbine is valuable whatever its actual amount: in the best scenario the water lifted by wind is sufficient to fulfill local demand. In the worst, it's anyway some spare of precious combustible. In these cases other kind of analyses should be made - which we'll not deepen here for reasons of space.

Is wind speed really Weibull-distributed?

In technical literature on wind power plant it is often assumed that wind follows a Weibull distribution, and in the preceding section we have used this hypothesis as well and drawn plant feasibility conclusion based on it.

But: it is always true for wind to be Weibull? Stated this (strong) way, the question is unlikely to have a positive answer. And in fact we may exhibit a counterexample, just based on the same CRA.01 data set we've just used in the preceding section.

By inspecting the output structure of `wind.power` we discover in fact two other fields related to Weibull distribution, namely `weibull.statistic` and `weibull.probability`. As explained in section 3.5, these fields report the result od Kolmogorov-Smirnov test made comparing the empirical cumulative distribution function to the modeled Weibull. Of these values `weibull.probability` is of more immediate interest to our case: it specifies the probability that our data set sample has been drawn from a Weibull distribution of specified parameters.

The value found on CRA.01 set, 0.453, is quite small: less than a 50 % chance! So, we can't honestly claim wind to be Weibull-distributed. We may even see it, if we plot the modeled Weibull density along with the gamma estimate of experimental data's, as we do in figure 8.23.

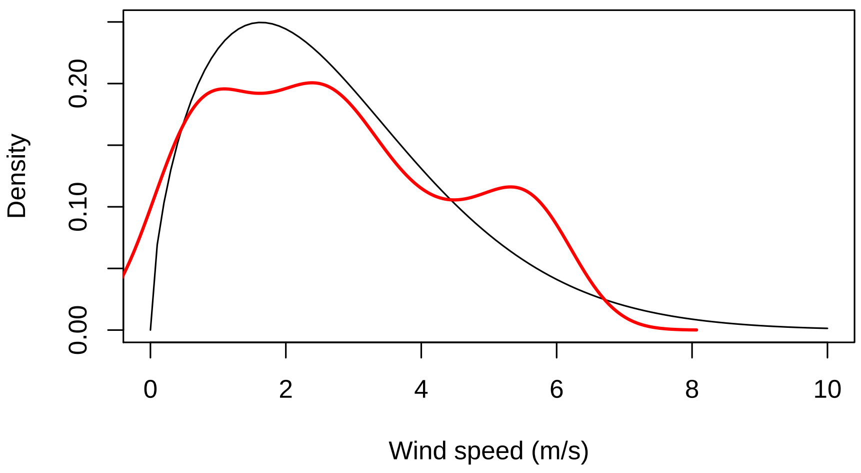


Figure 8.23: Density of estimated Weibull wind speed distribution overlapped to gamma estimate of empirical density (in red; data set: CRA.01; no cut speeds)

However: the empirical speed density is not that *dramatically* far from a Weibull, at least from a visual standpoint. If we limit attention to wind speeds whose value lays within the turbine operational interval 4 to 25 m/s as we've seen in the preceding section, Weibull probability rises to 0.814. The corresponding plot is shown in figure 8.24.

We retained less data (37 instead of 144), and the peaky result is more similar visually to a Weibull density. What all that means? An interpretation is, the Weibull model is not too bad, even is measurements are taken very precisely with an ultrasonic anemometer.

Why Weibull?

The Weibull density is characterized by an asymmetric shape (which fits well with wind speed non-negativity), and a tail which may be fat or lean depending

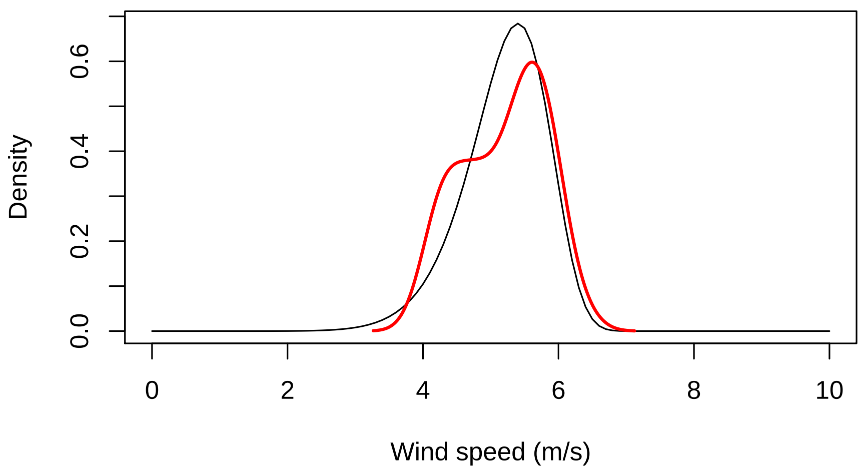


Figure 8.24: Density of estimated Weibull wind speed distribution overlapped to gamma estimate of empirical density (in red; data set: CRA.01, with lower and upper cut speeds respectively equal to 4 and 25 m/s)

on the value of α.

Of course this is not the only distribution gifted of the same properties. Among the ones sharing the same attributes, at least in given ranges of α and β, we can find the log-normal, the Gamma, the Chi-square, and others. Why not using them?

Indeed, some distribution other than Weibull might be computationally simpler. For example, to fit a log-normal distribution one should first identify its two parameters, (the so-called log-normal mean m and standard deviation s) starting from data mean μ and standard deviation σ. This operation may be carried on entirely on symbol manipulation, instead of using numerical methods as we have been compelled to in case of the Weibull distribution, as we can see from the two relations

$$m = \log \mu - \frac{1}{2} \log \left[1 + \left(\frac{\mu}{\sigma} \right)^2 \right] \tag{8.67}$$

and

$$s = \sqrt{\log \left[1 + \left(\frac{\mu}{\sigma} \right)^2 \right]} \tag{8.68}$$

Chapter 9

Visualizing Data and Their Structure

9.1 Plotting data the simple way

One important visualization of data, and maybe one of the most common, is just plotting them as functions of time.

SonicLib does not provide a function to perform this task, the almost-almighty `plot` command of base R being orders of magnitude more than plain sufficiency. Here we give some idea, however, by showing some practical interactive examples.

If you are not familiar with R reading this section may prove worth the bit effort, both for its own utility and the possibility to extend knowledge gained here to customize the SonicLib diagrams presented in later sections.

As mentioned shortly, the plotting command used in R is

```
plot(x, y, ...)
```

where `x` and `y` designate the two vectors (of the same length) containing the x and y coordinates on the plotting plane. Ellipsis are a placeholder indicating the (possibly long) set of parameters.

Parameters are of course of paramount importance, as their sensitive use allows to produce publication-quality plots with little effort. They exist in an impressive number, and if you like to browse among them we kindly invite you to have a look at R help (just as a starter, you may begin by giving command `?plot` from within an R session; as your needs become more advanced you may want to consult specialized texts like [29]).

9.1.1 Example: Understanding non-stationarity with time domain plots

As a first practical example suppose we investigate the reason of why relative non-stationarity of w peaked at 15:30 in CRA.01 data set, but not at 15:00. As we've discovered in section 8.1.6 data set CRA.01 is characterized by a “normal” value of relative non stationarity on w, from which a few distinct peaks emerge.

The one occurring at 15:30 characterizes for its narrow span, suggesting an unusual event suddenly coming in.

SonicLib raw data are stored in hourly file, whose naming convention (explained in 3.1) allows to predict the one we're interested in is `20100516.15.csv`: our first task is grabbing it into the R session:

```
f <- get.raw.data("../TestData/CRA.01/20100516.15.csv", verbose=TRUE)
```

Having specified `verbose=TRUE` allows to check for any error message. The function completed printing no message to session, so we can deduce all went right as far as SonicLib is concerned. But the final proof is in the result itself, which we may check giving the command

```
str(f)
```

This done, the system answers presenting a short sample of the data stored into file:

```
List of 2
 $ data         :'data.frame': 35990 obs. of 7 variables:
  ..$ time.stamp: num [1:35990] 0 0 0 0 ...
  ..$ u         : num [1:35990] -0.27 -0.61 ...
  ..$ v         : num [1:35990] -2.18 -1.89 ...
  ..$ w         : num [1:35990] 0.28 0.06 ...
  ..$ t         : num [1:35990] 15.4 15.3 ...
  ..$ q         : num [1:35990] 273 277 ...
  ..$ c         : num [1:35990] 16.2 16.2 ...
 $ sampling.rate: num 10
 - attr(*, "class")= chr "sonic.raw.data"
>
```

The summary given by `str()` is interesting and infrmative: it allows to get an impression of the nature of data, and a bit of intuition will easily arrive to figure out the structure. R allows other summaries however: for example, `summary()` applied to a data frame returns a complete statistic report of each column.

