RSEIS: Seismology in R

Jonathan M. Lees
University of North Carolina, Chapel Hill
Department of Geological Sciences
CB #3315, Mitchell Hall
Chapel Hill, NC 27599-3315
email: jonathan.lees@unc.edu
ph: (919) 962-1562

July 23, 2012

1 Abstract

I present several new packages for analyzing seismic data for time series analysis and earthquake focal mechanisms. The packages consists of modules that 1) read in seismic waveform data in various common exchange formats, 2) display data as either event or continuous recordings and 3) performs numerous standard analyses applied to earthquake and volcano monitoring. **REIS** is designed as a research tool aimed at investigators who need to quickly assess large amounts of time-series as they are related to the spatial distribution of geologic structure and wave propagation. In addition to time series analysis, a spatial mapping program is included that ties waveforms and radiation patterns to geographical data-base and mapping programs.

2 Waveform Analysis

The waveform module of **REIS** reads in seismic data in SEGY, SAC, AH, UW and various ASCII formats. The core of these modules are a set of C programs that pass waveforms back to **R** and wrappers that create lists of seismograms. **REIS** was written primarily for use with continuous data, so the **R** code is able to sort a large database consisting of continuous data from several stations and numerous components. Each component of the waveform database may have a different sample rate and may require very different handling in terms of instrument de-convolution. Time-windows provided by the user are used to select off parts of the continuous data and rectify timing so that all traces represent identical time slots. Seismic data (binary or ASCII format) are read into **R** and stored in structures that provide a platform for object oriented manipulation of complex information regarding earth dynamics. In my case, I use this package to investigate exploding volcanoes in Ecuador, Guatemala, Kamchatka and Italy.

When **swig** is started an initial, interactive display of the seismic records is presented to the user and a large array of useful options are available for further processing by buttons that surround the main display but are on the same graphics device. Some of the routines employed in the **REIS** package are drawn from packages already available on the **R** distribution, for example wavelet transforms - although these have been modified to some extent to accommodate specific concerns of seismologists. Other modules, like those dealing with focal mechanisms and radiation patterns are original and will prove useful for investigators searching for patterns of stress distribution in fault regions.

3 Getting started

Start by downloading packages and installing locally in the machine being used. The packages required by **REIS** include **RPMG** , and Rwave, and RFOC if focal mechanisms are going to be inspected.

```
library(RSEIS)
library(RPMG)
library(Rwave)
```

3.1 Example: Reventador Volcano Explosion

There several data sets included in the **REIS** distribution, and these can be loaded with simple calls to data(). For example,

```
data(KH)
names(KH)
```

loads a list structure (KH) that includes wave forms and other important meta-data about the earthquake. To view this data we call the main program and display the earthquake records stored in memory,

```
##############   code
swig(KH, SHOWONLY=FALSE)
```

```
dev.off()
```

In this example we display only the vertical component of an explosion of Reventador Volcano (Figure 8). The buttons shown along the top of the screen are defaults chosen from a large selection of buttons designed to be useful for analyzing seismic data. To zoom in on the trace, click twice on the trace with the left mouse button, and then terminate by clicking the middle mouse. (Clicking the middle mouse without left mouse clicks terminates the interactive session). When finished with

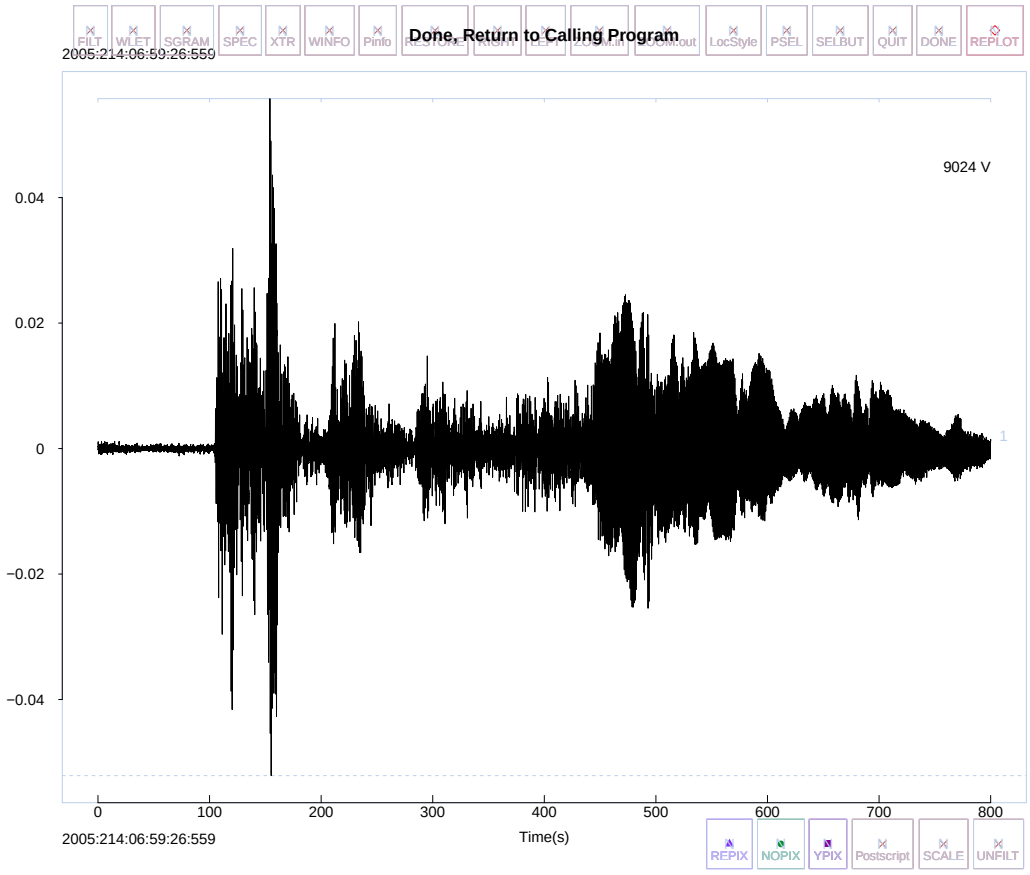

Figure 1: swig example with Reventador Data

REIS windows click the "Done" button to close the window. Avoid using the small "x" in the corner of the window to terminate because **R** does not know you have finished yet.

