eseis - A toolbox for environmental seismology

Michael Dietze

July 30, 2016

Basic concepts

Level of knowledge

This tutorial is intended to provide an overview of how to use the R-package eseis to handle seismic data. Thus, it requires a bit of prior knowledge about R and seismic data. Regarding R, it is assumed that you are familiar with the overall concept or R as a language and software. You should know about when to use which data types and data structures, how to convert them between each other, how to write, run and debug R-scripts, how to import ASCII files and save data sets. You should have some feeling of combining functions in scripts and passing or transforming their output to make them working for your ideas.

This tutorial cannot bring you to this level of expertise. If you are looking for gentle introductions to R, there is such a rich body of excellent online documents in the web, it is hard to point at the right ones. For basic introductions see for example Longhow Lams ebook. A very good resource for more advanced topics related to R is Hadley Wickhams Advanced R site. I have crafted a set of slides, mainly focused on introducing R to understand how end-member modelling of grain-size data works. These give a very brief outline of the essences of R and can be found GIVE LINK WHEN WEBSITE IS ONLINE.

You should also be familiar with the basic tools of a seismologist and what signal processing is about. Most of the functions do nothing else than these basic data preparation steps and the tutorial gives a brief motivation and justification for the use of each of them. Nevertheless, do not expect a reference base of seismic signal processing, here.

Balancing speed, user-friendliness, transparency and modifiability

An alternative pharse for this heading could have been: Why another (yet so unevolved) software for seismic data analysis? The answer is easy: environmental seismology is a very young field of science and sits right at the seams of seismology and many other Earth sciences. Thus, it is essential to offer a flexible tool, easy to understand and to learn by enthusiasts from a wide range of scientific fields, not just seismologists. R is the only software and language that can claim the truth of this statement. Beyond this it is the only such software, which is absolutely free and open to use.

R is slow, very slow compared to low level programming languages, such as C++. However, if needed R can speed up, although this requires the poor person writing the code must understand a bit more than "just writing R-scripts". But don't worry, this is not needed to use a package. It is required to develop a package. R is designed to be a very user friendly second skin around awkwardly readable computer languages. Writing R-scripts can be (with some imagination) like writing poems. In essence, you will write down a cooking recipe that your computer will go through and prepare. If the resulting meal is a tasty one depends on how well you wrote the recipe. But honestly, R is a very easy to learn language that can be really fast when appropriately written, it is as transparent as can be (the source code or definition of every function can be inspected and even changed if needed) and is also extremely flexible; nearly everything you create in R can be modified without worrying about much and can be passed to further analysis tortures using another of the almost 9000 packages hosted at the Comprehensive R Archive Network (CRAN).

So in essence, there is a need for another software to handle seismic data, for the purpose of linking seismic signals to Earth surface dynamics by a broad and open scientific community.

Vectorisation and parallelisation

Most functions can be applied to vectors or lists of vectors. This makes best use of the flexibility of R. The function will automatically apply itself to each vector in the input list using the general function `lapply`. The function can also be applied to matrices with signal traces organised as rows, using for example:

```
apply(X = data_matrix,
      MARGIN = 1,
      FUN = signal_detrend)
```

Where appropriate, signals can also be processed in a multicore environment, using the standard R function parLapply:

```
## get number of CPU cores
cores <- parallel::detectCores()

## initiate the cluster
cl <- parallel::makeCluster(getOption("mc.cores", cores))

## apply function in parallel mode
parallel::parLapply(cl = cl,
                    X = data_list,
                    fun = signal_deconvolve,
                    dt = 1/200)

## stop cluster
parallel::stopCluster(cl = cl)
```

With a leap of faith you are not disencouraged by the code you have (hopefully) read just above. The rest of the tutorial is less haevy. However, it would be perfect if the above section has felt not too far out. It used typcal R jargon and some parts of the further content might follow this theme. If you struggle at any time with the steps dicussed here or find issues with the package, please don't think it is your fault! There is a very good chance that I just missed typing the right letter at the required position. Please, write me a short email and let me know about the problem! See my website for contact details.

Getting started

A good software to effectively work with R is the open development environment RStudio. It can be downloaded and installed for many operating systems. See the RStudio-website for further information and for downloading the software. Make sure you have installed the latest version of R before installing R-Studio.

After successful installation of R and RStudio, it is time to install the R-package `eseis`. It is hosted on GitHub. This means all functions, documentation files, example data and additional package contents are kept up to date in a coherent and transparent way. To install the package, the easiest way is to use the package `devtools`, which provides convenient access to GitHub.

```
devtools::install_github(repo = "coffeemuggler/eseis", ref = "0.3.0")
```

If you choose for any reason not to use `devtools`, you can also download the source files of the package at any time from the `eseis` website directly. However, the package versions hosted there may not at the very latest stage, so check the time stamp in comparison to GitHub. The source file archives can be found here: eseis-website.

Starting with a new project from scratch a script should start with loading the package `eseis`, which will make all the functions available.

```
library("eseis")
```

That was not too hard, right? Actually, most of the time you will spend on getting your data in the right structure to use the package efficiently, not to tweak the package to process your data effectively (at least this is my hope). There are some commented code snippets at the end of the document that might help bridging the gaps getween the chain of functions discussed next. SET A LINK TO THE LAST CHAPTER!

Reading seismic data

The package `eseis` supports the two most commonly used formats of seismic data: `sac` (see IRIS website for more information about the format and its structure) and `mseed` (miniseed, see IRIS website for short description and FDSN website for the full reference manual).

```
x <- read_sac(file = "~/data/sac/file_1.bhz")
y <- read_mseed(file = "~/data/mseed/file_1.bhz")
```

The two import functions read one or more consecutive binary files and return one `eseis`-object, i.e. a list element with four elements:

- `$signal` - the signal vector
- `$time` - the corresponding time vector in `POSIXct` format
- `$meta` - another list with meta information
- `$header` - imported header information, also a list

While the `meta` part contains the same data regardless if a `sac` or `mseed` file was imported, the `header` part contains data type-specific information and is just kept for completeness. It is not used by the package. To see the full content of the imported object use the function `str()`. The `meta` part includes information about the imported file names, station, network, component, location, start time, number of samples and sampling period and (empty) fields of sensor and logger type. To inspect the `meta` part content use `x$meta`.

It is possible to create one long consecutive object from, for example many hourly seismic files. This requires submitting a vector of the desired file names to the import function and setting the argument `append = TRUE`, which is the default option.

```
x <- read_sac(file = c("~/data/sac/file_1.bhz",
                       "~/data/sac/file_2.bhz",
                       "~/data/sac/file_3.bhz"))
```

Likewise, it is possible to read three or more signals at once but not to append them but rather keep them as separate traces. This is for example useful when three component signals are used.

```
rockfall_xyz <- read_sac(file = c("~/data/sac/file_1.bhe",
                                  "~/data/sac/file_1.bhn",
                                  "~/data/sac/file_1.bhz"))
```

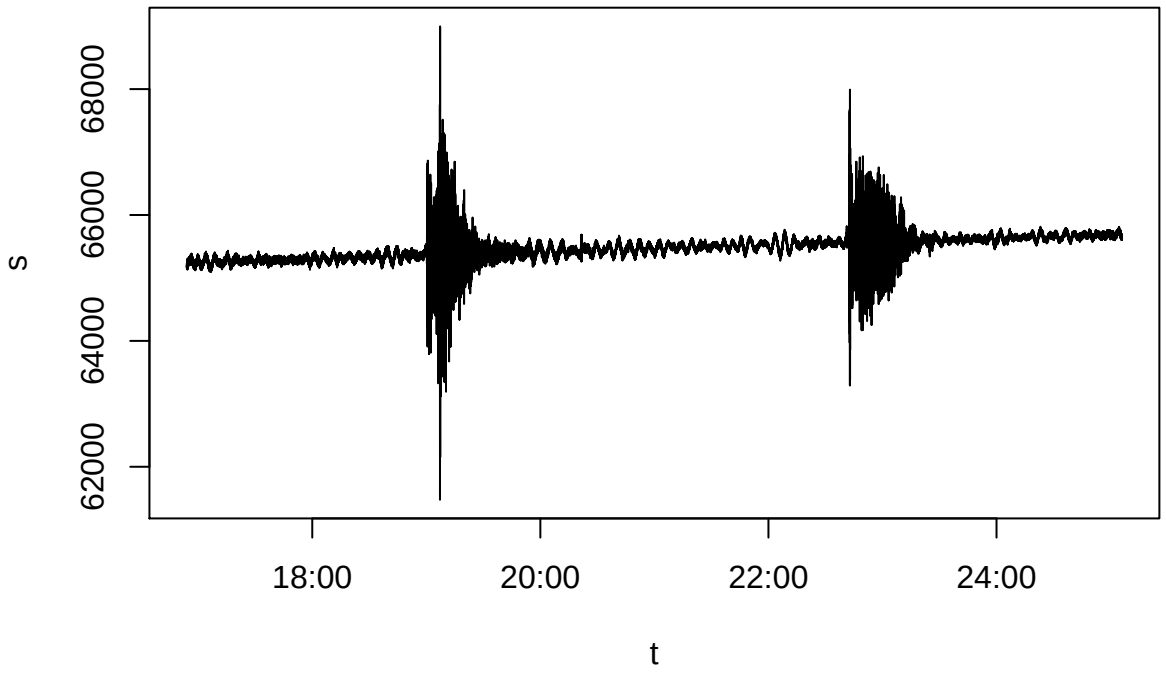
In this case, the resulting object is a list of the same length as the number of input files. Each imported file is organised as described above but in its own list element.