Among all other useful things, command `str()` allows us to understand where the data we're interested in may be found. We may decide to store this knowledge in our neurons for later use, or make use of some other commands to make our life less painful now. We follow this second path, by isolating time stamp and vertical wind component in variables whose names is easy to remind and shorter to type:

```
t.s <- f$data$time.stamp
w <- f$data$w
```

We may now engage the real problem, visualizing data. We may begin with the simplest possible plot:

```
plot(t.s,w)
```

Result is shown in figure 9.1. As you can see, the result is absolutely unsatisfactory! In fact, if no specific option is given the R default version of `plot` just builds a scatter plot, with open circles at every data point. The circles are wide, and data plots in large number: the final result is filling a third of the plot with a black band, losing completely any hope to follow the actual variation.

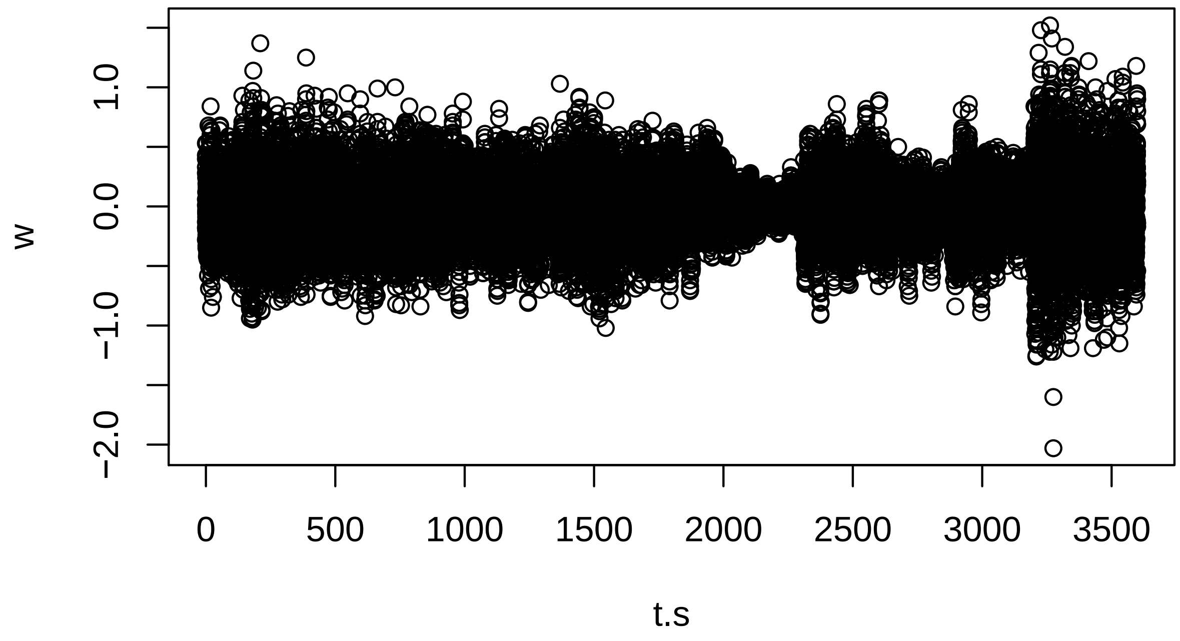


Figure 9.1: Variation of vertical wind speed with time, from 15:00 to 16:00 in CRA.01 set

What we really desire in fact is a *line plot*. In R this means to specify an additional parameter:

```
plot(t.s, w, type="l")
```

The new result is shown in figure 9.2.

The result of our second attempt is a bit better than before: we now have a better impression of actual data variation. Yet the plot is still unsatisfactory: to begin with, axes descriptions are at best uninformative to people not already knowing the facts.

A better approach would be to indicate clearly what the variables are, and which measurement units have been used. This is simple for the vertical axis. But what of the horizontal? In this case the figure caption says the plot refers to

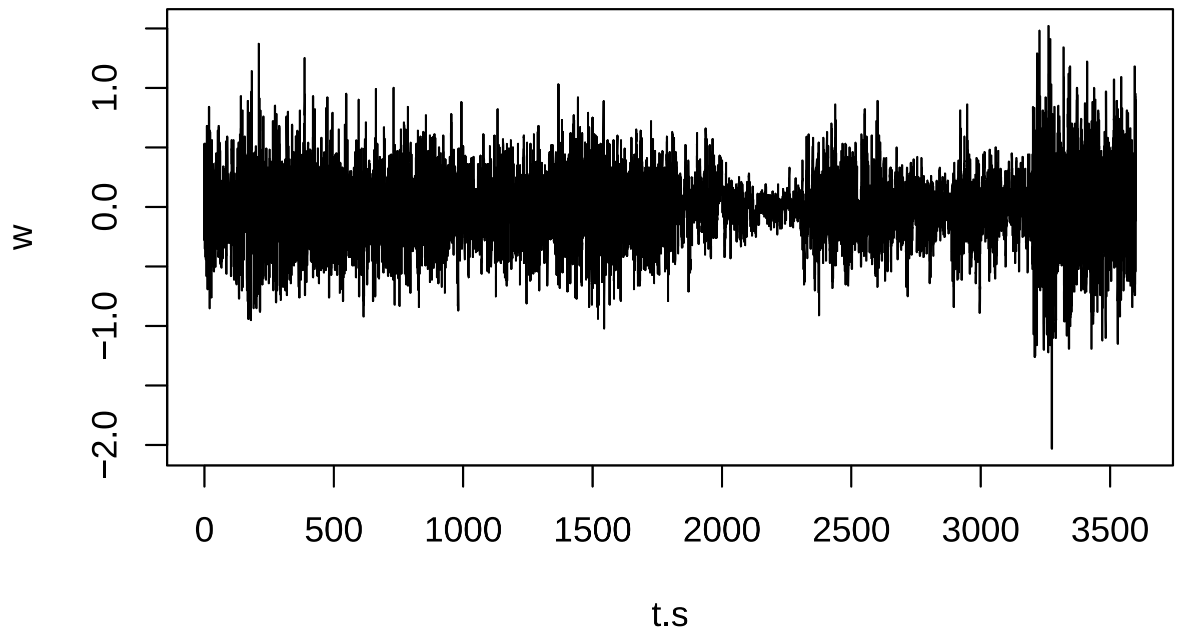


Figure 9.2: Variation of vertical wind speed with time, from 15:00 to 16:00 in CRA.01 set

an entire hour, and the range suggests seconds are used (this is consistent with the time stamp definition within SonicLib raw data files once read in memory). Seconds demands a lot of space to be printed: minutes may be preferable? And what of axis name? To tell, or to remove?