You can view spectra of the signal (SPEC) , spectrograms (SGRAM) and wavelet transforms (WLET). To illustrate, left click on this trace around t=1200 and t=2000, which windows the harmonic tremor part of this explosion. Click middle mouse to zoom in, or select one of the buttons at the top to analyze the time series in the (selected) sub-window. Choose *WLET* to show the wavelet transform of the harmonic tremor and important time variations of the volcano during eruption.

The **swig** program is normally run in interactive mode. In that case, once it is started **R** is waiting for the user to select traces and buttons for activating a variety of programs and analysis routines. Selection of traces is accomplished by clicking on the traces, one or more times depending on what is desired. The program needs to know what to do with the selections once that process is over, usually by clicking on a button around the perimeter of the screen. In the next example we will restrict the analysis to just the vertical motion seismic data, at least for now. If you expand the screen, you can re-arrange the buttons by clicking on the refresh button.

swig is a general analysis program designed for earthquake studies. It uses the **RPMG** *R*eally Poor Man's GUI package to navigate between seismic traces and various analysis procedures. Once the program is started it waits for the user to select on the screen a variety of operations, determined by the user via the button selection, STDLAB. In the main event loop, the user may click on the screen with the left mouse button to hi-light specific traces or windows in the panel. The right mouse click terminates the clicking sequence and a decision is made on what to do, unless a button has been clicked. Generally, one click selects a specific trace, two clicks specify a trace and window in that trace. If the clicking is terminated immediately, before a left mouse is clicked, the program stops and returns NULL. If it terminates after 1 click, a refresh screen command is produced. If there are two or more clicks, and no button is pressed, the last two clicks are used to zoom in the window.

If a button is clicked, however, the program uses the number of clicks to determine which traces to process and what to do. For example, if the "PickWin" button is selected, a new **swig** is spawned where the program gathers all the components for that station, Usually Vertical, North and East, although in the presence of acoustic channels they will also be displayed. The new window is called with a new set of Buttons set up specifically for picking the P, S and Acoustic arrivals. Once that window is finished, focus reverts to the main window and the new picks are registered. Selecting the "SavePF" button will save the new picks to a file for later use.

As another example, if the user clicks twice in a trace panel, and then selects the WLet Button, a wavelet transform of the selected time window is calculated and a special new screen is exposed where the user is now focused until that session is finished by clicking "Done".

4 Buttons in swig

in **REIS** buttons are defined as **R** functions.

Each Button has different properties based on the requirements for that process. Some buttons expect more than one click to operate properly, others are simple buttons that control the look and feel of the panel. For example, the "restore" button reverts the panel to its original time window. It can be pressed any time and the window will redraw and resize. Each button includes a small set of instructions designed to accomplish a specific task. There are many buttons currently defined, some described below, and there is mechanism for users to make their own on the fly. This is the great power of **RPMG** and **swig** . For user defined buttons see Section 6.

4.1 Example: Coso Geothermal Event

```
data(GH)
numstas = length(GH$STNS)
```

In this example, taken from the geothermal field at Coso, California, there are 49 stations, most of which have three components (Vertical, North and East), although there are a couple of stations that are missing some of the components. This situation is not atypical of earthquake seismic data recorded in the field. If we show only the vertical component traces (Figure 2), The plot is more manageable and easier to view:

```
##############   code
verts = which(GH$COMPS == "V")
STDLAB = c("DONE", "QUIT",  "NEXT","PREV",  "zoom in", "zoom out", "refresh", "restore",
"PickWin", "XTR", "SPEC", "SGRAM" ,"WLET", "FILT", "Pinfo", "WINFO", "PTS", "YPIX", "WPIX
swig(GH,  sel=verts,STDLAB =STDLAB,  SHOWONLY=TRUE)
```

```
dev.off()
```

We see that the stations here are 'mixed up', i.e. arriving at different times.