To avoid redundancies and save time and memory it is possible to restrict the import to any of the four elements listed above. To do so, the arguments of the elements to be omitted can be specified.

```
x <- read_sac(file = "~/data/sac/file_1.bhz",
              time = FALSE,
              meta = FALSE,
              header = FALSE)
```

Usually one will not work with the raw import object but rather the signal part and sometimes the time part. To isolate these vectors use the `$`-operator. The result may be plotted straightforward:

```
t <- x$time
s <- x$signal

plot(x = t, y = s, type = "l")
```



Export is currently only possible for the `sac` format using the function `write_sac()`. It will need the following information to create a `sac` file: a signal vector, the target file name, either the corresponding time vector or the start time and sampling period, as well as optionally the component, unit, location and network information. It is also possoible to provide the entire list of sac-file parameters as a list (`parameters`), which will override the previous settings and is only recommended in special cases. The file parameters can be obtained by calling the function `list_parameters()` and may be modified accordingly.

```
## the quick-and-dirty way
write_sac(data = s, file = "sacfile.sac", time = t)

## the a-bit-more-tidy way
write_sac(data = s,
          file = "sacfile.sac",
          time = t,
          component = "BHZ",
          unit = "counts",
          station = "LAU05",
          location = c(5157766,
                       416029,
                       900,
                       0.5),
          network = "LAU")
```

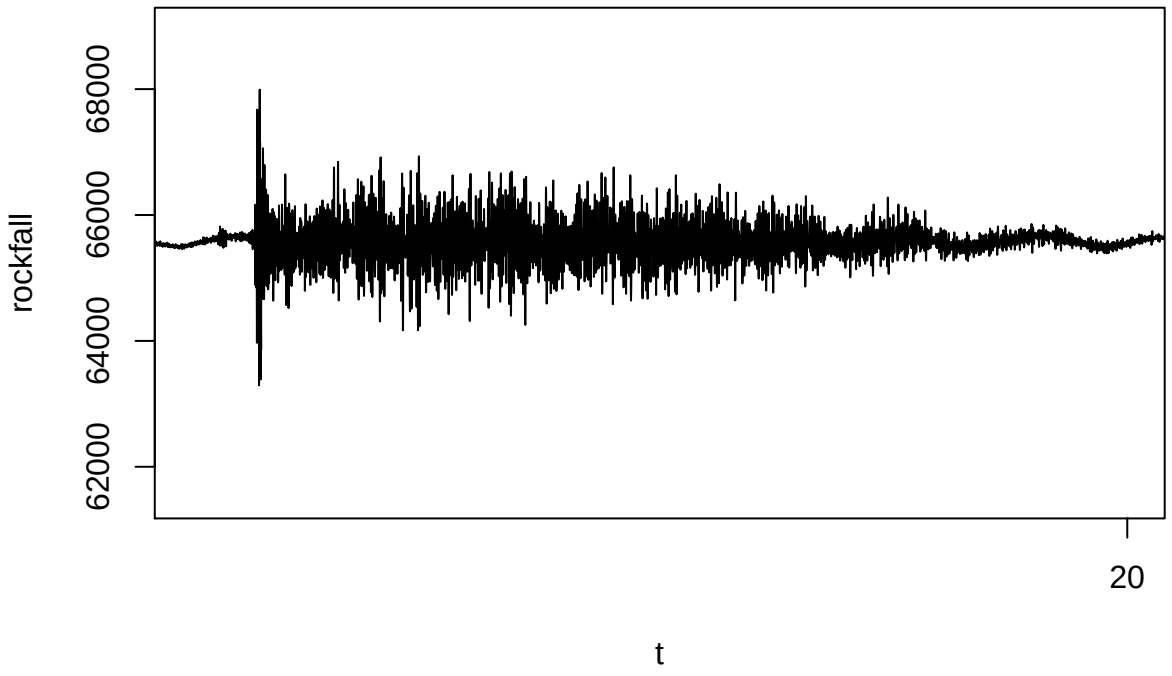
Handling the time format

One of the peculiarities of data in environmental seismology is that it depicts processes that last from less than a second (e.g., boulders impacting along a hillslope) to several hours (e.g., river bedload activity or rain) or even days (e.g., storm events). Thus, there is no universal unit for the time vector associated with each seismic signal vector. The most appropriate units may range from milliseconds to months, e.g., when plotting a spectrogram of environmental activity of an entire season. Furthermore, usually a series of signal vectors from the different seismic stations of an array is investigated. Therefore, a suitable and consistent time handling is essential.

In R there exist many different time formats, each for its specific purpose. In `eseis` the format `POSIXct` (Portable Operating System Interface Calendar Time) is used. This appears to be a complex thing in the beginning but is the most logical solution actually. A `POSIXct` date value describes the number of seconds that have passed since a given origin date, `1970-01-01 00:00:00` by default. A `POSIXct` date also supports setting of and switching between time zones and can be converted into many different date and time formats. It can be used for calculating and also allows flexible plots spanning millesconds to millennia.

The drawback of using `POSIXct` instead of, say seconds, is that one always needs to work with the entire time string when indicating a given time. A plastic example may be when truncating the x-axis of a signal plot. Even though the time axis may show seconds or minutes as label, trucation only works if the axes limits are given as `POSIXct`. Plotting for example the rockfall signal (i.e., the example data set `data(rockfall)`) only when the rockfall takes place requires the following long argument:

```
plot(x = t,
     y = rockfall,
     type = "l",
     xlim = as.POSIXct(c("2015-04-06 13:22:40 UTC",
                         "2015-04-06 13:23:20 UTC"),
                       tz = "UTC"))
```



This means, there is a significant bit of typing required for the generic plot function of R. There are other, more elegant ways. See the visualisation chapter for details.

Dates can be given in a variety of formats. Among the most common are POSIX, YYYY-MM-DD (Year-Month-Day) and JD (Julian Day). The function `time_convert()` can convert these three date formats

among each other. This is especially handy when the seismic data files are organised in directories with temporal structure such as years > Julian days > Hours.

Seismic data processing

Deconvolution

Deconvolution or removing the instrument response is one of the most fundamental steps in seismic data processing. It can also be a demanding step, especially when different types of sensors and loggers are used in a seismic network or array. The raw seismic signal, as it is recorded by the data loggers does not represent the ground motion in real units, yet. There are frequency and phase shifts due to the sensor characteristics and the way a data logger digitises the analogue signal. These effects need to be removed from the signal in order to use it for further quantifications and interpretations.

The function `signal_deconvolve()` does this job. It requires the raw seismic signal `data`, the sampling rate `dt`, information about the characteristics of the `sensor` and `logger`, as well as optionally the fraction of the signal `p` that will be tapered to remove boundary effects and specifying an arbitrary low water level `waterlevel` to avoid dividing by zero.

The tricky part is pointing at the characteristics of the sensor and logger. The package `eseis` contains a list of commonly used seismic sensors and loggers, which can be accessed by the functions `list_sensor()` and `list_logger()`. If your sensor and logger is in the collection: congratulations, you just need to provide the keyword of the sensor and logger to `signal_deconvolve()` and things should work. If not, see below. So in the easiest case, assuming you work with a Trillium Compact TC120s broadband seismometer and an Omnirecs Cube3ext with breakout box and you want to deconvolve the rockfall signal from above, which is recorded with 200 Hz, only the following line of code is needed:

```
s <- signal_deconvolve(data = rockfall, dt = 1/200)

range(rockfall)
```

```
## [1] 61480 68993
```

```
range(s)
```

```
## [1] -1.284166e-05  1.176809e-05
```

This is possible because the author of this package mainly uses this combination of sensor and logger and thus deliberately set these keywords as default. However, it is only necessary to specify different sensors and loggers, which does not require that much more code actually. Note that the range of the seismic signal changed and that the deconvolved units are in m/s.

If the sensor or logger is not in the list of documented instruments, it is possible to provide the information manually. For this, let us first take an exemplary look at the entry of of one seismic sensor.

```
str(list_sensor()[[1]])
```

```
## List of 10
##  $ ID          : chr "TC120s"
##  $ name        : chr "Trillium Compact 120s"
##  $ manufacturer: chr "Nanometrics"
##  $ type        : chr "broadband seismometer"
##  $ n_components: num 3
##  $ comment     : chr "Data taken from data base of Arnaud Burtin"
##  $ poles       : cplx [1:11] 0.0369+0.037i 0.0369-0.037i -343+0i ...
##  $ zeros       : cplx [1:6] 0+0i 0+0i -392+0i ...
##  $ s           : num 749
```

```
##  $ k           : num 4.34e+17
```

We will not go through all the details. The essential parts are the elements `$poles`, `$zeros`, `$s` and `$k`. These are used in the deconvolution procedure. Hence, to define the characteristics of a new sensor (same for loggers, though there it is only the `$AD` element that needs to be changed) we first need to extract any given sensor entry from the list of supported instruments:

```
sensor_new <- list_sensor()[[1]]
```

Then, the respective elements can be changed:

```
sensor_new$ID <- "sensor_manual"
sensor_new$poles <- c(-8.7965+8.9742i,
                      -8.7965-8.9742i)
sensor_new$zeros <- c(0+0i,
                      0+0i)
sensor_new$s <- 1920
sensor_new$k <- 4.1037
```

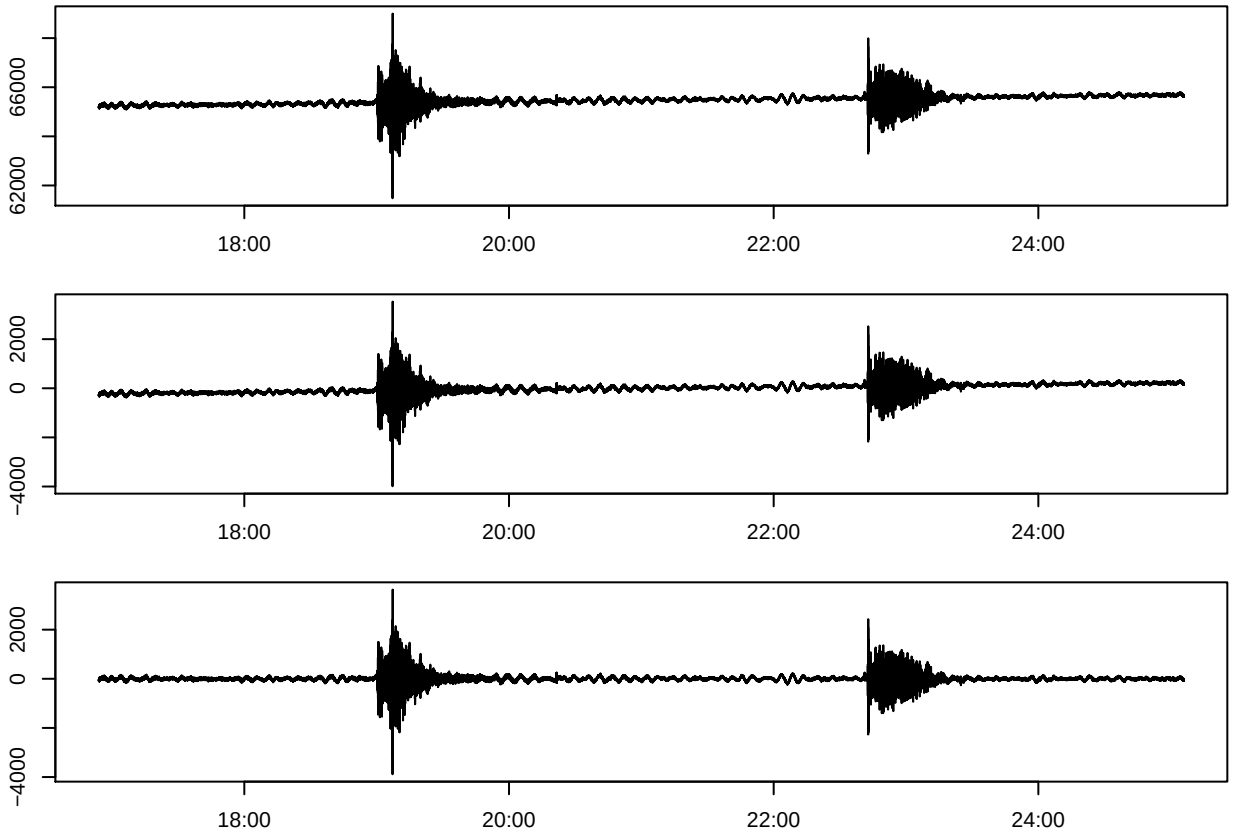
And the new sensor can be provided to the deconvolution function:

```
s_dummy <- signal_deconvolve(data = rockfall,
                             dt = 1/200,
                             sensor = sensor_new)
```