We try by improving the vertical axis, and removing axis name from the horizontal. We also convert to minutes:

```
plot(t.s/60, w, type="l", ylab="w (m/s)", xlab="")
```

Here the result:

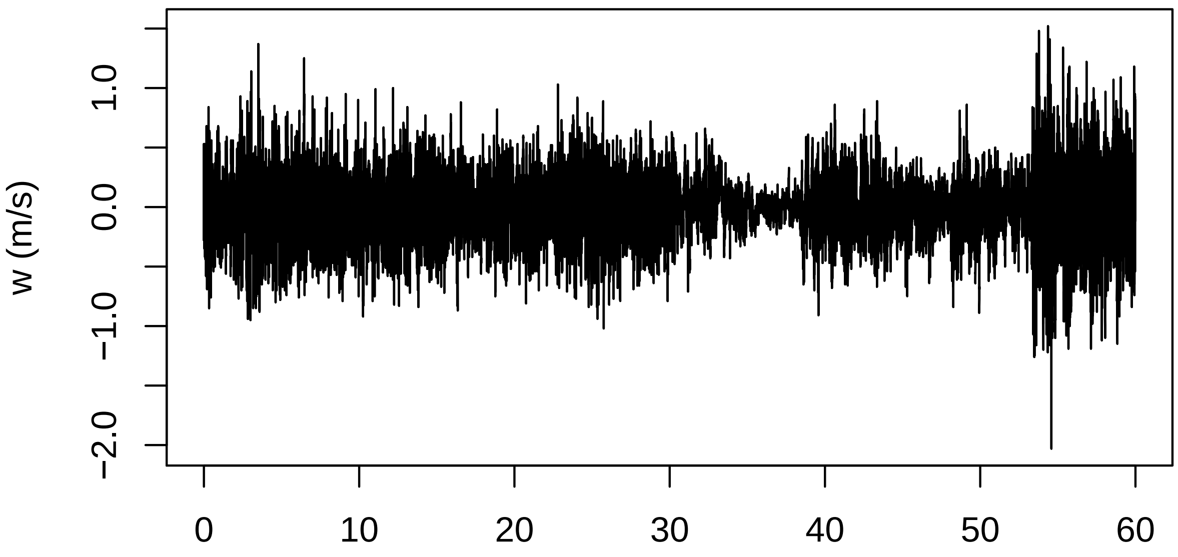


Figure 9.3: Variation of vertical wind speed with time, from 15:00 to 16:00 in CRA.01 set

Figure 9.3, too, is far from ideal, yet much better than 9.1. We might now try to place ticks not where R likes, but where our aesthetical sense could suggest. Or, me may adopt a lighter color for w.

All these improvements can be made adding parameters. We might go on, refining our plot further, but we imagine you got the message. Any details may be found in [32].

But, we notice an important thing. Our last graph is finally *useful*: it shows us a large change occurred in wind *variation* (a second order like statistics). This variation occurs exactly where we expected it, in the second half-hour. Last but not least, if placed in a presentation would make immediately the point even to a casual listener.

Before closing this example we love to highlight an important concept:

Plotting data involves of course learning R powerful graphics commands, but its real purpose is *telling a story*, and making a point. No story, poor graph. Before delving into the development of the fanciest plot it is advisable to think first what we would like to transmit.

This in turn demands a clear understanding of whom our story is aimed at. Stated differently, writing programs or using a top-notch software package is at least 90% human relation and communication, and 10% technology (and not be impressed by the commonly observed fact that industry rewards technology 90% and good communication 10%: it says a lot about the past, present and, likely, future "financial crises": Buggy by Design?)

Sure this manual has not space enough to get deep on this essential point, but fortunately there are many excellent references. Among them the authors have used (and found subjectively useful) [15] and [7].

We proceed now to check where the anomalous relative non-stationarity comes from. According to its definition, the relative non-stationarity is proportional to the difference between minimum and maximum value of signal regression line. As the regression line is so important, it would be nice showing it.

The first step to get distinct regression lines is partitioning data with respect to time in first and second half hour. We may accomplish this result, by using the powerful indexing capabilities of R on the time stamp:

```
t.lower <- t.s <= 1800
t.upper <- t.s >  1800
```

`t.lower` and `t.upper`, generated this way, are two boolean vectors of same length as `w`. These vectors can be used to select the upper and lower part of vertical wind speed:

```
w.lower <- w[t.lower]
w.upper <- w[t.upper]
```

We used time stamps as a selection tool, but it is not wise to employ them as independent variable in linear modeling. This, because in SonicLib raw data files time stamps are allowed to be integer (they are, in CRA.01 set) so repeated stamps are possible. An alternative is using positional indices, and we'll do this method. Partitioning indices is the same as with w; in addition we scale them so they can function as replacement of minutes.

```
idx <- 1:length(w)
idx.lower <- idx[t.lower]/600
idx.upper <- idx[t.upper]/600
```

The next step is computing the models themselves. This is done using command `lm()`, using partitioned indexes as independent variables, and partitioned w as dependent:

```
l.lower <- lm(w.lower~idx.lower)
l.upper <- lm(w.upper~idx.upper)
```

The result we obtained can then be inspected by giving commands `l.lower` and `l.upper` respectively. This is what we get with the lower half hour:

```
Call:
lm(formula = w.lower ~ idx.lower)

Coefficients:
(Intercept)    idx.lower
 -0.0422577    0.0004308
```

And this is the upper:

```
Call:
lm(formula = w.upper ~ idx.upper)

Coefficients:
(Intercept)    idx.upper
   0.059042    -0.001693
```

Note slope and intercept have opposite signs in the two cases. But values are small enough to make very difficult to predict visually what happens. We wish to display data and the regression lines, and our first attempt is by applying the command `abline()`, applied to the regression models after having plotted the actual data:

```
plot(idx/600, w, type="l", ylab="w (m/s)", xlab="")
abline(l.lower, col="red")
abline(l.upper, col="blue")
```

The result is shown in figure 9.4.

This is not the best we may do: according to the way `abline()` operates on regression lines, the whole span of independent variable is used, and this gives a really misleading view: it would be much better to plot the two lines only where they are defined. To do this, we could evaluate the regression models in correspondence of their definition intervals, then plot them.

The first step, evaluating the models in correspondence of the independent variable values, may be done the straightforward way once we know the regression coefficients:

```
m.lower <- l.lower$coefficients
m.upper <- l.lower$coefficients
```

Proceeding this way instead of copying their values shown previously we get coefficients in full precision. Because R is done, `m.lower[1]` represents the intercept and `m.lower[2]` the slope of regression model `l.lower` (and identically with `l.upper`).This allows to perform our estimation:

```
w.est.lower <- m.lower[1] + idx.lower*m.lower[2]
w.est.upper <- m.upper[1] + idx.upper*m.upper[2]
```

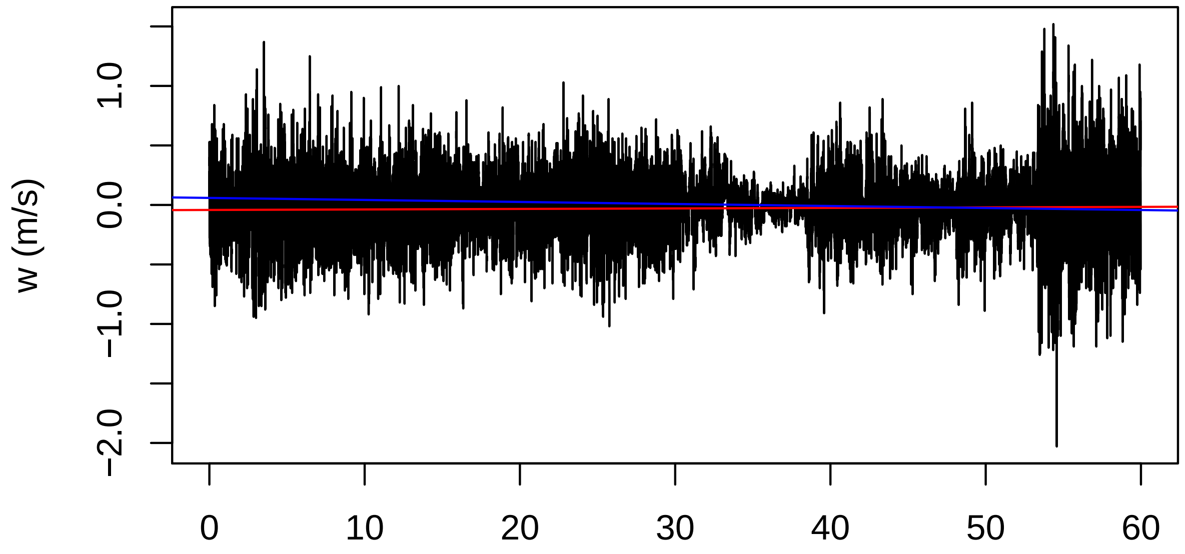


Figure 9.4: Variation of vertical wind speed with time, from 15:00 to 16:00 in CRA.01 set. In red the regression line computed on first half hour; in blue the regression line of second half hour.