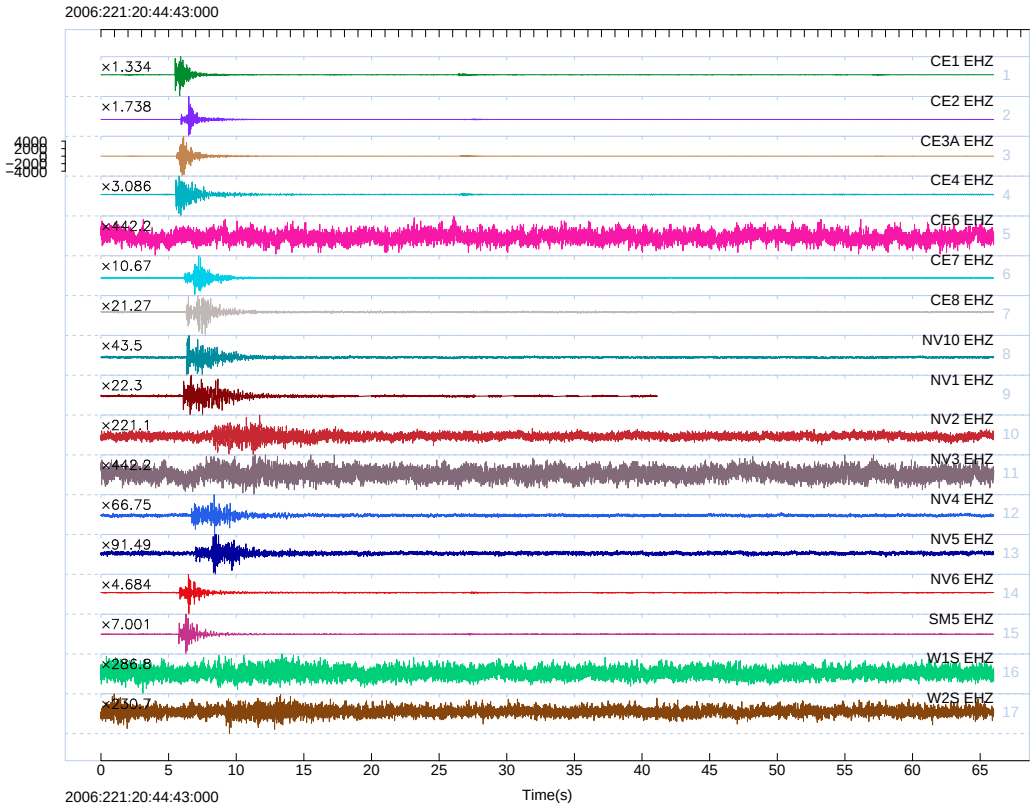

Figure 2: Example of Swig

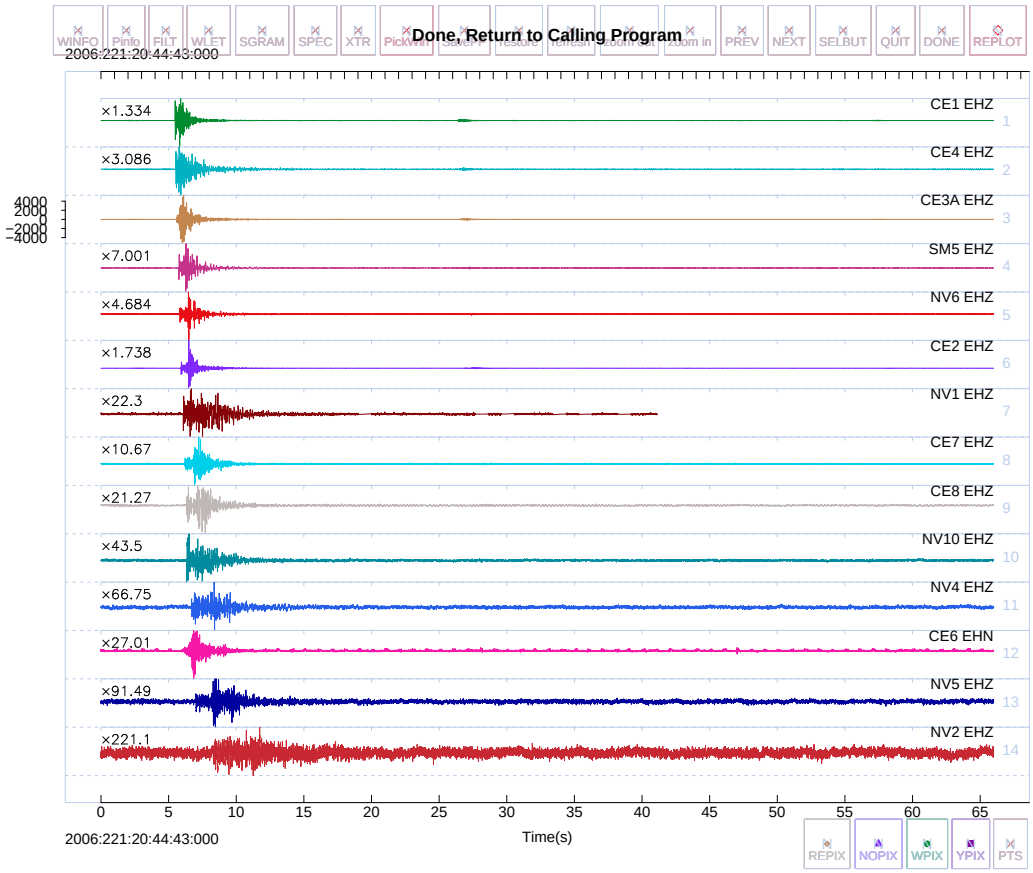

Figure 3: Coso vertical components ordered

```
###############    code
vertord = getvertsorder(GH$pickfile, GH)
swig(GH,  sel=vertord$sel,       STDLAB =STDLAB,  SHOWONLY=FALSE)



dev.off()
```

A seismic event is usually stored as a combination of waveform information and meta-date associated with the phase arrivals. Phase arrivals are commonly called "picks" since and analyst had to pick the arrival times from a representation of the seismic signals, either on a computer or on a paper record. The picks are stored in **REIS** in a list structure called pickfile which is an optional component of the name waveform structure. The pickfile structure is a list comprising several sub-lists with important information associated with stations and the event (earthquake) source.

```
names(GH$pickfile)
```

```
 [1] "PF"       "AC"       "LOC"      "MC"       "STAS"
 [6] "LIP"      "E"        "F"        "filename" "UWFILEID"
[11] "comments" "OSTAS"    "H"        "N"
```

For now we consider the most relevant meta-data,

```
names(GH$pickfile$STAS)
```

```
 [1] "tag"   "name"  "comp"  "c3"    "phase" "sec"   "err"
 [8] "pol"   "flg"   "res"   "lat"   "lon"   "z"
```

which is a list of vectors, one for each meta-datum and one element each for each station that has meta-data. We see in this example there are a couple of picks per station, some picks are on the vertical components and some are on the North component or East, there are P and S-wave phase picks.

```
data.frame(cbind(name=GH$pickfile$STAS$name, comp=GH$pickfile$STAS$comp, phase=GH$pickfil
```

```
   name comp phase   time       lat         lon
1   CE1    V     P 48.476   36.0131   -117.8025
2   CE4    V     P 48.532   35.9998   -117.8023
3  CE3A    V     P   48.6   36.0145   -117.8198
4   SM5    V     P  48.74  35.99965 -117.830261
5   NV6    V     P 48.812   35.9823   -117.8076
6   CE2    V     P 48.876   36.0337   -117.7883
7   NV1    V     P 49.072   35.9827   -117.7649
8   CE7    V     P 49.176    36.053   -117.8046
9  NV10    V     P 49.312 35.999056 -117.745194
10  CE8    V     P 49.292   36.0512   -117.8387
11  NV4    V     P 49.688   36.0477   -117.7403
12  NV5    V     P 49.996   36.0839   -117.7536
13  NV2    V     P 51.292   36.0255   -117.6213
14  CE1    N     S 48.752   36.0131   -117.8025
15  CE4    N     S 48.872   35.9998   -117.8023
16 CE3A    N     S 48.908   36.0145   -117.8198
17  SM5    N     S 49.216  35.99965 -117.830261
18  NV6    N     S 49.372   35.9823   -117.8076
19  CE2    N     S 49.444   36.0337   -117.7883
```

```
20  CE6    N     S 49.704 36.033665 -117.772726
21  CE7    N     S 49.876    36.053   -117.8046
22  CE8    E     S 50.316   36.0512   -117.8387
23  NV4    N     S 50.984   36.0477   -117.7403
24  NV5    N     S  51.28   36.0839   -117.7536
```

We also store event information:

```
names(GH$pickfile$LOC)
```

```
 [1] "yr"     "mo"     "dom"    "hr"     "mi"     "sec"
 [7] "jd"     "lat"    "lon"    "z"      "mag"    "gap"
[13] "delta"  "rms"    "hozerr"
```