Removing mean and trend

Sometimes seismic signals show a very long term trend that would cause artefacts in applied signal processing functions. This trend can be removed. In the simplest case, subtracting the mean of the entire signal vector will center the signal. Additionally, a linear trend can/should be removed, as well. There are two functions for these jobs: `signal_demean()` and `signal_detrend()`, which are pretty straightforward to use:

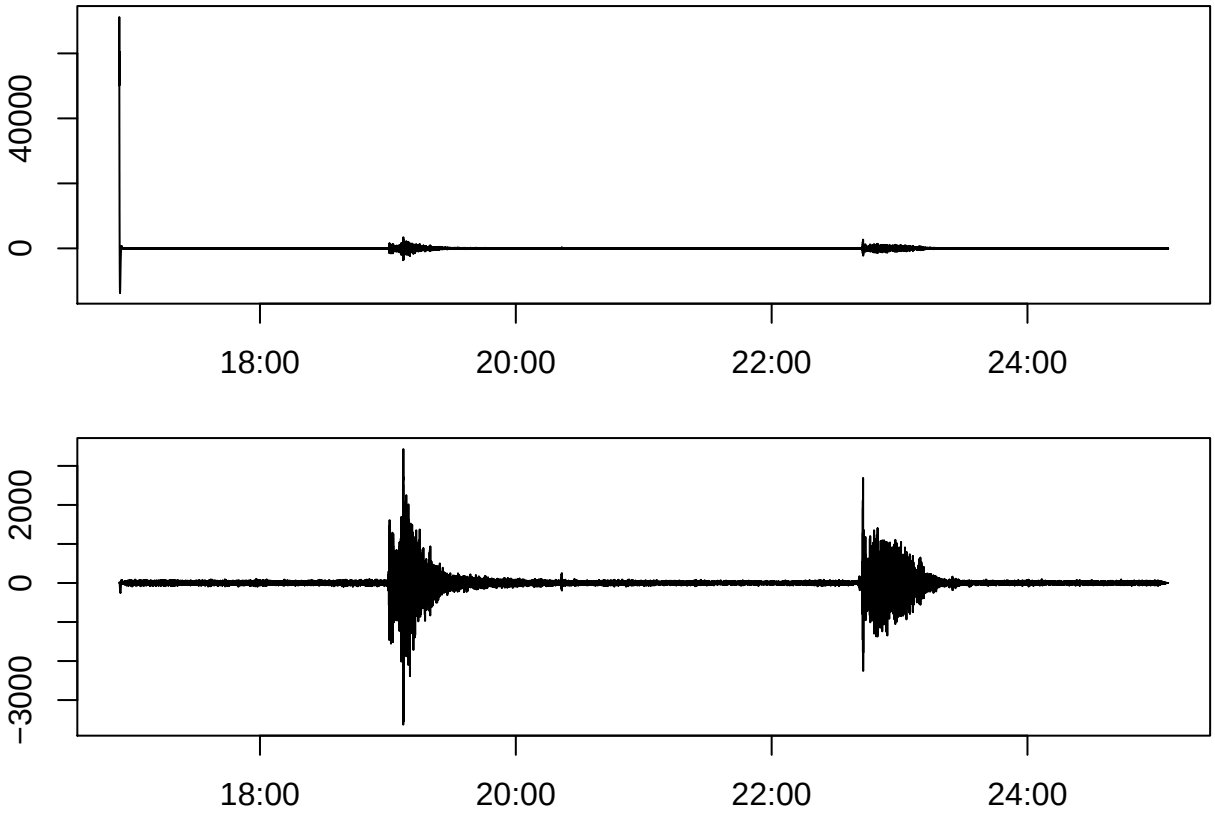
```
s_demean <- signal_demean(data = rockfall)

s_detrend <- signal_detrend(data = rockfall)

## adjust plot setup, three rows, smaller margins
par(mfcol = c(3, 1), mar = c(2.5, 2.5, 1, 0.5))

## plot all three signals
plot(x = t, y = rockfall, type = "l")
plot(x = t, y = s_demean, type = "l")
plot(x = t, y = s_detrend, type = "l")
```



Apply a taper

Tapering signals is required to account for boundary or edge effects. A classic edge effect is the boost of signal amplitudes at the beginning and end of a signal vector after applying a bandpass filter. To suppress these effects, the signal can be multiplied by a cosine taper, which is almost 1 for most of the signal but falls off rapidly at the edges of the vector. The proportion of the signal that is tapered can be specified in the function `signal_taper()`, either by the argument `p` or the argument `n`. While `p` denotes the proportion, e.g., `10^-6`, `n` denotes the number of samples that are affected by the taper.

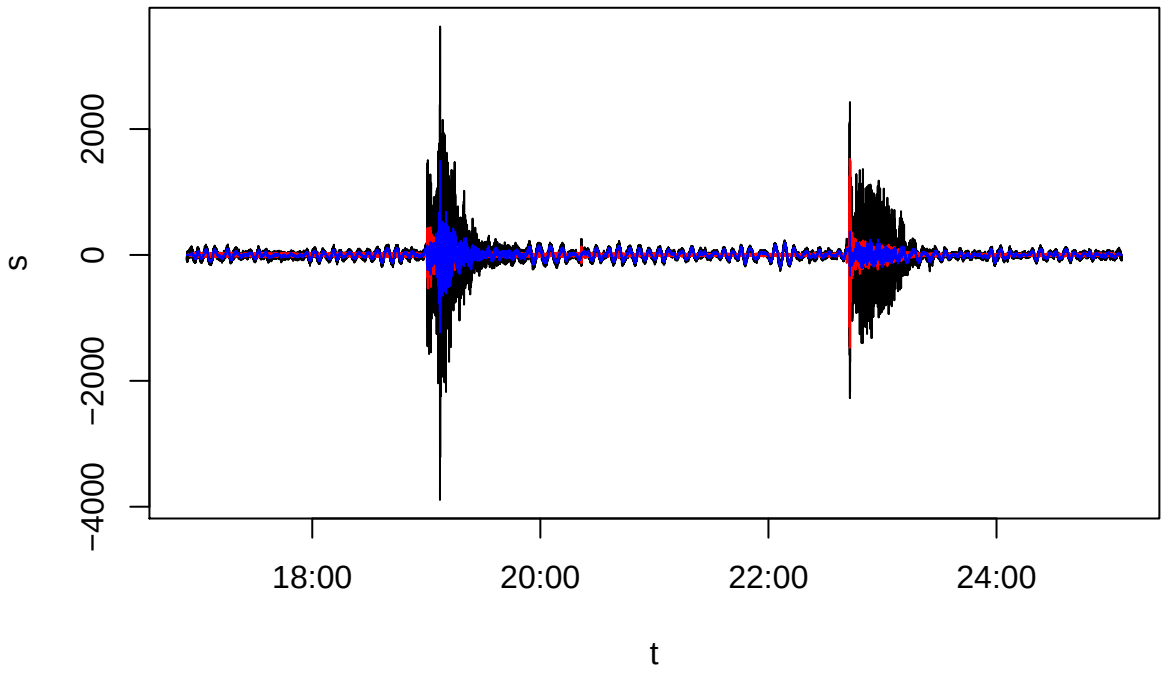
```
## create artefact by filtering the signal
s <- signal_filter(data = rockfall, dt = 1/200, f = c(1, 90))

## taper the signal
s_taper <- signal_taper(data = s, p = 10^-2)

## adjust plot setup, three rows, smaller margins
par(mfcol = c(2, 1), mar = c(2.5, 2.5, 1, 0.5))

## plot all three signals
plot(x = t, y = s, type = "l")
plot(x = t, y = s_taper, type = "l")
```



Filtering

Using filters to isolate or exclude the frequency content of interest in a signal is one of the most common preparation steps in any kind of signal processing analsis. Currently `eseis` supports only butterworth filter shapes which can be used as lowpass (`type = "LP"`), highpass (`type = "HP"`), bandpass (`type = "BP"`) or bandreject (`type = "BR"`) filters in the function `signal_filter()`. The function also requires providing the signal vector to filter (`data`), the sampling period (`dt`) and the corner frequency (or frequencies in the case of bandpass and bandreject filtering), i.e., the frequencies at which to cut off the signal's frequency content. Optionally, the order of the filter and the proportion of the posteriour taper step (cf. Apply a taper) can be provided.

```
## remove trend from signal
s <- signal_detrend(data = rockfall)

## apply highpass filter
s_HP <- signal_filter(data = s, dt = 1/200, f = 40, type = "HP", p = 10^-2)

## apply highpass filter
s_LP <- signal_filter(data = s, dt = 1/200, f = 5, type = "LP", p = 10^-2)

## plot all signals
plot(x = t, y = s, type = "l")
lines(x = t, y = s_HP, col = 2)
lines(x = t, y = s_LP, col = 4)
```



Note the difference between earthquake (first event) and rockfall (second event): The earthquake, especially the arrival of the S-wave part, is dominated by the low frequency content, whereas the rockfall shows a burst of seismic energy during the first impact of rocks that affects the high frequency realm.

Rotating signals

Rotation of seismic signals is a common task from the field of classic seismology and can be used, for example to decipher different wave types or to estimate the source direction of an event. However, in environmental seismology, the wave content of the emitted signals is much more heterogeneous and mixed, which makes application of these classic approaches sometimes difficult. Rotation can be performed with the two horizontal components of three-component signals.