We are now ready to plot:

```
plot(idx/600, w, type="l", ylab="w (m/s)", xlab="")
lines(idx.lower,w.est.lower,col="red",lwd=3)
lines(idx.upper,w.est.upper,col="blue",lwd=3)
```

The command `lines()` plots a line graph without removing the underlying graphics, as `plot()` would do by default. Parameter `lwd` specifies the line width, with a minimum value of 1; value 3 is a sensible choice to make the line visible. The result is 9.5.

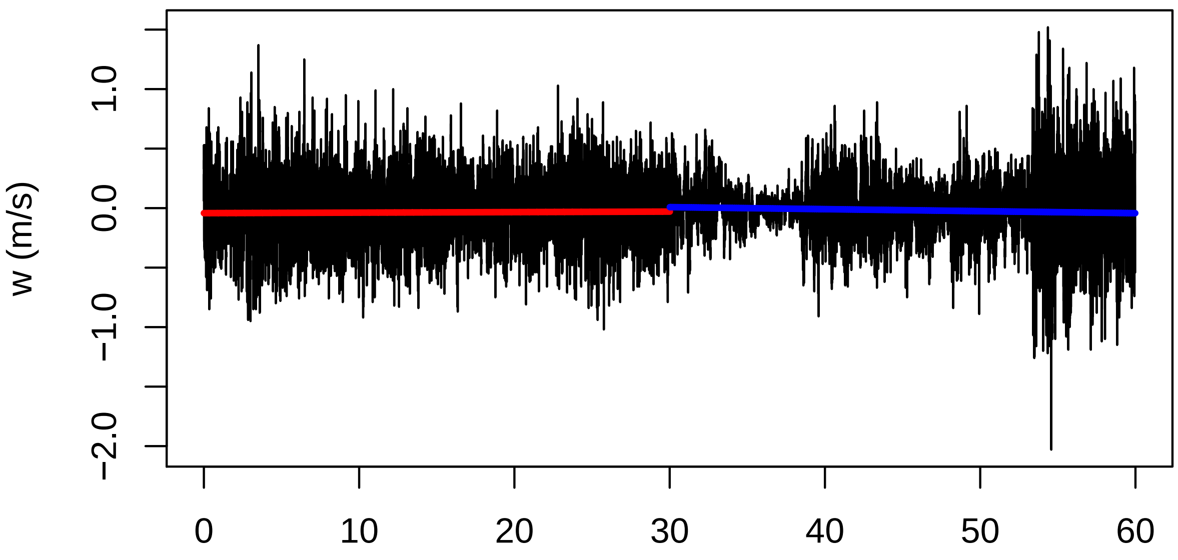


Figure 9.5: Variation of vertical wind speed with time, from 15:00 to 16:00 in CRA.01 set. In red the regression line computed on first half hour; in blue the regression line of second half hour.

The two regression lines are different, as we had seed when printing the model numerical values. But visually, their dissimilarity seems not striking.

9.1.2 Visualizing directional dependencies using R as it is

One common reason of non-uniformity is connected to the different "history" experienced by wind as it flows through different environments. An indicator of this history is wind provenance direction.

One way to display the dependence of a variable on wind direction is a scatter plot with the wind direction used as an independent variable. But such a diagram is quite uninformative if we employ a scatter plot of the usual kind, like the one shown in figure 9.6, showing the change of the ratio of friction velocity to the wind speed with respect of a "long" data set.

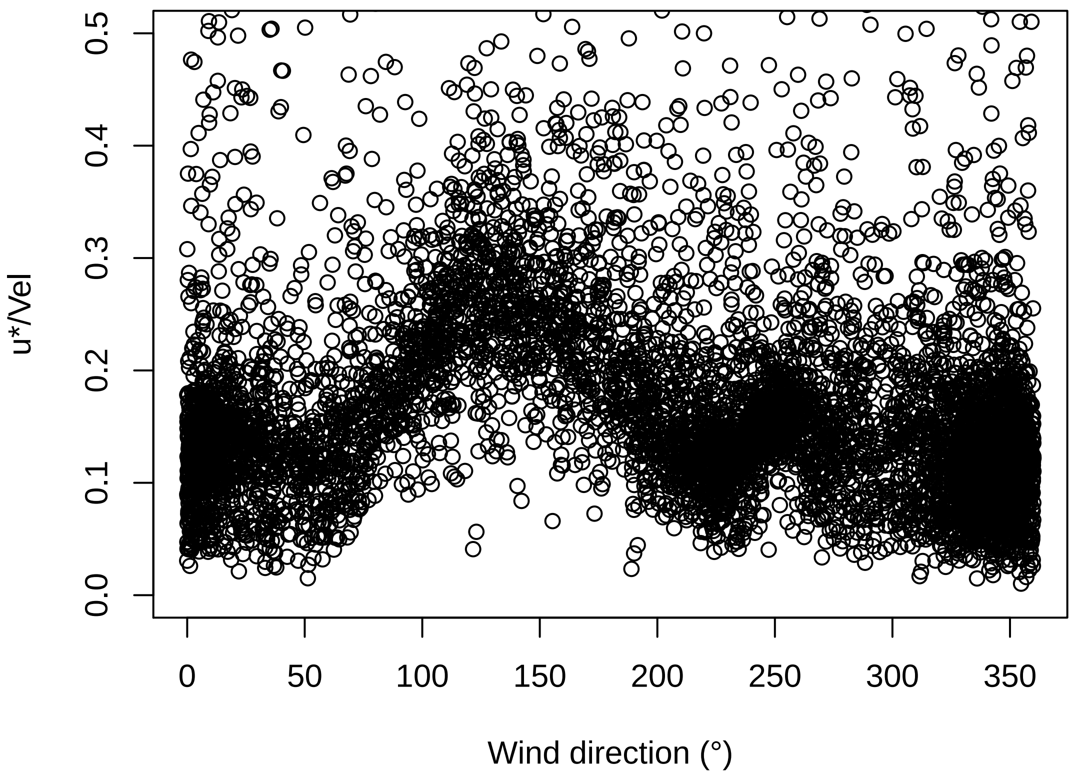


Figure 9.6: Directional scatter plot of u_*/U drawn using the standard "plot" command

The reason figure 9.6 is not the best is the heavy clutter of circles: if data are too much a significant danger exists they overlap in a way not allowing to appreciate where they're most concentrated.

An alternative in this case is using command `smoothScatter`, now part of the standard R distribution. The syntax is identical to `plot`'s, but instead of small circles in correspondence of data points, pixel in shades of gray are drawn proportional to the number of data points in the nearby. In practice, the higher the concentration, the darker the tone. As we may see in figure 9.7.

What happens around 130 degrees? As we may see, the ratio of friction

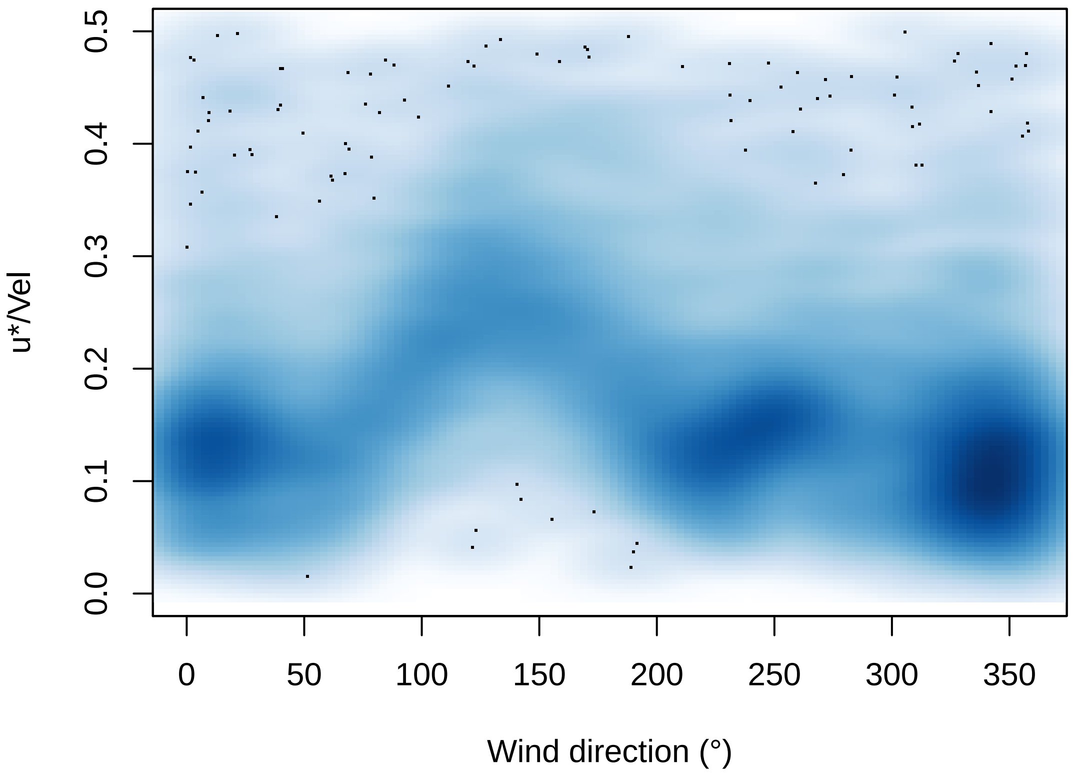


Figure 9.7: Directional scatter plot of u_*/U drawn using command "smooth-Scatter"