Using this information we can associate the p-pick with the waveforms, match the timing information and plot together. finally we add the picks to the section (Figure 4):

```
###############   code
apx = uwpfile2ypx(GH$pickfile)
swig(GH,  sel=vertord$sel, WIN=c(0, 20), APIX=apx, STDLAB =STDLAB,  SHOWONLY=FALSE, velfi
```

```
dev.off()
```

Brief documentation for buttons (see 6) in the **swig** program can be seen by calling the documentation function, either for a specific button, as in:

```
PICK.DOC('WLET')
```

```
DONE = wavelet analysis
```

or for all possible buttons (not shown here because it is a long list).

```
PICK.DOC()
```

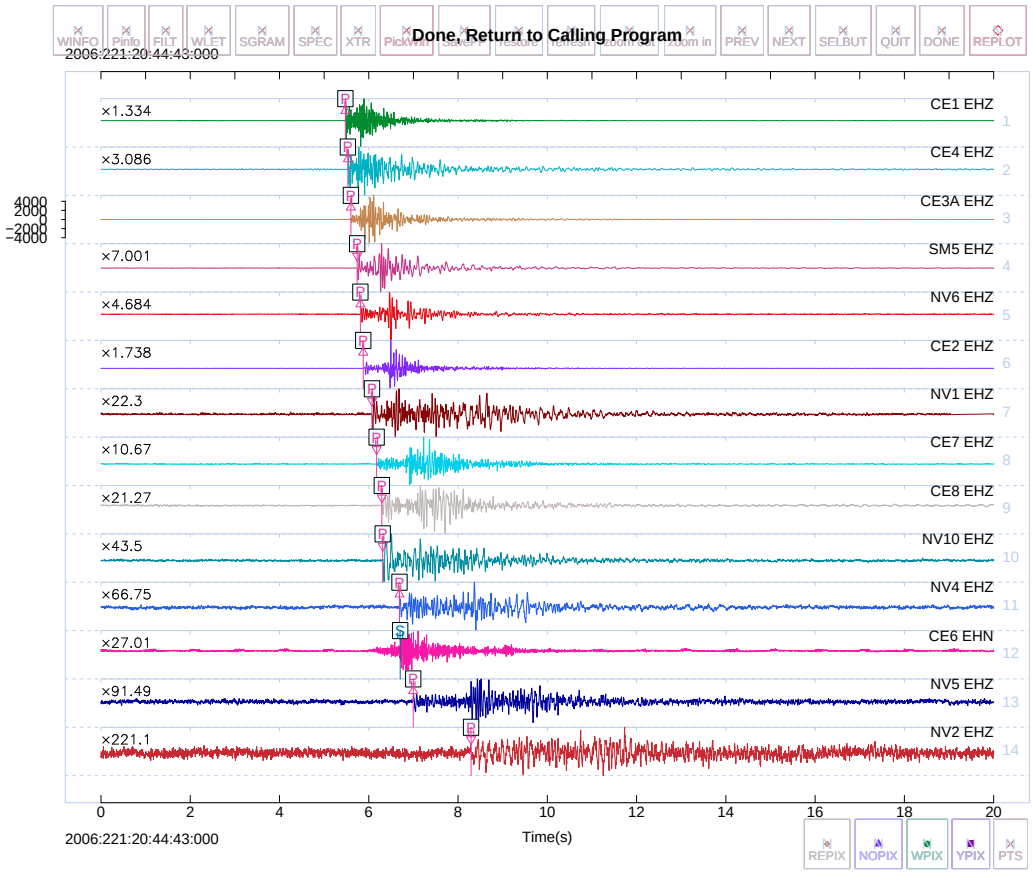

Figure 4: Coso vertical components with arrival picks

5 Seismic Data I/O

One of the big problems with seismic data is format and exchange. Unfortunately, seismologists spend an inordinate amount of time writing codes to reformat data so that it conforms with one or anotehr programs that are commonly used. Even though there are standard formats defined and in use today, many times these standards are not adhered to. In many circumstances the original definitions were too restrictive and investigators chose to extend the format in one way or another, making the standard "non-standard". A case in point is the SEGY standard and the PASSCAL-SEGY modification.

Another problem with exchange of seismic data is platform compatibility. To get a good binary format that is compatible on MAC, Windows and Linux systems is apparently difficult. This is further complicated by differences in CPU models (e.g. 64 bit versus 32 bit) and other compiler issues. I discovered some years ago that on some systems a "long int" is misnamed and is actually defined as a "short" This can cause havoc when reading in binary format data.

A few (somewhat) standard data formats can be read in directly in **REIS** . These are `SAC` and `SEGY` as defined by PASSCAL-distributed software. I have not written an **R** function for reading `SEED` format, but it is probably not too difficult. Maybe in the future.

I am currently developing a new package called `TELES` aimed at analysis of teleseismic data extracted from the IRIS DMC web site. The code has tau-p code for predicting global travel times. This work is still in progress. `TELES` currently works in LINUX and MAC environments and can be obtained by contacting me directly.

5.1 SAC format

SAC format data can now be read directly using native **R** binary codes. Earlier I/O functions in package `SACR` relied on C-code for the binary input, and this lead to some problems when transferring data across platforms.

The basic code for I/O on SAC data is:

```
j1 = JSAC.seis(f1, Iendian=1, HEADONLY=TRUE , BIGLONG=FALSE, PLOT=FALSE)
```

This is a short explanation of the arguments to `JSAC.seis`.

f1 vector of file names to be extracted and converted

Iendian Endian-ness of the data: 1,2,3: ”little”, ”big”, ”swap”

HEADONLY logical, TRUE= header information only

BIGLONG logical, TRUE=long=8 bytes

PLOT logical, whether to plot the data after reading in

Here *f1* is the path to one, or many, file names on the local system. When HEADONLY=TRUE only the SAC header is returned, and this can be used to set up the input of large digital signal files. The other arguments are important for making **REIS** platform independent. Argument *Iendian* is critical if the data were created on one platform transferred and read in on another. This argument refers to the “endian-ness” (byte order) of memory in the computer. In **R** one can find out the “endian-ness” of the system by accessing the variable

```
print(.Platform$endian)
```

```
[1] "little"
```

If data is created on the same system on which it is analyzed, and you stay consistent, there should be no problem. The problem of compatibility arises when data is shared across platforms. If you know what the endian-ness of the data is from the platform where the data was written in binary format and it is different than your system, use “swap”. Else, stay consistent. My desktop Linux machine and my laptop MAC are both “little-endian”. My older SUN computers were “big-endian”.