To rotate the signal of an event, all three components must be present. We did alread handle such a case (cf. Reading seismic data) and have imported the three components of the example events into the object `rockfall_xyz`. This data set is a list object. The function `signal_rotate()` requires however a matrix object with signal traces organised in rows. This is for implemented for clarity, i.e., that the user is aware of the special case of using three components of the same recorded event, and also to secure that all traces have the same length. To convert a list to a matrix, there is an elegant but cumbersome function in R:

```
xyz <- do.call(what = rbind, args = rockfall_xyz)
```

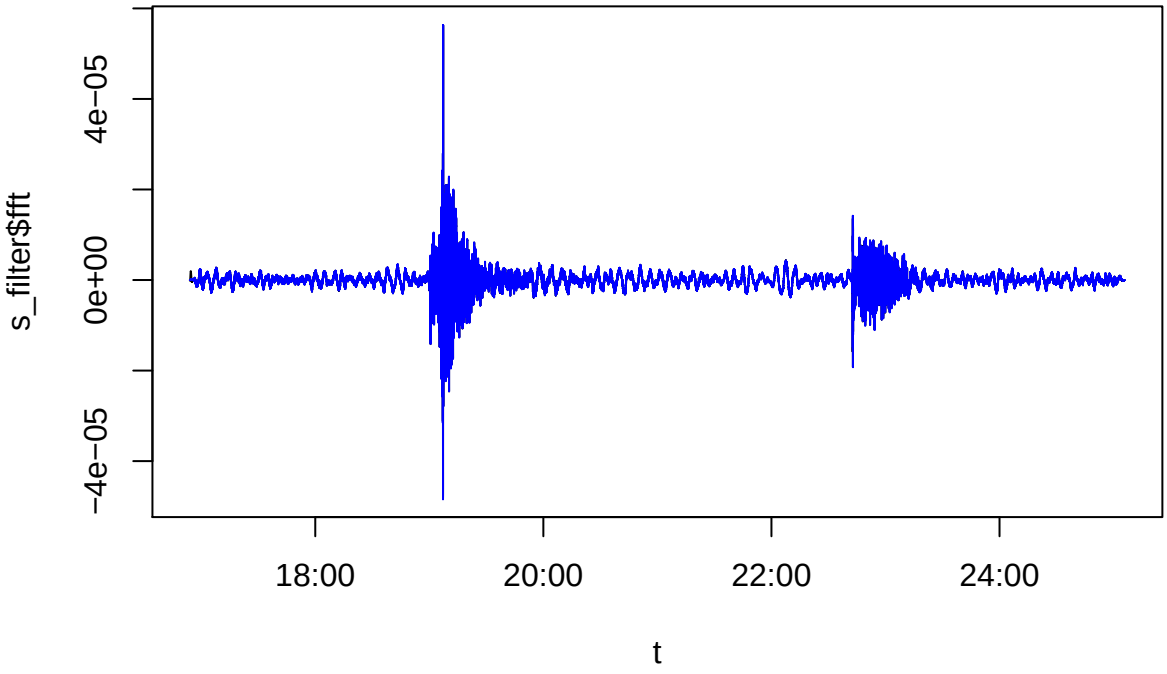
If you want to read more about this, I suggest you take a look at my R cook book. Now we can use this matrix with the function `signal_rotate`, providing any arbitrary rotation angle in degrees:

```
xyz_rotate <- signal_rotate(data = xyz, angle = 90)
```

Integrating signals (convert to displacement)

Broadband seismometers usually measure ground movement in terms of velocity, i.e., m/s. To work with displacement the signal needs to be integrated according to $x = v \cdot t$. There are two ways to do this. The correct way would be doing this in the frequency domain, the default option. But signals can also be integrated using the trapezoidal rule. Both approaches are supported by the function `signal_integrate()`:

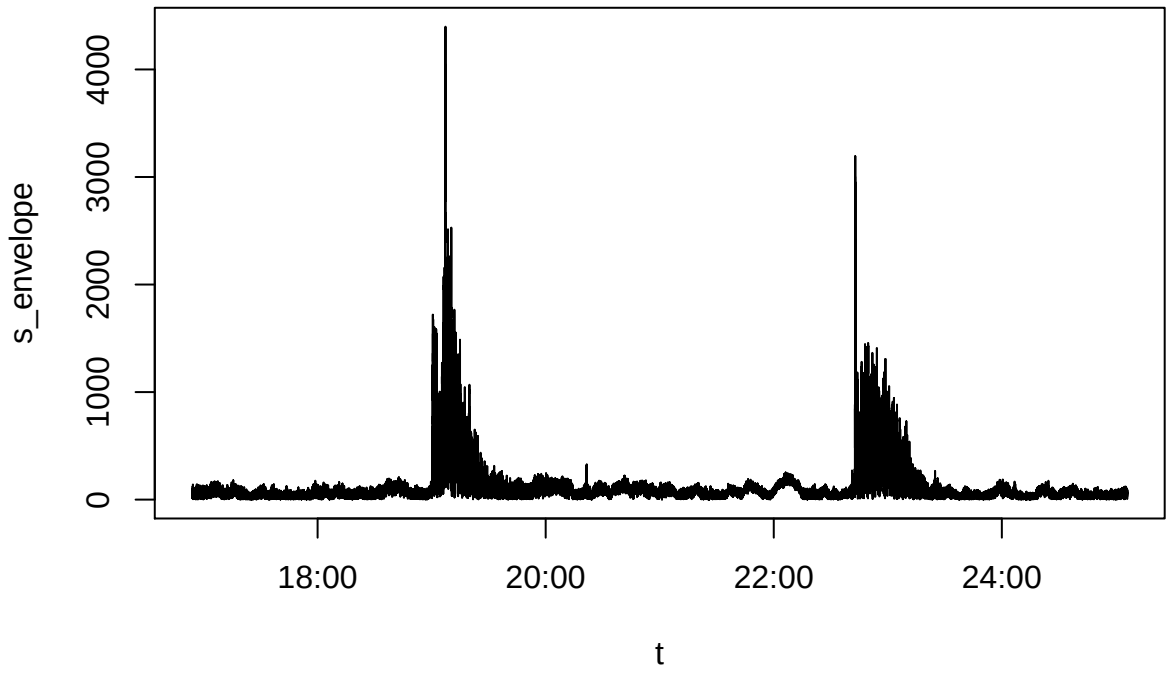
```
## deconvolve signal
s <- signal_deconvolve(data = rockfall, dt = 1/200)

## integrate in the frequency domain
s_integrate_1 <- signal_integrate(data = s,
                                  dt = 1/200)

## integrate in the time domain
s_integrate_2 <- signal_integrate(data = s,
                                  dt = 1/200,
                                  method = "trapezoid")

## filter the signal to remove unwanted content
s_filter <- signal_filter(data = list(fft = s_integrate_1,
                                      trapez = s_integrate_2),
                          dt = 1/200,
                          f = c(1, 90), p = 10^-2)

## plot both results
plot(x = t, y = s_filter$fft, type = "l")
lines(x = t, y = s_filter$trapez, col = 4)
```



Signal envelopes

Signal envelopes are the used to represent the energy carried by a seismic signal. The function `signal_envelope()` also allows tapering the function output. To get the envelope fo the rockfall signal use:

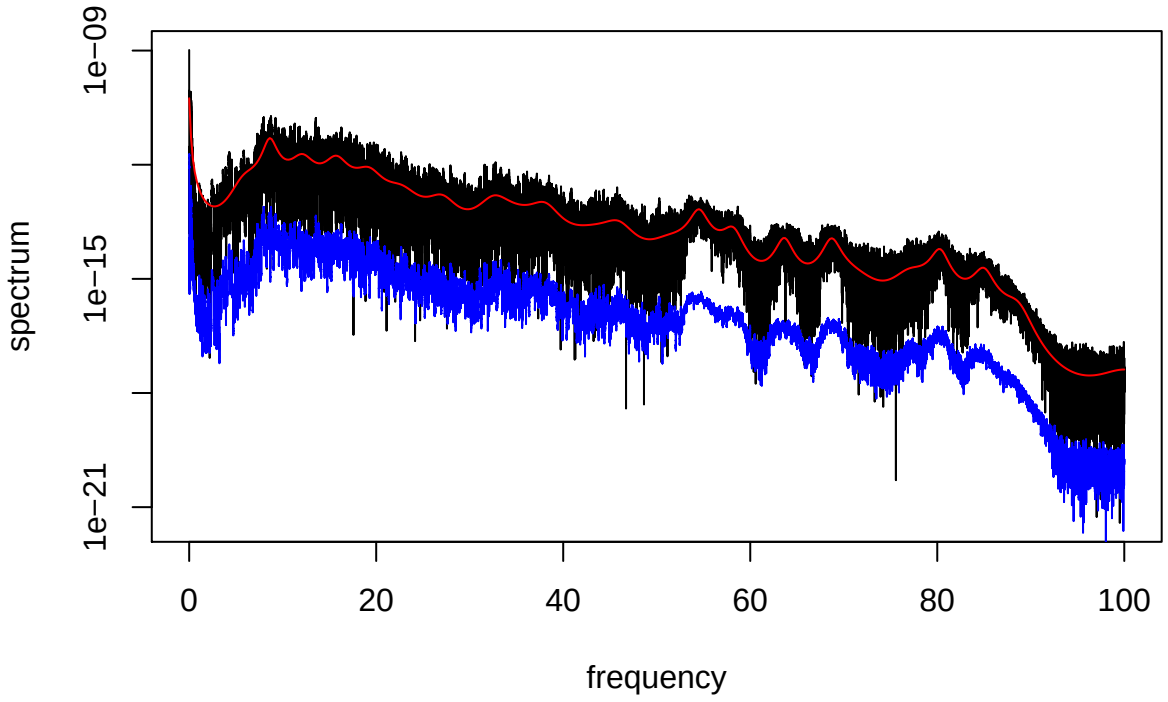
```
s_envelope <- signal_envelope(data = s_detrend)

plot(x = t, y = s_envelope, type = "l")
```