velocity to wind speed is close to a fashionable 10% in all directions, with the exception of the sector 100 to 170 degrees, where we can see a sharp rise to a 30%, followed by a complementary sharp decrease.

Visual inspection of aerial photographs of field site suggests a possible reason: a tall and wide tree standing to the SE of measurement point, at around 25 meters distance. In theory, fixed stations should be placed at least at a distance ten times the height of major obstructions at any possible direction. But this requirement is really difficult to fulfil in practice, especially in densely populated areas. And more so at *urban* sites.

If this is the case, a directional diagram of some key turbulence indicators may work well as a data quality assessment tool. It may also be helpful in quality *assurance*, for example by labeling data associated to wind direction from some critical directions. If “all” directions are critical, then a convincing argument for relocating an existing station is also easy to build.

9.2 SonicLib-specific Plots and Related Functions

9.2.1 Function dir.mean

Function use

To use `dir.mean` the following command should be issued:

```
dir.mean(
  dir,
  val,
  width,
  step,
  confidence,
  boot.size,
  conf.level
)
```

where

dir Vector containing direction values for each data set elements (degrees from North; no default).

val Vector, having the same dize as “dir”, and containing all values for each data set element (unit depends on actual value meanings; no default).

width Sector width (degrees; default = 11.25).

step Angle between the center directions of two consecutive steps (degrees; default = 1).

confidence Method to be used when computing confidence intervals (alphanumeric string, with possible values “none” and “boot”; default: “boot”).

boot.size Sample size to be used when computing confidence limits using bootstrap method (positive integer number; default = 1000; this value is ignored if “confidence” is not “boot”).

conf.level Value of the confidence level at which confidence intervals are computed (default = 0.95).

The result is an instance of an object `dir.stat` in case of successful completion, or NULL otherwise.

Separate or overlapping sectors?

If the function defaults are used, sectors are guaranteed to overlap widely as each of them is 11.25 degrees wide (half the familiar 16-wind-sectors rule), and the sector-to-sector angular spacing is just 1 degree.

This way, passing from a sector to the next many data values get the chance of being counted twice. Is this a big deal? In many cases "no" could be an appropriate answer: an indicative mean is just good.

This needs not be the universal case however. Proper selection of `width` and `step` allows customization. A common non-default choice consists in selecting 16 sectors, with `width=22.5` and `step=22.5`, in which case sectors overlap only on their boundary (a line, with zero area), behaving as practically separate.

Confidence intervals

Function `dir.mean` provides parameters allowing to estimate confidence intervals on each directional mean. The meaning of confidence interval is not to give an idea of dispersion of values around the mean (although it could be mathematically related to it), but rather to specify an interval such that the true directional mean belongs to it at a specified confidence level.

The true and computed directional means should in fact not be confused, the latter being an estimate of the former, whose actual value is not a priori known.

Confidence limits can be used as a visual tool (they will, using directional mean plotting functions within SonicLib), but may also be used numerically to solve common problems on statistically firmer ground than just proceding intuitively. Suppose for example you are interested in checking the value of friction velocity seen by wind coming from a given direction in, say, winter and summer - possibly to check for changes in turbulence generation by deciduous shrubs upstream.

Without confidence limits you'd presumably get the directional means computed in the two periods, compare them, and likely decide they are somewhat different. This is not a strong conclusion, indeed: the two numerical values of means along the same direction may not be equal, but this do not prove beyond any doubt their true values to be also different.

If we have confidence interval at hand, solving the problem in more meaningful terms is easier: it suffices to see whether the two confidence intervals have void intersection or not. In the first case you can be "reasonably sure" (within confidence level) the true directional means are different. In the second, you can't exclude their equality. Even if the overlap zone is small compared to interval overall size.

Similar considerations can be made for other problems of practical interest. For example: is temperature changing with wind direction, or not? Overall, confidence limits are a precious indication.

Examples

Using function `dir.mean` is straightforward: all you need are two equal size vectors holding wind direction and the quantity you want to average directionally. Data may actually be raw sonic wind vectors, averaged sonic data, eddy covariance results, chemical concentrations from given directions. Literally, whatever (in fact `dir.mean` could be used for applications having nothing in common with sonic anemometry).

Here is a very simple example from eddy covariance results:

```
e.dm<-dir.mean(e$data$Dir, e$data$u.star)
```

Once you have computed the directional mean you may do interesting things jut using numerical values. Here for example we check whether friction velocity is uniform with respect to wind directions or not, using Vertemate data set.

Of course, if we reason the simple way the answer will almost invariably be "friction velocity is not uniform with respect to wind direction". Indeed, the value of directional means will show some degree of change as we may see easily:

```
> min(e.dm$mean, na.rm=TRUE)
[1] 0.134949
> max(e.dm$mean, na.rm=TRUE)
[1] 0.4044134
```

The maximum directional mean found is three times the minimum! Little doubt they look "different". But this is true of the means computed from experimental data, that is, *estimations* of the true means. What about the latter?

To answer we need some more information: first of all, a more precise definition of what we mean by "not uniform with respect to direction". Second, but not less important, we need knowing where we may find the true value, and here is where confidence limits come in.

We begin with the definition of (non-)uniformity. We have many directional mean values, each computed from their equally-spaced sectors. On each sector, corresponding to its direction δ, we also have confidence limits, say them $l(\delta)$ and $u(\delta)$ with $l(\delta) \leq u(\delta)$.

Now get two distinct directions, δ_1 and δ_2. In case of non-uniformity their intervals have an empty intersection. Closed intervals on the real time are special cases of closed sets. The complement of a closed set is an open set. The complement of the intersection of two set is the union of their complements, which are both open. Now, the union of two open sets is also open. This might not seem on the first instance, but this is a very important conclusion: it implies the two non-intersecting closed sets are separated by an open sets which, by its very nature, is fat enough to contain quite huge subsets (closed included).

In terms of intervals, this means that either $u(\delta_1) < l(\delta_2)$ or $u(\delta_2) < l(\delta_1)$.

But: we have many confidence intervals, one per direction value, not just two. Saying a quantity is non-uniform could reasonably mean the confidence intervals of at least two directions do not overlap. In case of our set:

```
> max(e.dm$conf.min, na.rm=TRUE)
[1] 0.372677
```

```
> min(e.dm$conf.max, na.rm=TRUE)
[1] 0.1478555
```

The minimum of confidence intervals upper limits is smaller than the maximum of lower confidence limits, so at least two intervals do not overlap.

9.2.2 Function plot.dir.mean

Function use

To invoke `plot.dir.mean` a command like this is to be used:

```
plot.dir.mean(
  d.m,
  min.val,
  max.val,
  col,
  add,
  conf.limits,
  verbose
)
```

where

d.m is an instance of `dir.stat` type, described in section 3.6, and returned by function `dir.mean`.

min.val Minimum value to be used in plotting (no default).

max.val Maximum value to be used in plotting (no default).

col Color (alphanumeric string, defaulting to "black").

add Boolean, stating whether current plot is to be added (TRUE) to a currently existing directional mean plot, or not (FALSE) (default: FALSE).

conf.limits Boolean, specifying whether confidence limits should be plotted or not.

verbose Boolean, specifying whether error message are desired (TRUE) or not (FALSE) (default: FALSE).