The *BIGLONG* argument was introduced because the SAC header has both long and short integer numbers. The issue stems from the fact that many systems (32 bit) do not recognize the LONG definition and internally convert to short, i.e. long is defined as 4 bytes. This can create a problem when transferring data created on a 64 bit machine to a 32 bit machine, and vice versa. So, if the format of the source machine is known - use that for the *BIGLONG* argument to indicate how to treat LONG ints.

5.2 SEGY format

SEGY formatted data follow the same convention that SAC data do, except that there is slightly different information in the header.

5.3 WIN format

There is a routine for reading `WIN` format from Japan, in a separate package called `WINR`. These codes were written in C, actually converted from the original FORTRAN code. They are not platform independent and they require re-compilation when converting from Windows to Linux types of systems. While they work well on my Linux system, I have had trouble getting them to work on different systems when the endian-ness is changing and the BIGLONG problems arise. You can try to use these, but I recommend simply converting WIN format to some native **R** format and reading the files in **REIS** .

5.4 UW format

There are many routines in **REIS** for handling UW format seismic data. UW format comes from the University of Washington and is used for earthquake event data. In that case many traces are stored for each event, arrival time information is stored in a pickfile, as well as polarities. Event location and focal mechanism solutions are also gathered and saved in the RSEIS list. See package **RFOC** for instructions on how to plot and manipulate focal mechanisms.

5.5 REIS format

One way to store data is in native **REIS** format. In this case one might read in the data in one of the previous formats and follow with a save to a binary **R** file on the local system. Then consequent I/O is simply a load command in **R** . I use this method when I have isolated a specific section of data that I am working on and need to read it for different purposes on different platforms, or share it with others.

As an example, suppose I have isolated a set of date/times that have events of interest. The event times, or windows, are stored in a list of $day, hr, s1, s2$ where $s1$ and $s2$ are starting and ending seconds for the event.

A database (DB, see 9) has been created earlier that describes the location of the SEGY files and their content. I use RSEIS program *Mine.seis* to extract the selected time window from the full data set. Here is snippet of code:

```
for(i in 1:length(chugs$day))
{

  print(i)
```

```
  at1 = chugs$day[i]+chugs$hr[i]/24 + chugs$s1[i]/(24*3600)

  if(chugs$s2[i]>3600) {
    at2  = chugs$day[i]+(chugs$hr[i]+1)/24 + (chugs$s2[i]-3600)/(24*3600)

  }
  else
    {
      at2  = chugs$day[i]+chugs$hr[i]/24 + chugs$s2[i]/(24*3600)
    }

  CH = Mine.seis(at1, at2, DB, usta, ucomp)

  fnsave =  paste(sep=".", Zdate(CH$info, sel=1, t=0), "RCHUGseis")
  print(paste(sep=" ", "Working on",fnsave))
  save(file=fnsave, CH)
  ##  sbut = swig(CH, sel=which(CH$STNS=="CAL") )

}
```

The Mine.seis call extracts the data from the database and the data is saved in the file *fnsave* with the **REIS** list named "CH".

In the future this data can be recalled in **REIS** by loading. Here that operation is put in a loop that breaks when the `QUIT` button is clicked in **swig**

```
for(i in 1:length(LCHUG ))
{

  load(LCHUG[i])
  sbut = swig(CH, sel=which(CH$STNS=="CAL" & CH$COMPS %in% c(VNE, IJK[c(1,2)] ) ) )
  if(sbut$but=="QUIT") { break }

}
```

Data stored in this format can be shared with others using **REIS** (or other **R**) software. The advantage is that the data will work on any platform (Linux, MAC or Windows) seamlessly.

5.6 ASCII format

Data may be stored in simple ASCII format and read in to **R** . To use **swig** , however, a proper list should be created. In this section I will present an example illustraing how to create the appropriate list for input into swig.

Suppose I have a data set consisting of seismic, infrasound and gravity recordings stored in 3 different files on disc. First the data is loaded into R via any means available (scan, read.table, load, etc...).

Here, create two time series using ricker wavelets and combine them together for analysis in swig:

```
freq1=1/50
dt1=1/100
nw1= 300/dt1
g1 = genrick(freq1, dt1, nw1)
date1  = recdate(45, 11, 11, 4, yr=2011)
sig1  = prep1wig(wig = g1, dt = dt1, sta = "STA1", comp = "CMP",
    units = "BLAH", starttime =date1 )
freq2=1/300
dt2=1/100
nw2= 100/dt2
g2 = genrick(freq2, dt2, nw2)
date2  = recdate(45, 11, 11, 4+100, yr=2011)
sig2  = prep1wig(wig = g2, dt = dt2, sta = "STA2", comp = "CMP",
    units = "BLAH", starttime =date2 )
```

Combine the wiggles into one list, and prepare for swig:

```
SIG = list(sig1=sig1[[1]], sig2=sig2[[1]])
EH=prepSEIS(SIG)
```