Creating a spectrum

To inspect the frequency content of a seismic signal (or rather a part of it) it is useful to calculate and plot its spectrum, which is its fast Fourirer transform. There are different ways to do this. The function `signal_spectrum()` supports the following: `"periodogram"` (the default), `"autoregressive"` and `"mutlitaper"`. Autoregressive specrae appear rather smoothed and may not always be appropriate. The `"multitaper"` option should be used for rather short signal vectors because spectrae will then be corrected for edge effects by applying many tapers. This comes at the cost of computational time, so using the `"multitaper"` option for minute or even hour long signals will take ages.

```
## calculate the spectrae
s_periodogram <- signal_spectrum(data = s,
                                 dt = 1/200)

s_autoregressive <- signal_spectrum(data = s,
                                    dt = 1/200,
                                    method = "autoregressive")

s_multitaper <- signal_spectrum(data = s,
                                dt = 1/200,
                                method = "multitaper")

## plot the results
plot(s_periodogram, type = "l", log = "y")
lines(s_autoregressive, col = 2)
lines(s_multitaper, col = 4)
```

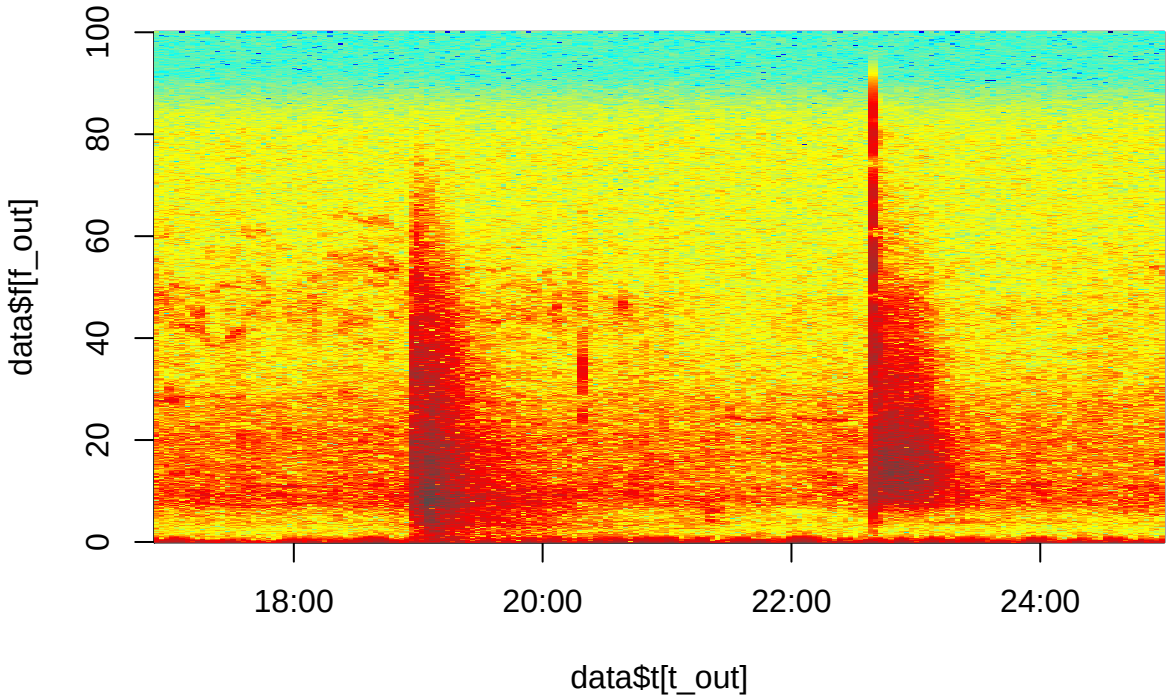


Creating a spectrogram

A step beyond calculating the spectral characteristics of the entire signal is to calculate how spectral properties change over time. This is called a power spectral density estimate (PSD), spectrogram or time-frequency-plot. The best visualisation technique is an image that plots time along the x-axis, frequency along the y-axis and lets the colour scale depict the spectral power as a function of time and frequency.

In the easiest way, the seismic time series is cut into smaller time slices and for each slice a spectrum is calculated and appended to a matrix. The time slices can also overlap, which gives a more gradual change of properties. The size of each time slice (`window`) is defined in seconds, the overlap as a number between 0 and 1 (`overlap`).

```
PSD <- signal_spectrogram(data = s, time = t, dt = 1/200, plot = TRUE)
```

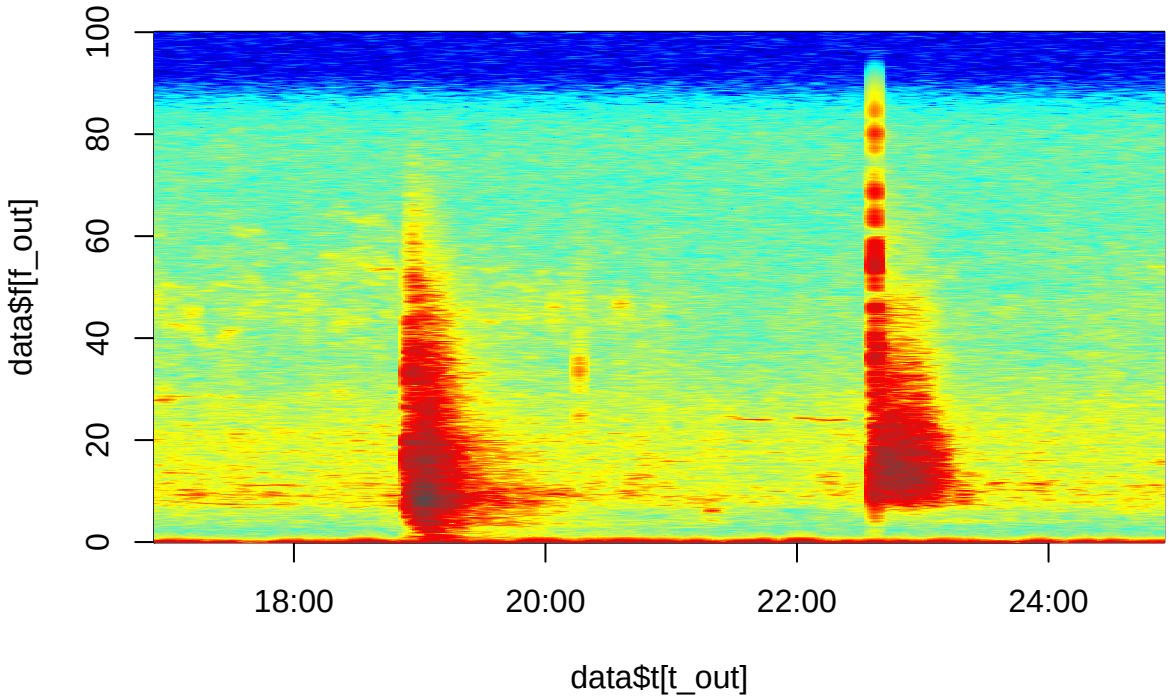


The function `signal_spectrogram()` will automatically set values if the window length is not specified (default is 1 % of the signal length). It is however useful to adjust this value to get meaningful results in some cases. The function output is a list with three elements, the vector of start time values for each time slice `$t`, the frequency vector `$f` and the PSD matrix object `$S`. Setting the argument `plot = TRUE` calls the function `plot_spectrogram()` without any further options and creates an image plot of the calculated PSD, as in the figure above. For more elaborated comments on plotting PSD see chapter Visualisation.

A PSD as calculated above may be only a poor approach when the time series is short (i.e., shorter than hours or days) or when temporal resolution must be high. There is always a tradeoff between high temporal resolution (small window size) and high frequency resolution (large window size). Also, simply appending frequency spectrae results in crisp, coarse-looking plots.

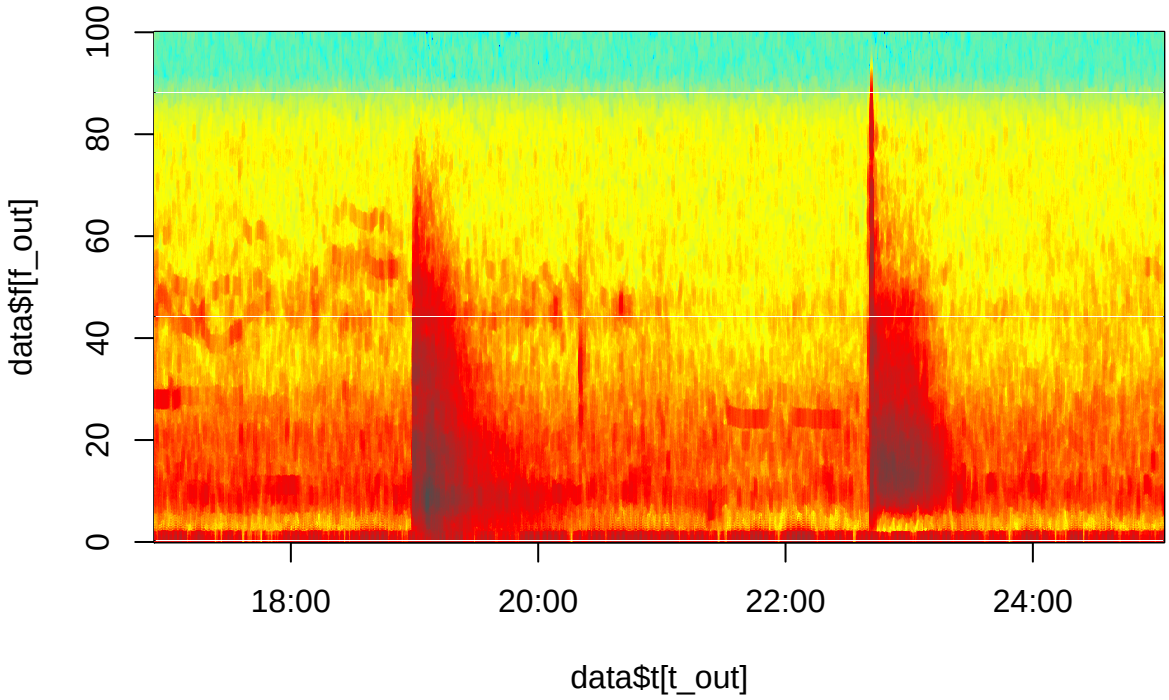
Welch (GIVE YEAR) introduced a workaround for the latter issue (setting `Welch = TRUE`). By cutting each time window in even smaler time windows (`window_sub`) with some overlap (`overlap_sub`) and averaging the set of resulting spectrae, one gets an improved verion of a PSD. Of course, this comes at the cost of some computational time because the computer will have to calculate a spectrum much more times, depending on the number of sub-windows in each window and the overall number of windows. To run `signal_spectrogram()` using the Welch-option use the following code:

```
PSD <- signal_spectrogram(data = s,
                          time = t,
                          dt = 1/200,
                          Welch = TRUE,
                          window = 11,
                          overlap = 0.9,
                          window_sub = 6,
                          overlap_sub = 0.9,
                          plot = TRUE)
```



For very short time series, even the Welch-option reaches a limit. This is when each spectrum should be calculated using the multitaper approach (see above). This comes again at the cost of computation time depending on the number of tapers (`k`, default is 7). To run the function using the mutlitaper option set `multitaper = TRUE`:

```
PSD <- signal_spectrogram(data = s,
                          time = t,
                          dt = 1/200,
                          window = 2,
                          overlap = 0.8,
                          multitaper = TRUE,
                          plot = TRUE)
```



It is also possible to combine the Welch and multitaper options and get a really "nice-looking result" but this will raise high computational demands, depending on the length of the signal to process. You should not do this for signals longer than several seconds and certainly not for signals that last hours.