How to select minimum and maximum values

Two of the function parameters specify the minimum and maximum values to be used as radial limits during plotting. Only the part of diagram within these limits are actually drawn, so a sensible selection is of paramount importance.

Directional mean plots are no different from other types of statistical plots. They adopt polar form, but selecting in different way the limits allows to present the same data letting people understand quite different things. This is the reason minimum and maximum values have no default: they are too important to ask SonicLib doing an arbitrary choice.

But this choice conduce to the need of discovering an interval guaranteeing the plot to be entirely visible. The conduct is different depending on whether confidence limits are to be plotted or not. In the first case, the minimum spanning interval is obtained using commands

```
min.val <- min(d.m$c95.inf)
max.val <- max(d.m$c95.sup)
```

where `d.m` denotes the directional mean object returned by function `dir.mean`. If confidence intervals are not interesting, the commands reduce to

```
min.val <- min(d.m$mean)
max.val <- max(d.m$mean)
```

The mere fact an interval allowing to plot the whole directional mean does not imply the corresponding graph makes actual sense. But at least provides us with a good starting point. Producing good graph is both a matter of sound science and good communication, and many guidelines can be found easily by surfing on the web or consulting any text of applied statistics. So we'll not pursue this important detail any more.

Examples

As directional statistics are so important as practical tool, in SonicLib a few functions address them specifically (in particular `dir.mean` to perform the actual computing and `plot.dir.mean` for the actual plotting). Using SonicLib to plot directional means yields both aesthetical and practical advantages: on one side it represents directions in polar form, for a more intuitive feeling of data meanings; on the other it allows to visualize quantities helping to assess the quality of data, as for example confidence limits.

Figure 9.8 shows the simplest possible graph of directional mean of friction velocity u_* plotted using SonicLib functions. The data set has been prepared by averaging raw data files in standard SonicLib format, then running standard eddy covariance (with no frills) on the results. Finally, directional means have been computed along with confidence limits using `dir.mean` function:

```
d<-average.sonic.file.set(file.name)
e<-eddy.covariance(d, 300)
e.dm<-dir.mean(e$data$Dir, e$data$u.star)
```

And this is the actual plot command:

```
plot.dir.mean(e.dm, 0, 0.5)
```

Plot 9.8 says not so much more than directional mean scatter plot as we've seen already. But does in a way allowing to better match, say, directions with a site map or photo. We may do more, however, for example to see how solid the directional mean is around the whole directions circle.

We can check this using confidence limits, as in figure 9.9, obtained using a slightly revised form of the plot command:

```
plot.dir.mean(e.dm, 0, 0.5,
   conf.limits=TRUE)
```

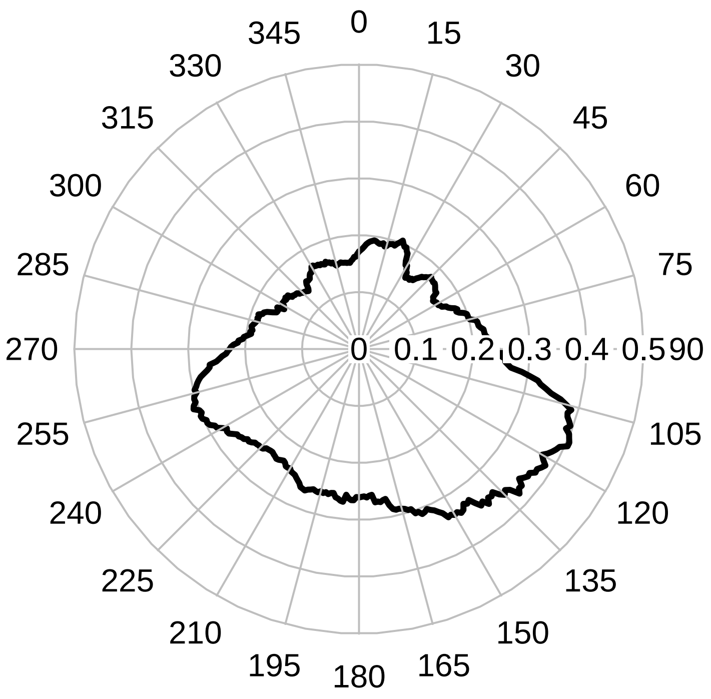


Figure 9.8: Directional plot of u_* drawn using command "plot.dir.mean" without confidence limits

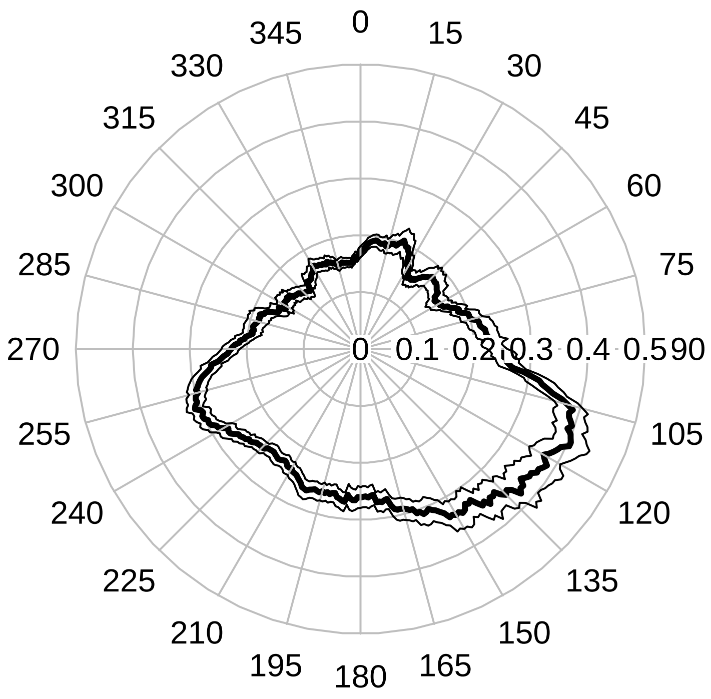


Figure 9.9: Directional plot of u_* drawn using command “plot.dir.mean” with confidence limits

9.2.3 Function plot.dir.num

Function use

The width of confidence interval increases with decreasing sample size, so it is sometimes important to have an idea of how many data are present in each direction sector. This result is accomplished by a specific function:

```
plot.dir.num(d.m,col,add)
```

where

d.m is an instance of `dir.stat` type, described in section 3.6, and returned by function `dir.mean`.

col Color (alphanumeric string, defaulting to "black").

add Boolean, stating whether current plot is to be added (TRUE) to a currently existing directional mean plot, or not (FALSE) (default: FALSE).

Examples

We use the same friction velocity data seen in `plot.dir.mean`, with results in figure 9.10.

Obtaining the graph has been very simple:

```
plot.dir.num(e.dm)
```

where `e.dm` is the same u_* directional average set used in the description of `plot.dir.mean`.

Despite being an extremely simple test, the numerosity plot often yields interesting information. In our case for example we may distinguish two very strong dominant directions, around 240 and 345 degrees. Other directions, for example from 45 degrees, are very much less represented.

Cases like this may happen if a very short period is considered, as in our test data set. Would a similar situation occur over a longer period (one year or more) the suspect would be some directions are severely screened by nearby obstructions. This indeed is not a rare finding, especially in urban and peri-urban situations.

9.3 Using statistics

As a statistical system, R is well versed to answer questions in a statistically founded way. This may be an important advantage, in a field characterized by large sets of data sampled with high accuracy, but also by a large intrinsic variability.