Now they are ready for plotting:

```
swig(EH, SHOWONLY = TRUE)
```

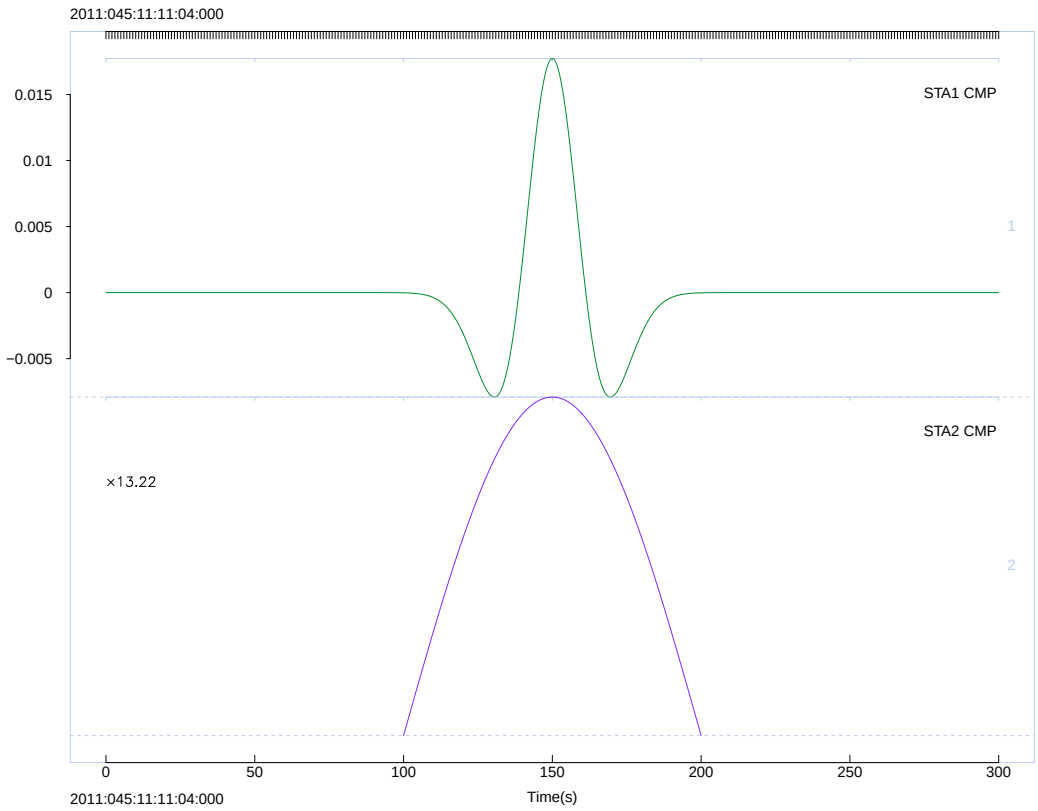

Figure 5: swig example from prepSEIS

```
dev.off()
```

6 Defining New Buttons

The program **swig** attains its real strength from its flexibility in defining new processes and actions to be applied to time series typical of seismic and geophysical applications. The codes was designed to allow the user maximum control of processing while maintains the organizing principle of structured coding. Information is passed from the main **swig** session to defined functions via buttons and instructions contained in the associated button definitions.

One can create new buttons in **REIS** by defining a function and calling it by clicking. There is a lot of flexibility in **R** because of the way data can be stored in expandable lists.

In **REIS** the basic structure is list with station names, component names, timing information and digital signal data. This data structure can be passed and modified by buttons in **REIS** . The basic function has two arguments, typically called, "nh" and "g" in my codes. These are passed into the button definition, acted upon and then returned, maybe in some modified form. Most Buttons do not modify the waveforms structures, but some do, like the filtering functions.

6.1 Button example

As an example of a case that does change the waveforms, consider the BUTTON that takes takes several clicks on traces and reverses polarity (flip selected traces). The definition of this function is:

```
FLIP<-function(nh, g)
  {
    Nclick = length(g$zloc$x)

    if(Nclick>1)
      {
        nc = 1:(Nclick-1)
        lnc = length(nc)

        ypick = length(g$sel)-floor(length(g$sel)*g$zloc$y[nc])
        ipick = unique( g$sel[ypick] )

        cat("FLIP: pwig POLARITY REVERSED: "); cat(ipick, sep=" " ); cat("\n")

        for(JJ in 1:length(ipick) )
          {
            jtr  = ipick[JJ]
            nh$JSTR[[jtr]] = (-1)*nh$JSTR[[jtr]]
          }
      }
    else
      {
        cat("FLIP: No traces selected: Try Again"); cat("\n")

      }
     g$zloc = list(x=NULL, y=NULL)

    g$action = "replace"
    invisible(list(NH=nh, global.vars=g))
```

```
}
```

Once the `FLIP` function is defined (and sourced) it can be added to the vector of buttons and executed within the *swig* session. The number of clicks and their locations are passed to the button definition via the "g" (global parameter) list. The "g" list contains many attributes that control the plotting and appearance of the plot. It has the selection vector "sel" that indicates which traces are plotted from the "nh" structure.

A signal called "action" is returned to swig to convey what to do with the returning changed parameters. Currently there are seven action signals that can be sent back to the swig main code.

The action options are:

continue Default Action

donothing Do nothing in the main code (commonly used)

break Break out of the main loop

replot Replot the main panel

replace Replace the current nh list with the modified list

revert Revert back to the original nh data prior to changes

exit Exit the program

Many defined buttons depend on the number and location of clicks on the screen. The button may have some logic embedded that has to be tested or vetted prior to execution to avoid crashes. Some buttons require, for example, that a time-window be defined on each traces prior to analysis. In that case there must be an even number of legitimate clicks to proceed. a good button will test for possible misuse before proceeding with the analysis. If the number of location of clicks is somehow incorrect, a warning should be issued and a "donothing" action command returned to swig.

The main ingredients of button definition in *swig* are a few parameters that can be used to extract and manipulate the passing structures. First is the number of clicks passed, here extracted by accessing the output of the locator function stored in the "g" list as zloc:

```
Nclick = length(g$zloc$x)
```

Since the last click saved in `zloc` is the click on the button itself, it is discarded and only the first `(Nclick-1)` points are used. In the FLIP function defined above `ypick` are the panel locations of the clicks and `ipick` are the selected traces associated with those clicks. The `JJ` loop selects only those traces indicated and reverses their polarity. The *g$action* indicates that on return the traces are to be replaced by the list in function.

If it is necessary to open a new plotting device it might be useful to store the dev number for later use, passing it through the "g" list. In this small code snippet I check to see if this device is already available. If not open a new device.

```
if(PLOT)
        {
          if(is.null(g$ternmatDEV))
            {
              dev.new()
              g$ternmatDEV =dev.cur()
            }
          else
            {
              dev.set(g$ternmatDEV)


            }
```

And this should be finished with setting the device focus back to the main window when leaving the function environment:

```
dev.set( g$MAINdev)
```

6.2 Accessing Button Functionality

Finally, I show here how to install and access the functions described in the previous section on defining a new button.

Once a button such as `FLIP` is defined and sourced or pasted into an **R** session, it can be called from within a **swig** session by adding it to the list of available buttons. The standard (default) list of buttons is defined as a vector of functions called `STDLAB`:

```
STDLAB = c("REPLOT","DONE", "SELBUT", "PSEL","LocStyle",
```

```
"ZOOM.out", "ZOOM.in", "LEFT", "RIGHT", "RESTORE",  "Pinfo","WINFO",
"XTR", "SPEC", "SGRAM" ,"WLET", "FILT", "UNFILT", "SCALE", "Postscript")
```

Naturally, `STDLAB` can be replaced by an alternative, although to insure that there are at least some buttons always present for navigation, a minimal list of buttons is always present in **swig** . To see these try executing:

```
swig(GH, STDLAB = c("TEST"))
```

these are the so called "fixedbuttons". (Note that since `TEST` is not a function, wehen it is pushed a warning comes up indicating that.)

```
"REPLOT", "DONE", "QUIT", "SELBUT"
```

and the fixed "pick" buttons:

```
 "NOPIX", "REPIX"
```

The `REPLOT` button is always located on the upper right hand of **swig** so the screen can be re-drawn and the buttons re-established. If the screen is resized, the buttons may appear to go off the end of the plot and they will need to be replotted. See section ReSizing below.