Picking events

Seismic data is extensive: a day of monitored Earth surface activity with one three-component station yields more than 300 million individual samples. Finding activity events from this storm of data is nothing you want to do manually. There is a wide range of techniques that are used to do this semi-automatically or automatically. Most of them are very deterministic and require setting fix thresholds a signal needs to pass in order to be handled as an event. One of the most widely used "picker" or "trigger" approaches is the short-term-average-long-term-average-ratio (STA/LTA) picker.

The STA/LTA-picker is suited for events with rather instantaneous onsets and short durations. It is important to run the picker on the envelopes of filtered signals, not the pure signals. Such events will have almost no effect on a long-term signal average (long means, many seconds or some minutes) but a significant effect on a short-term signal average (short means a few or even less than one second). Thus, the onset of an instantaneous event will yield a high STA/LTA value. If the event is over after a few seconds at best, but certainly well below the averaging range of the long-term average window the ratio will return to the mean noise value. Thus, an STA/LTA-picker can be used to define the start and end time of an event.

If the event is longer than the LTA window size it will raise the latter value and thus decrease the STA/LTA value, which will result in lower than desired values and consequently an underestimation of the real event duration. The problem is that many Earth surface processes last longer than a few seconds. Imagine for example a river bedload transport event, a debris flow, a large rock avalanche or a strong rain event. To account for this, it is useful to freeze the LTA value at the onset of the event.

Now, how is this handled by the R-package `eseis`? Currently, only the STA/LTA picker method is implemented, simply because it does such a good job for most of the cases, encoutered so far. It can be used in the classic way or with the option to freeze the LTA value upon the onset of an event. Since this requires sample-by-sample evaluation it is not suited for the usual R-environment and has been written in C++ which reduces screening a day of data for one station from more than 5 minutes to less than 15 seconds (Thanks,

Sebastian at this point!).

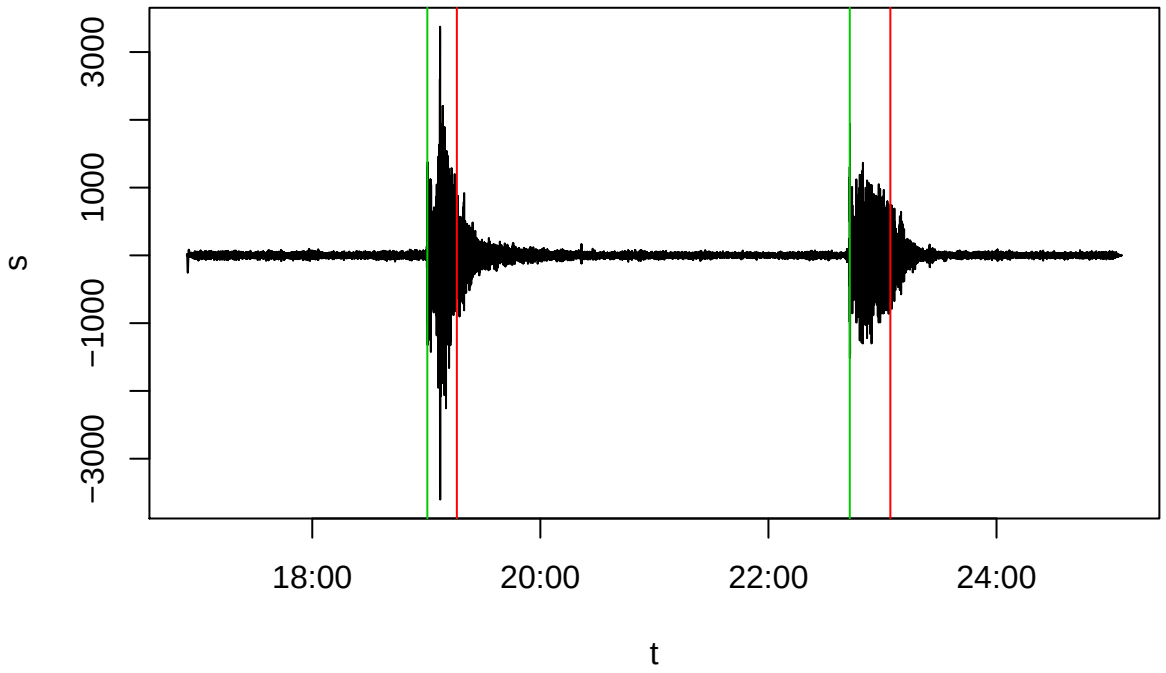
Let us come to the action. We take again the example sequence used a couple of times before: an earthquake, followed by a rockfall. Two clear events that should be detected by any kind of meaningful picker. Note that the function `signal_stalta()` should be run with the envelope of a seismic signal, filtered to the window of interest, e.g., for rockfalls 10 to 40 Hz. So we must first filter the signal and then calculate its envelope. The essential arguments that need to be provided are the number of samples for the short-term (`sta`) and long-term (`lta`) window and the onset (`on`) and end (`off`) thresholds. Finding good estimates or guesses of is a broad field, peaking into already published experiences might help. In the following example the values are set to correspondants of 0.5 and 90 seconds and thresholds of 5 and 2.

```
## filter the signal to a useful frequency window
s <- signal_filter(data = rockfall,
                   dt = 1/200,
                   f = c(1, 30),
                     p = 10^-2)

## calculate the signal envelope
e <- signal_envelope(data = s)

## pick events
events <- signal_stalta(data = e,
                        time = t,
                        dt = 1/200,
                        sta = 100,
                        lta = 18000,
                        on = 5,
                        off = 2)

print(events)
```

```
##   ID               start duration
## 1  1 2015-04-06 13:19:00    15.53
## 2  2 2015-04-06 13:22:42    21.36
```

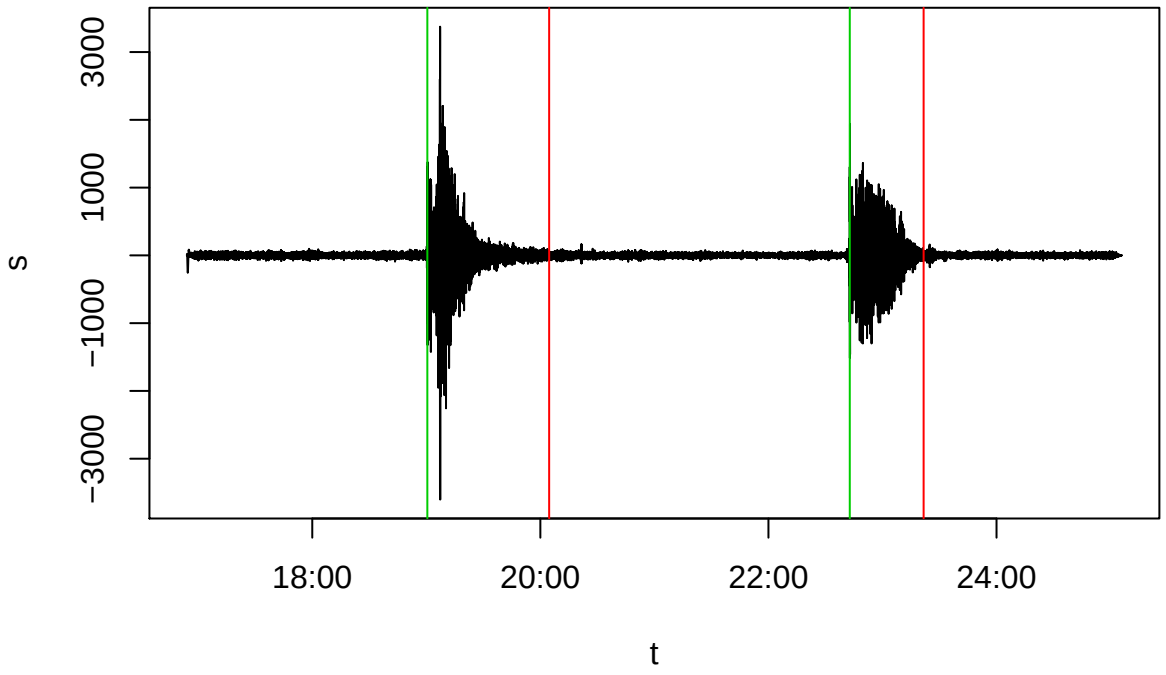
This returns us two events. The first starts at 13:19:00 and lasts about 15 seconds (the earthquake), the second starts 3.7 minutes later and lasts more that 20 seconds (the rockfall). We can actually plot the output of `signal_stalta()` onto the signal vector:

```
plot(x = t, y = s, type = "l")
abline(v = events$start, col = 3)
abline(v = events$start + events$duration, col = 2)
```



An off-value of 2 appears to be a fat underestimate of the duration of both events, actually. Feel free to optimise the threshold value. Or additionally use the freeze-option:

```
## pick events
events <- signal_stalta(data = e,
                        time = t,
                        dt = 1/200,
                        sta = 100,
                        lta = 18000,
                        on = 5,
                        off = 1,
                        freeze = TRUE)

plot(x = t, y = s, type = "l")
abline(v = events$start, col = 3)
abline(v = events$start + events$duration, col = 2)
```



Further processing functions

Further, rather auxiliary functions that are frequently used in the processing chain include reducing the sampling resolution by a given integer factor (downsampling or aggregation of the signal: `signal_aggregate()`), calculating the vector sum of a list of signals – mainly to get the total ground displacement or velocity of all three seismic components (`signal_sum()`), calculating the signal-to-noise ratio of a signal (`signal_snr()`), which is defined as the ratio of the maximum and the mean value of the submitted signal, padding a signal with zeros until it reaches a length corresponding to the next higher level of `2^n` (`signal_pad()`), a requirement for many fft-operations, and calculating the Hilbert transform of a signal (`signal_hilbert()`).