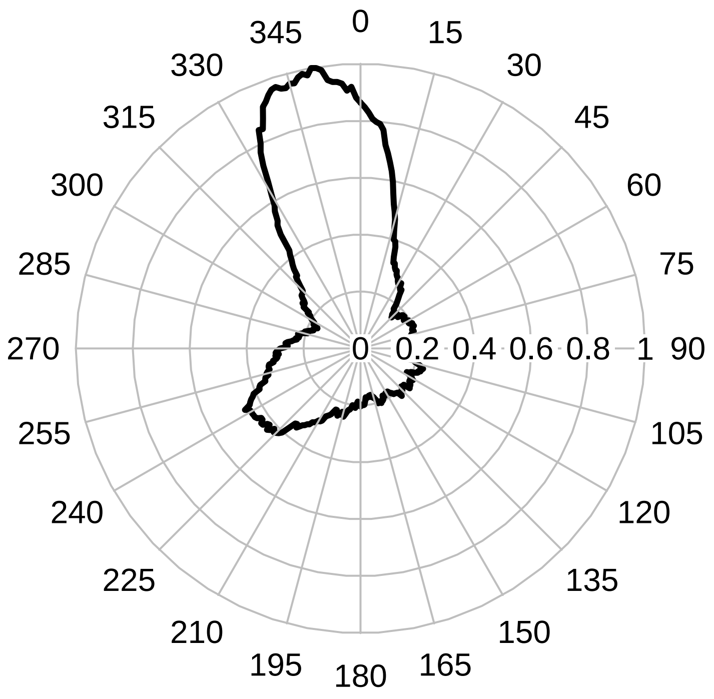


Figure 9.10: Directional plot of data numerosity for u_* set obtained using "plot.dir.num"

Which direction is wind blowing from?

Short-range pollutant dispersion

One useful application of ultrasonic anemometers is performing high-accuracy measurements of mean wind and turbulent to feed atmospheric pollutant dispersion models. In this respect, besides seeing things normal anemometers do not sense, ultrasonic anemometers are also immune from "wind calms" and "variable winds", that is when wind speed is not fast enough to allow movement of the delicate mechanical parts constituting conventional anemometers. Immunity to wind calms and variable winds is very important in air quality studies because these condition correspond to the most dangerous situations, when transport by mean wind is minimum thus favoring the accumulation of large bulks of concentration.

Existing pollutant dispersion models are complex mathematical tools, whose description is beyond the scope of this work. What's important to our purposes is these models greatly benefit when fed with the least "cooked" averages possible. Basically, dispersion models may be given meteorological input ranging from a minimum, when wind speed and direction and air temperature are sufficient, to complex sets with turbulence data. Many models allow to specify various forms for meteorological input, all differing by the detail and quality of observation.

Surely the easiest way to use a model is feeding it with the less detailed meteorological data set possible. But in this case the model will derive all the turbulence data it needs for its internal working by *estimates*, obtained applying various algorithms to the more or less scant input data. The less detailed the input meteo data, the more aggressive the estimation process. But also, the more assumption will be made behind the scene, and the larger the error will be. In extreme case, the model estimates are made using assumptions so far from reality that results are little more than fantasy work.

As an example, let's consider turbulence and the diffusion associated to it. A very popular way to express turbulence, at least until some years ago, was by using *stability categories*. The introduction of stability categories to express turbulent diffusion quaitatively dates bach to the times of First World War, when empirical observation were made on field about the "effectiveness" of chemical aggressives depending on wheather conditions and time of day. Gradually various classifications emerged, not exactly comparable to each other, but all expressing the same basic concept.

Clearly stability categories are not a strictly quantitative data, and their use in models also gave rise to the production of output with a quantitative appearance (e.g. time series of ground concentration fields), but with a deeply qualitative nature which most practitioners (engineers with unsophisticate mathematical skills) did not appreciate.

Of course dispersion models can not use stability category directly. Rather, they translate them to "plume dispersion indicators" like the "downwind horizontal and vertical standard deviations of concentration" in Gaussian models. Horizontal and vertical standard deviations are functions of downwind distance from emission point, whose mathematical nature is usually quite simple (for example, in one of the most used scheme, due to Briggs, the plume dispersions are power laws whose coefficients change with stability category, and on whether a "rural" or "urban" case is in use).

Some models, more recent, allow to give turbulence information in quantitative form, but use a rather indirect approach. Instead of being satisfied of estimating the sigmas from the qualitative information stored in stability category, they estimate them from turbulence indicators like Obukhov length or friction velocity (both readily obtained from ultrasonic anemometer data using eddy covariance).

This is too an estimate. But is less aggressive and its output changes continuously with input values: we're in front to truly *quantitative* data.

A further step is in using *measured* sigmas directly, by relating the dispersion of a plume to staistical characteristics of wind as it is. Of course the sigmas of plume are not the same thing as the standard deviation of wind in the three directions, but a strong physical connection can be reasonably expected among them.

The last possible step is, using ultrasonic anemometer data themselves, turbulent fluctuations included.

Chapter 10

Auxiliary. But no less useful!

10.1 Giving time a meaning

In R exist some powerful facilities allowing to represent dates and times. Two of them, the data types `POSIXct` and `POSIXlt` along with their associated functions, are frequently used in SonicLib to represent time stamps of the most varied kinds.

These data structures allow easy conversion of dates and times, but unfortunately do not address all the nitty-gritty issues arising in ultrasonic anemometry. There is no reason to panic however: R can be easily extended - and we've just done, introducing a bunch of little utility functions.

10.1.1 Generating time sequences

It sometimes happens needing to check for data completeness with respect to time. In these cases it comes handy to have a simple tool to generate equally spaced sequences of POSIXct data spanning a given time interval.

Needless to say, standard R has a solution simple enough to avoid any programming: the `seq()` command. We see how to use it in the most common situation by an example. Suppose we want to generate all hours between a first time, say

```
date1<-as.POSIXct("2011-12-27 00:00:00", tz="UTC")
```

and a second, imagine

```
date2<-as.POSIXct("2012-01-04 23:59:59", tz="UTC")
```

As any POSIXct quantity, both `date1` and `date2` are represented as seconds from the Epoch, that is midnight of January 1st, 1970. One hour is 3600 seconds, so our sequence is

```
s <- seq(from=date1, to=date2, by=3600)
```

That's all!

10.1.2 Building higher-level time stamps

When reporting data it happens quite often that raw measurements are aggregated as hourly, or 30 minutes, or whatever, averages. And that these averages are further aggregated in, say, daily or yearly averages.

When this is done the problem immediately arises to attribute each of these higher-order average its own time stamp.

This task is generally speaking quite simple, but occurs often enough to suggest writing little routines worth saving just a few lines of recurring code.

A few special cases are common in ultrasonic anemometry, and we dovote to them the sub-sections of this section.

Function 'floor.day'

The first function dealing with aggregate time stamps is devoted to daily averages, one of whose common uses is detecting seasonal patterns without the need to plot all raw data (with the likely effect of filling the plot space with a black band, as horrible aesthetically as uncommunicative).

To invoke the function just do this:

```
floor.day(date.time, time.zone)
```

The first parameter, `date.time`, is a vector of `POSIXct` values containing an ordered sequence of time stamps, while the second, `time.zone`, is a string indicating which time zone is to be used for time transformation, with default "UTC".

The result is once again a vector of `POSIXct` elements, the same size as `date.time`, containing time stamps obtained from the input vector's by clipping all time stamps to the beginning of the corresponding day.

To make the routine simple no special check has been implemented, so the user is warned to employ it reasonably. In particular, we strongly advise passing `date.time` and `time.zone` of the same length, and the right types.

Also, some care must be exerted when using time ones different from "UTC", as the resulting times may yield some confusion. In many cases it may be better to invoke this function (and the other allowing to specify time zones) accepting the "UTC" default even if a non-UTC zone is dealt with.

Avoiding repetitions Used in the simplest way, as for example in

```
floor.day(s[2:(length(s)-6)])
```

on the same data set `s` generated in the preceding section, the call to `floor.day` will yield quite a long sequence containing many repetitions (on average, 24 per days). This may seem to be overkill, but is justified by added flexibility when higher order time stamps have to be assigned to data in a fine-grained time sequence.

In some cases it is useful to have *non-repeating* sequences of time stamps. Instead of writing a special purpose version of the command we may resort to the large library of predefined R commands, like so:

```
unique()floor.day(s[2:(length(s)-6)]))
```

The `unique()` command takes a sequence as input, and produces another sequence on output with all consecutive repetitions removed so that only unique values are obtained.

Function 'floor.year'

Function `floor.year` is very similar to `floor.day`, with the difference that it clips the POSIXct input values to the first instant in their corresponding year, allowing to label easily yearly averages and other statistics.