Once user defined buttons are set (like `FLIP` above) they can be added to the list by calling:

```
swig(GH, PADDLAB = c("FLIP"))
```

The `FLIP` function can be accessed by first clicking on a one of the (traces) panels in *swig* and then clicking the `FLIP` button. Control is transferred to the user defined function, the GH list is modified, the action is *replace*, so the list is replaced and and control is returned to the swig environment for further user interaction.

7 ReSizing

R sessions generally are not especially aware of the graphics environments. When a device is called and plotting actions are determined the device characteristics are used to set the scales and units of the screen. If the user resizes the screen after the plot has been made, **R** may not be able to adjust properly. In that case the user should replot the existing plot so the correct aspect ratio and other coordinate systems can be set properly.

In **swig** this can be accomplished easily by clicking the `REPLOT` button at any time. The figure will be recast and the buttons will be redisplayed correctly.

8 Bugs and Problems

If there are more buttons defined than can fit on the top and bottom rows of **swig** or any GUI defined using package **RPMG** , they will go off the edge of the screen on the lower left side and disappear. I may fix this in the future, perhaps by assigning a button panel over the top and keep all defined buttons there. This would entail a major change and I have not considered implementing this at the present time.

If a button is depressed and careful error handling has not been established within the button, the **swig** session may crash. Since the user defines the action of the buttons it is virtually impossible to protect against this. I recommend coders pay attention to error handling.

9 Setting up an RSEIS Database

Often we have large datasets of continuous seismic data on several stations and several components. This would be the case for a temporary PASSCAL experiment, where data comes as station-component files in SEGY format, typically in time slices of 1 hour depending on the acquisition parameters. The files, as they are retrieved in the field using PASSCAL and REFTEK software are ordered by day or DAS number or some other method. and they are stored in some directory on the computer or disk.

If the station and component names have been written in the data headers, a simple **REIS** data base system can be created for easy access to the full data set. The database is organized as a simple R list such that searching, sorting, and data extracting are accomplished with standard R commands. The database is thus equivalent to a flat file, or spread sheet organization.

9.1 DB Example

In this example the data has been extracted from the field using REFTEK (REFraction Technology) 130 dataloggers. The IRIS corporation PASSCAL software package has a routine called ref2segy that was used to convert the data to PASSCAL-SEGY format. A similar program can convert the data to SAC format. These two standard formats are coded as "kind" 1 or 2, respectively. Other formats can be coded and added to the RSEIS package as needed.

Once the data is converted to a standard format, the files are stored as records on disk. In the following example, they are stored in folders starting with the token "RO" followed by the julian day. **REIS** routines read in the data file headers and create the data-base from which data can be extracted as events or continuous records.

```
#############  set directory
path = '/Users/lees/Site/Santiaguito/SG09'
pattern = "RO*"

###   get DB information
XDB  =  makeDB(path, pattern, kind =1)
```

Then data can be extracted by time, station and component. In this case 24 hours from one component.

```
##### select a station
usta = "CAL"
acomp = "V"

  #####   extract 24 hours worth of data
JJ = getseis24(DB, 2009, 2, usta, acomp, kind = 1)
```

10 MakeDB

In this document I will illustrate how to create a simple flat database for use in **REIS** . The data base is constructed from files, usually SEGY or SAC, but they could be native **R** files already processed so that conversion is not necessary (better still).

11 File Structure

The basic structure for this code is based on the output of a program written by PASSCAL called ref2segy (or ref2sac). After extracting data from disks in the field the "ref" files are dumped into a directory on the hard drive of a computer. The program ref2segy extracts the data from messy reftek format, and converts them to SEGY format. A log is created and other output useful for getting information about the field operations. For now we do not need to pay attention.

As an example, the files for my 2009 santiaguito experiment in Guatemala are stored on my computer as:

```
wegener% ls
/Users/lees/Site/Santiaguito/SG09
#########################
segyDB    R365.02/                2009:019:15:39.9026.log  2009:007:17:11.run
filesDB   R366.02/                2009:019:15:39.run       2009:007:17:11.SMI.log
R001.02/  R006.02/                2009:007:17:12.CAL.log   2009:007:17:11.KAM.log
R002.02/  2009:019:15:40.9024.err 2009:007:17:12.run       2009:007:17:10.KAM.log
R003.02/  2009:019:15:40.9024.log 2009:007:17:12.DOM.log   2009:007:17:10.run
R004.02/  2009:019:15:40.run      2009:007:17:11.DOM.err   2009:007:17:10.CAL.err
R005.02/  2009:019:15:39.9026.err 2009:007:17:11.DOM.log   2009:007:17:10.CAL.log
```

The actual waveform files are in the directories starting with "R00" etc. The Julian day is on each folder name. This data was recorded in late December, 2008 and into January 2009. so the high julian days are at the beginning of the experiment and the low day numbers at the end. (The spanning of the new year actually presents some date problems that need to be overcome.)

For example, a listing of one of the subdirectories is:

```
wegener% ls
/Users/lees/Site/Santiaguito/SG09/R002.02
```

```
########################
09.002.00.47.50.CAS.I
09.002.00.47.50.CAS.J
09.002.00.47.50.CAS.K
09.002.01.47.50.CAS.I
```

Note that the files have already been altered, in that information from the headers has been placed in the file names. This is not critical, but it is important to get information about the station name and component into the file headers.

12 makeDB

The program *makeDB* will read in the data once its is told where to look, what to look for and what format to use.

The call uses a path and pattern to read in the data, file by file and store the header (and other) information for quick access. The path variable is a pointer to the base location of the data to be extracted. The pattern argument is used to direct the the program to read in some information and ignore extraneous folders or files. In this example, all the data is stored in directories starting with "R0" so the pattern is simple. We use a wild card to get all the folders for this experiment.

```
path = '/Users/lees/Site/Santiaguito/SG09'
pattern = "R0*"
XDB  =  makeDB(path, pattern, kind =1)
```

The other parameters are critical and care must be taken to make sure they are executed correctly. The default parameters are:

- kind = 1
- Iendian = 1,
- BIGLONG = FALSE

12.1 kind

The *kind* argument signals **REIS** that the data is a specific format. The standards now are

- 1=SEGY
- 2=SAC
- 0=native RSEIS

The program reads in each file, extracts station name, component, sample rate and timing information from the files and saves these in a list. The SEGY and SAC formats are read in using native **R** binary read commands. If the data is already in **REIS** format, then the processing uses **R** command *load* to read the data in and extract from the list already available.