Spatial data processing

One of the central goals of environmental seismology is, after having identified a seismic source, estimating the location where the process has acted to generate the seismic signal. In contrast to classic seismology, Earth surface processes usually generate signals with an emerging onset that eventually fade into background noise. This makes utilisation of established localisation approaches, based on picking first arrival times, challenging.

There are other, more appropriate methods available. One of them is based on cross-correlating the entire signals of an event among different stations and model the location in space that yields the best overall correlation imposing time lags corresponding to the estimated seismic wave velocities. For more information and justification of this method called signal migration see Burtin et al. (2012, 2016) or Hilbert et al. (2014).

In principal, to migrate a signal and thus find the most realistic loaction estimate, one needs to build a lookup table of the distance between a seismic station and every pixel in a grid corresponding to the DEM used to represent the topography of the area of interest, a so called distance map. Such a distance map needs to be built for each seismic station. Furthermore, the distance cannot be calculated simply using the euclidian distance because seismic waves travel in 3D, either along the direct path between surface source and seismic station when the path is in bedrock (or sediment or soil or regolith, doesn't matter as long as it is a solid) or along the surface if the direct path would be through air. This is essential, though it is only of marginal relevance in flat terrain. However, in steep alpine terrain, this point does matter. Based on the distance maps and a given seismic wave velocity, all seismic signals are shifted in time pixel by pixel according to the

respective distances and then cross-correlated. Hence, depending on the size of your area of interest, the DEM (or distance map) resolution, length of the event and the number of stations, the computational time can become exhaustive. However, the code used in the package can handle an event of 30 seconds, measured at 200 Hz by six seismic stations within an area of interest depicted by 100000 pixels in a few seconds.

Apart from the distance maps one also needs a matrix of inter-station distances, also respecting the travel path of seismic waves within bedrock or along the surface. The computation of both objects is similar and will be handled by the same function. The last ingredient of this recipe is an estimate of the most likely average seismic wave velocity, which can be found by changing wave velocities and test which value gives the best overall loacation quality in terms of coefficient of determination. A better approach would be to use an active source with a known location and measure the time offset of the signal arrival at all stations.

Convert coordinates

A precondition of most of the following content is a common geographic coordinate system. If data from different sources (DEM, recorded GPS data, GPS-based station locations, independently mapped process locations and so on) is combined, mostly the coordinates will need to be homogenised to a common format. This can be achieved with the function `spatial_convert()`. It converts a set of input coordinates (organised as matrix or data frame) from the specified input reference system to the target reference system. This step makes use of the strong support of spatial data in R and relies on the established nomenclature of proj4-strings from the proj4-library. For more information and examples see the Spatial Reference website that has a more or less convenient search option and lets you choos between different represenations of the projection of your coice. You should select the Proj4 link and copy paste the line of text.

To convert, say decimal degree to UTM coordinates you can use the following code:

```
## define some arbitrary DD coordinates
xy <- cbind(13, 55)

## define input projection string
proj_in <- "+proj=longlat + ellps=WGS84"

## define output projection string
proj_out <- "+proj=utm +zone=32 +datum=WGS84"

## convert the coordinates
spatial_convert(data = xy,
                from = proj_in,
                to = proj_out)
```

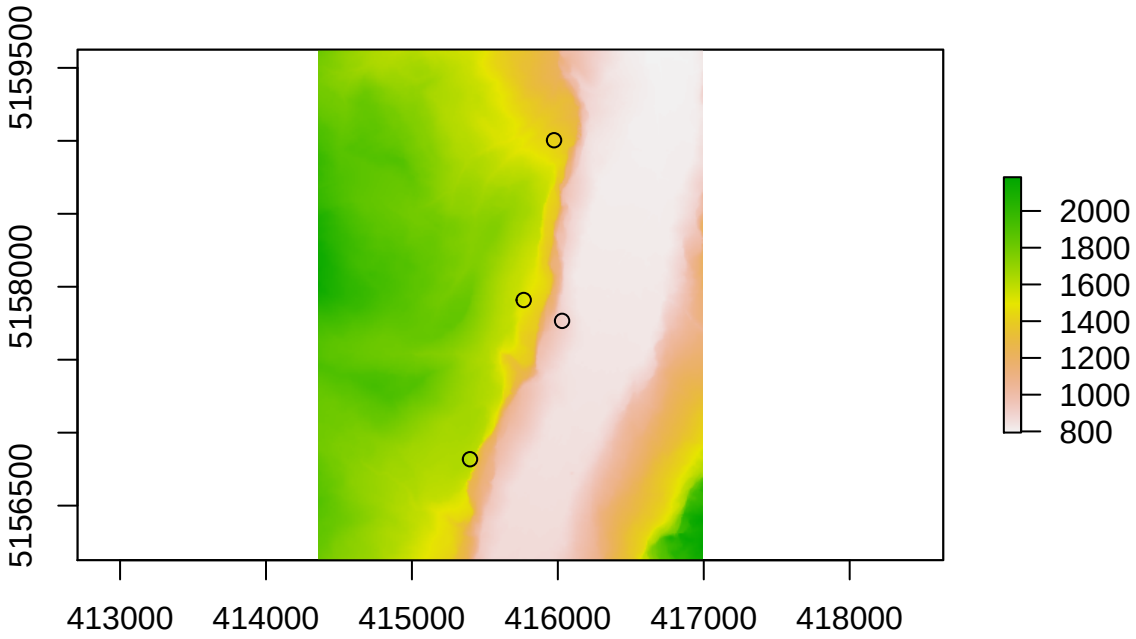
```
##   coords.x1 coords.x2
## 1    755803   6102111
```

Create distance data

The function `spatial_distance()` helps creating both, distance maps (`dmap = TRUE`) and interstation distance data (`dstation = TRUE`). It requires a matrix with the coordinates of the seismic stations and the DEM of the area of interest. The DEM must be free of NA-values (see my R cook book for ways to read raster data, handle NA-values, cropping and aggregating DEM data). The function can also be used without topography corresction (i.e., respecting that seismic waves travel directly in bedrock or along the surface, `topography = FALSE`) and it can be run on more than one CPU core, which is useful to save time for highly resolved and/or large areas of interest. To create a distance map of, for example a part of the Lauterbrunnen Valley in Switzerland we will use the following spatial data:

```
## read DEM
dem <- raster(x = "~/data/dem_10m.img")

## read station coordinates
stations <- read.table(file = "~/data/stations.txt")

## plot basic map
raster::plot(dem)
points(points(x = stations[,1], y = stations[,2]))
```

```
## Loading required package: sp
```



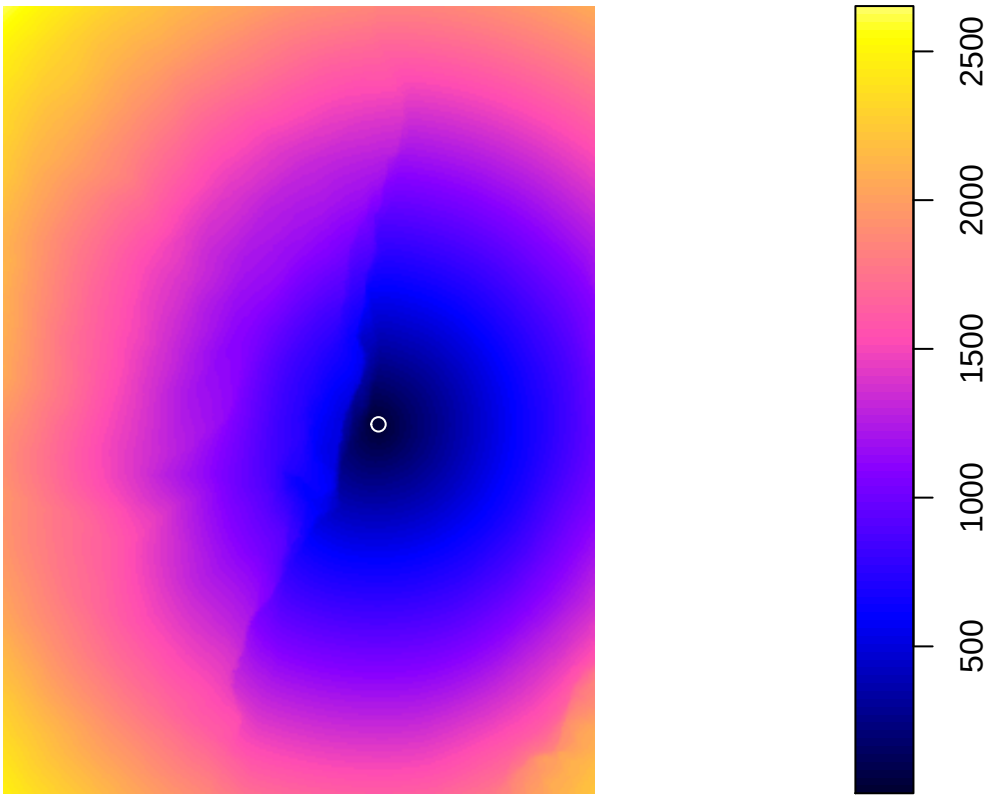
The Lauterbrunnen Valley is a steep limestone cliff with 600 m high, nearly vertical walls. Three seismic stations are located on top, one at the base of this impressive cliff. To create distance maps and interstation distances for this setting the following code is used (computation time for this 10 m DEM on one CPU more than one hour!):

```
D <- spatial_distance(stations = stations, dem = dem)
```

```
## [1] "Processing station distances"
```

A simple plot of the distance matrix for the station at the base of the cliff shows the effect of respecting topography for such impressive limestone cliffs. On flat terrain, the distance colour scale would show almost concentric patterns. However, here the long way up along the cliff face deforms all distance values west of the station:

```
raster::plot(x = D$maps[[4]])
points(x = stations[4,1], y = stations[4,2], col = "white")
```



The interstation distances show a similar pattern, stations above versus below the cliff face are several hundred metres apart from each other, much more than the plan view of the DEM map above would suggest:

```
print(D$stations)
```

```
##          1        2        3        5
## 1    0.000 1395.154 2921.721 1816.739
## 2 1395.154    0.000 1479.050  785.414
## 3 2921.721 1479.050    0.000 2800.795
## 5 1816.739  785.414 2800.795    0.000
```