Its interface is practically indistinguishable from its daily counterpart:

```
floor.year(date.time, time.zone)
```

The same cautions apply, too.

Moving time forth and back

It often happens to need converting a date expressed, say, as the number of days since year beginning, to a more conventional date and time value which can be immediately located on a normal calendar. For these cases SonicLib provides a specific function:

```
date.shift(ref.instant, num.days, time.zone)
```

where

- `ref.instant` is a POSIXct value meaning the reference time to be used (most often the beginning instant of a year).
- `num.days` is an integer number, or vector of integer numbers, containing the displacement from reference instant expressed in whole days (positive, zero or negative values are all right).
- `time.zone` is a string encoding the time zone (default value is "UTC").

On exit the function yields a (vector of) POSIXct value(s). The function meaning is quite straightforward, but an example makes it even easier. Suppose we want to attribute lazy-readable date and time values to the two days, 104 and 181, used in AMERIFLUX open-path golden standard data set (see section 4.3).

First of all we need to prepare a vector containing the two displacements we need:

```
dspl <- c(104, 181)
```

This made, we have to select a reference time. In AMERIFLUX Gold Standard documents I've not found a reference to a year, but this is not a big deal, as we may invent one on ourselves (knowledge of year is mandated by SonicLib, but is not relevant to validation against AMERIFLUX data). In lack of any specific constraint we may then decide to use, say, year 2012, whose first instant is

```
ref <- as.POSIXct("2012-01-01 00:00:00", tz="UTC")
```

With these data items defined we can finally invoke our function,

```
date.shift(ref, num.days=dspl)
```

And here is the result:

```
[1] "2012-04-14 UTC" "2012-06-30 UTC"
```

10.1.3 Computing time-related indexes

In addition to time stamping, there are many situations where time-related indexes are of some utility. One example is when integer or factor values are needed to compute typical days and statistics alike.

Time related indexes can be broadly sorted into two large classes: unlimited, and periodic. Unlimited indexes are suitable to assume any integer value, as happens for example for year. This means not a realistic series will span many distinct values: in most cases a relatively short sequence will be used. But *in principle* the possibility of spanning an indefinitely large interval exists.

Periodic indexes, on the contrary, wind up after some times. For example, hours run from 0 up to 23, to return to 0 on the next index increase. Periodic indexes can then assume only a finite number of values, up to a limit well known in advance.

But: let's just see which indexes are defined in SonicLib. We'll encounter them all in the next sub-sections.

Function 'julian.day'

Sometimes it comes handy to associate a unique integer index to any day. One such association is the Julian day, often used in applications. Notice the uppercase 'J': the Julian day is a very well defined concept for which a complicate definition exists.

In SonicLib the average need is more modest: to know the number of days have expired from a given reference day to a specified date. This day number can be used to calculate the difference (measured in days) between two dates, or in astronomical calculations of the kind often found in meteorological processors. So we speak of julian day (with lowercase 'j') to indicate this count.

Computing the SonicLib julian day is accomplished through function

```
julian.day(date.time, base.time)
```

Function input are `date.time`, a vector of POSIXct values, and `base.time`, another POSIXct value indicating the reference and defaulting to the Epoch,

```
as.POSIXct("1970-01-01 00:00:00", "UTC")
```

In principle, `base.time` may not be the exact beginning of a day. But in this case the non-zero hour to use will define a sort of day start for any counting purposes. Our recommendation is using this possibility sparingly, only when absolutely necessary (in the very vast majority of cases the default Epoch is all what really needed).

Use of non-Epoch reference times is occasionally of use, however (and also in these cases the reference occurs at the beginning of a day).

Function ’year.number’

Function ’month.number’

Function ’day.in.year’

Function ’step.in.day’

10.1.4 Forming typical years, days, and more

Bibliography

[1] Multiscale and multiresolution methods: Theory and application.

[2] *The principles of surface flux physics: theory, practice and description of the ECPACK library.*

[3] *The Web of Inclusion.* Doubleday Business, 1995.

[4] D.B. Percival A.T. Walden. *Wavelet Methods for Time Series Analysis.* Cambridge University Press, 2000.

[5] T. Foken B. Wichura. Tools for quality assessment of surface-based flux measurements. *Agric For Meteorol*, 1996.

[6] W.N. Venables B.D. Ripley. *S Programming.* Springer, 2000.

[7] W.N. Venables B.D. Ripley. *Modern Applied Statistics with S.* Springer, 2002.

[8] M. Aubinet C. Feigenwinter B. Heinesch C. Bernhofer E. Canepa A. Lindroth L. Montagnani C. Rebmann P. Sedlak E. Van Gorsel. Direct advection measurements do not help to solve the night-time CO_2 closure problem: Evidence from three different forests. *Agric For Meteorol*, 2010.

[9] R.K. Sakai D.R. Fitzjarrald K.E. Moore. Importance of low-frequency contributions to eddy fluxes observed over rough surfaces. *J Appl Meteorol*, 2001.

[10] E. Canepa E. Georgieva G. Manca C. Feigenwinter. Application of a mass consistent flow model to study the CO_2 mass balance of forests. *Agric For Meteorol*, 2010.

[11] P. Schotanus F.T.M. Nieuwstadt H.A.R. De Bruin. Temperature measurement with a sonic anemometer and its application to heat and moisture flux. *Bound Layer Meteorol*, 1983.

[12] L. Montagnani G. Manca E. Canepa E. Georgieva. Assessing the method-specific differences in quantification of CO_2 advection at three forest sites during the advex campaign. *Agric For Meteorol*, 2010.

[13] E.K. Webb G.I. Pearman R. Leuning. Correction of the flux measurements for density effects due to heat and water vapour transfer. *Q J R Meteorol Soc*, 1980.

[14] M.A.R. Ferreira H.K.H. Lee. *Multiscale Modeling: A Bayesian Approach.* Springer, 2007.

[15] J. Maindonald J. Braun. *Data Analysis and Graphics Using R - An Example-Based Approach.* Cambridge University Press, 2007.

[16] J.C. Kaimal J.E. Gaynor. Another look at sonic anemometry. *Bound Layer Meteorol*, 1991.

[17] J.C. Kaimal J.J. Finnigan. *Atmospheric boundary layer flows.* Oxford University Press, 1994.

[18] D. Vickers L. Mahrt. Quality control and flux sampling problems for tower and aircraft data. *J Atmos Ocean Technol*, 1997.

[19] J.F. Howell L. Mahrt. Multiresolution flux decomposition. *Bound Layer Meteorol*, 1997.

[20] A.J. Chorin O.H. Hald. *Stochastic Tools in Mathematics and Science.* Springer, 2006.

[21] R.H. Waring S.W. Running. *Forest Ecosystems - Analysis at Multiple Scales.* Elsevier, 2007.

[22] R. Sozzi T. Georgiadis M. Valentini. *Introduzione alla turbolenza atmosferica - Concetti, stime, misure.* Pitagora Editrice, 2002.

[23] M. Aubinet T. Vesala D. Papale, editor. *Eddy Covariance - A Practical Guide to Measurement and Data Analysis.* Springer, 2012.

[24] S.P. Arya. *Introduction to Micrometeorology.* Academic Press, 2001.

[25] T. Foken. *Micrometeorology.* Springer, 2008.

[26] J.R. Garratt. *The Atmospheric Boundary Layer.* Cambridge University Press, 1992.

[27] F.C. Klebaner. *Introduction to Stochastic Calculus with Applications.* Imperial College Press, 2005.

[28] M. Leeder. *Sedimentology and Sedimentary Basins - From Turbulence to Tectonics.* Wiley-Blackwell, 2011.

[29] P. Murrell. *R Graphics.* Chapman & Hall, 2006.

[30] P.S. Nobel. *Physicochemical and Environmental Plant Physiology.* Academic Press, 2009.

[31] T.R. Oke. *Boundary Layer Climates.* Routledge, 1978.

[32] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[33] R.B. Stull. *An Introduction to Boundary Layer Meteorology.* Kluwer, 1988.

[34] R. Valentini, editor. *Fluxes of Carbon, Water and Energy of European Forest.* Springer, 2002.