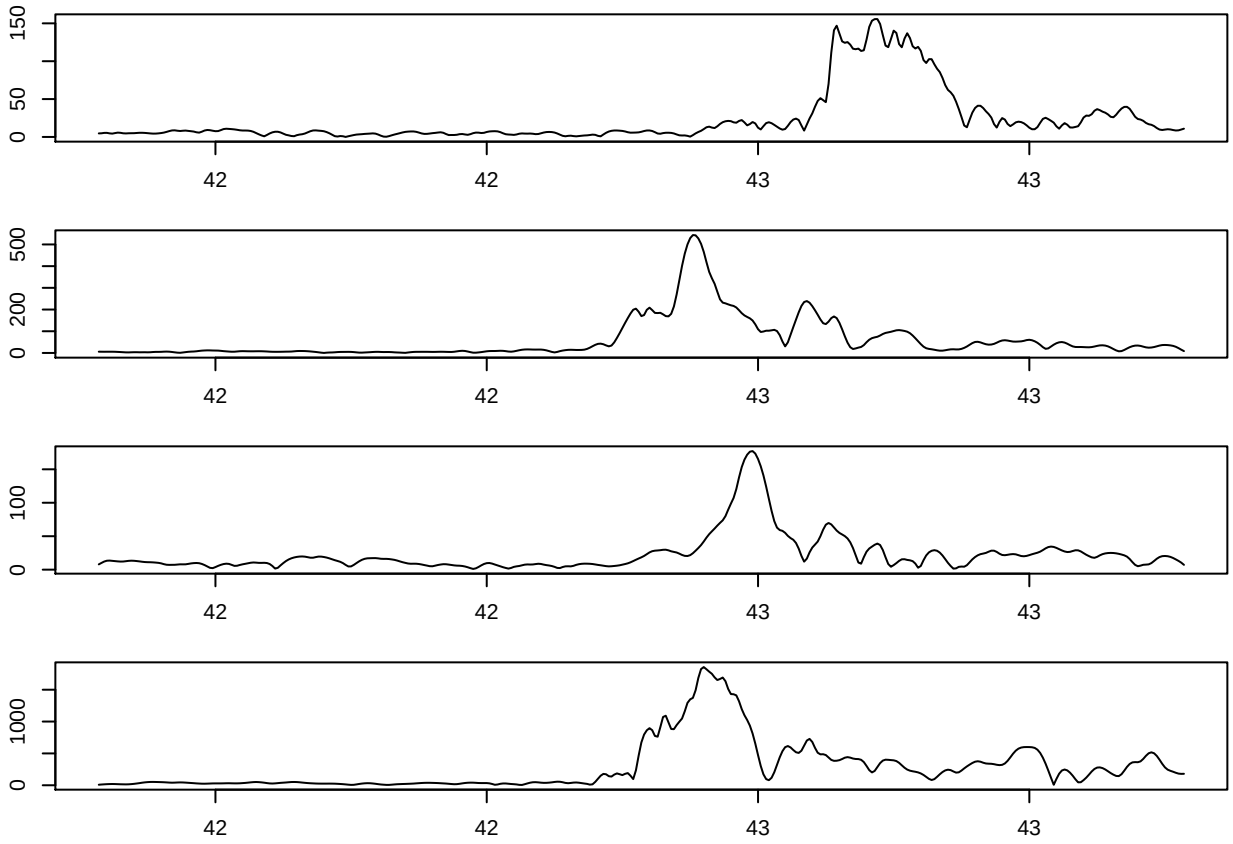
Migrating seismic signals

Now we have all things in hand to migrate a seismic signal within our area of interest. The example signal used so many times before actually comes from the Lauterbrunnen Valley (not really a surprise, right?). First we need to clip the signal to the actual event, for example using the output of the picking exercise. It is however only the first strong impact we want to localise. This is why we do not clip after the entire event duration but clip it to one second before and after the event onset.

But there is one more thing: we need the representation of the event not only by one but by all stations. Thus, the files of all stations that contain the signal need to be imported, filtered to the frequency window of interest, clipped and their envelopes must be calculated. This is actually the really extensive part if the work, writing R-code to import the right files and do the nasty preparation work woth them (see appendix for some extra help on this). The result of this work is shown in the plot below (only four stations were at operation during the event, so only four instead of six signals are imported):

```
## magically import four seismic signals
load(file = "rockfall_location.rda")

## filter signals
```

```
rockfall_location <- signal_filter(data = rockfall_location,
                                   dt = 1/200,
                                   f = c(5, 20),
                                   p = 10^-2)

## make envelope
rockfall_location <- signal_envelope(data = rockfall_location)

## make an index of the samples within the time period of interest
i_event <- seq(1, length(t))[
  t >= events$start[2] - 1 & t <= events$start[2] + 1]

## clip all vectors to the time of interest
s_locate <- vector(mode = "list",
                   length = length(rockfall_location))

for(i in 1:length(s_locate)) {

  s_locate[[i]] <- rockfall_location[[i]][i_event]
}

## adjust plot setup, three rows, smaller margins
par(mfcol = c(4, 1), mar = c(2.5, 2.5, 1, 0.5))

## plot all three signals
for(i in 1:length(s_locate)) {

plot(x = t[i_event], y = s_locate[[i]], type = "l")
}
```

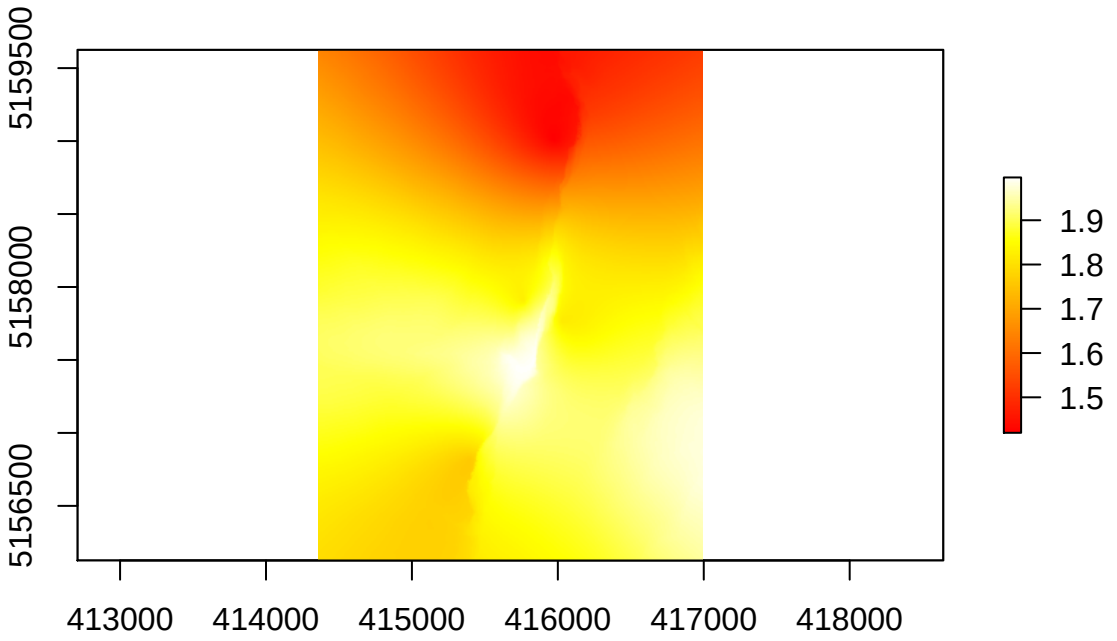


Note the different arrival times and signal amplitudes of the seismic waves at each of the four stations. Now everything is in hand to migrate signal. The hardest part is done. The function `spatial_migrate()` will just ask for everything that has already been prepared. Only the input signals need to be converted from a list to a matrix for the same reasons as dicussed above (GIVE LINK), e.g., using `do.call(rbind, s_locate)`. The seismic wave velocity is set to 2700 m/s in this case, a reasonable value in compact limestone. Note that the output of the migration is a Raster layer with summed coefficients of determination of the location probability for each pixel and can be plotted straight away:

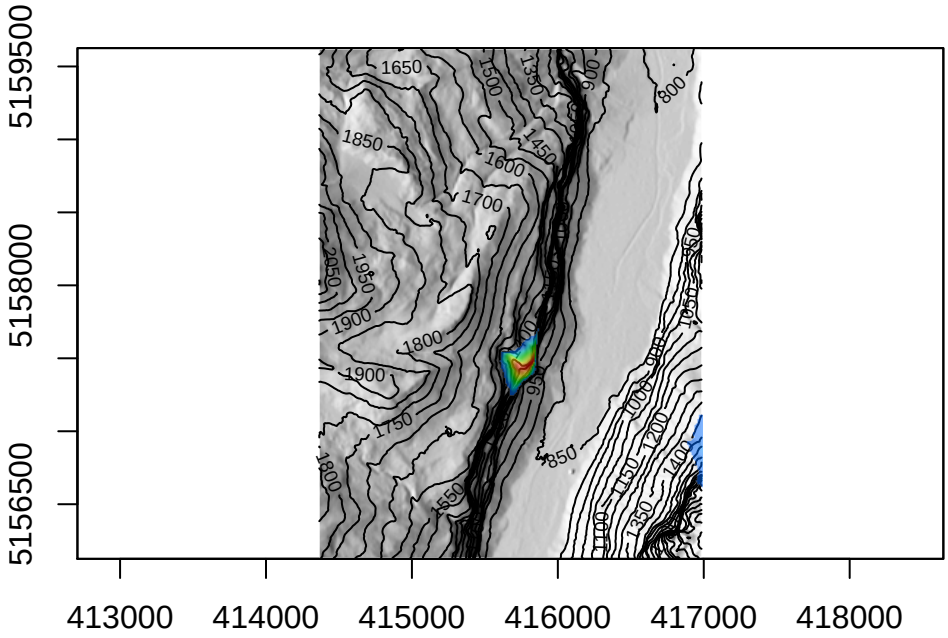
```
E <- spatial_migrate(data = do.call(rbind, s_locate),
                     d_stations = D$stations,
                     d_map = D$maps,
                     v = 2700,
                     dt = 1/200)
```

```
## [1] "No snr given. Will be calculated from signals"
```

```
plot(E, col = (heat.colors(200)))
```



The image is a direct visualisation of potential locations of the seismic source. Usually, only values above the 95 (or 99) percentile are used to illustrate the location likelyhood. See the visualisation chapter from how to illustrate this trunctation and see the plot below for an example:



There is little room for speculation. The signal waveform shows a strong burst of seismic energy (stretching over a wide frequency range when looking at the corrsponding PSD) followed by an emerging onset of prolonged activity for almost a half minute. This argues for a strong impact of a rock mass, that disintegrates and rains down as an avalanche of smaller particles until all fragments come to rest. The signal migration places the event in the middles of the cliff.