The two other arguments relate to the format that the data is stored in and depend on the computer system they are read on.

12.2 Iendian

Iendian is a flag indicating the endian-ness of the data and whether swapping needs to be performed. The byte-order ("endian-ness") is different for different operating systems. You can determine the endianness of a system by accessing the **R** .Platform variable,

```
.Platform$endian
```

```
[1] "little"
```

If the data was written on a little endian machine, then the little option should be provided. Likewise id big endian was used to create the data, and the machine reading it is also big-endian, then use big. If the machine writing the data and the machine reading the data use different conventions, then "swap" should be invoked.

```
endian: The endian-ness ('"big"' or '"little"' of the target system
          for the file.  Using '"swap"' will force swapping
          endian-ness.
```

12.3 BIGLONG

The BIGLONG variable is set so that data written with long=8 on a machine with long=4 can be accomodated. This problem arises mainly with SAC format data as the header for SAC data

calls for a long, even though most 32 bit machines actually use long=short. To determine what a machine is using one can query the .Machine variable in **R** :

```
.Machine$sizeof.long
```

```
[1] 8
```

If the size is 8 use TRUE, if 4 use FALSE.

If you get your data from someone else, or you download the binary files, you need to determine how to set these parameters. Having the wrong arguments may lead to **R** crashing, or even crashing the whole system.

13 Extracting Data

The purpose of makeDB is to allow quick and easy access to the data files and to make it easy to extract time slices from the large set of files.

The **REIS** program I use for small data sets like the one illustrated above is called *Mine.seis*.

With *Mine.seis* you give it a the database and it finds the files that need to be accessed, extracts the waveforms and glues the files together to get a single trace for each station/component. This list is suitable for plotting and processing in swig. (swig=seismic wiggle).

This is a short document explaining how to access data using the RSEIS package. RSEIS was created to make handling of seismic time series easy (or easier). Typically data is stored on disk in some binary format (SAC, SEGY or R-format) and read into memory. RSEIS manipulates the data and puts it into a structure in R called a list. It is easy to access complex data structures. One advantage of R is that the list can be extended, i.e. new meta-data can be attached to the data list simply in a way that does not generally interfere with other processes in RSEIS.

Programs for reading in data include JGET.seis, JSAC.seis, JSEGY.seis. I will not illustrate these here. I will assume these have already been called and the list is returned.

An example list is provided by RSEIS: GH.

```
data(GH)
verts = which(GH$COMPS == "V")
swig(GH,  sel=verts,  WIN=c(0, 25), SHOWONLY=TRUE)



dev.off()
```

If we want to extract a specific trace, say the vertical component of station CE8, try this:

```
iw = which(GH$STNS=="CE8" & GH$COMPS=="V" )
wig = GH$JSTR[[iw]]
dt = GH$dt[iw]
```

One could use the standard time-series (ts) routines that are in the base package of R:

```
par(mfrow=c(2,1))
plot.ts(ts(wig, deltat=dt))
title("Time Series plot using plot.ts")
plot.ts(ts(wig, deltat=dt), xlim=c(6, 10) )
title("Zoomed Time Series plot using plot.ts")



dev.off()
```

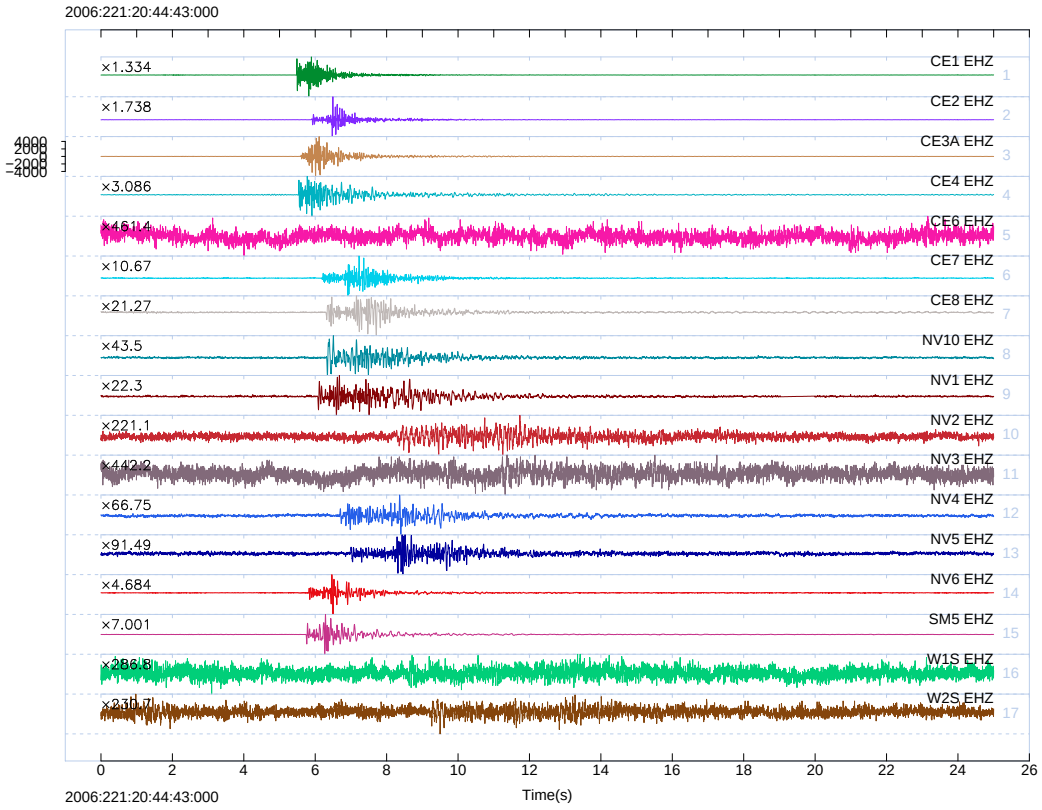

Figure 6: Swig Example Plot

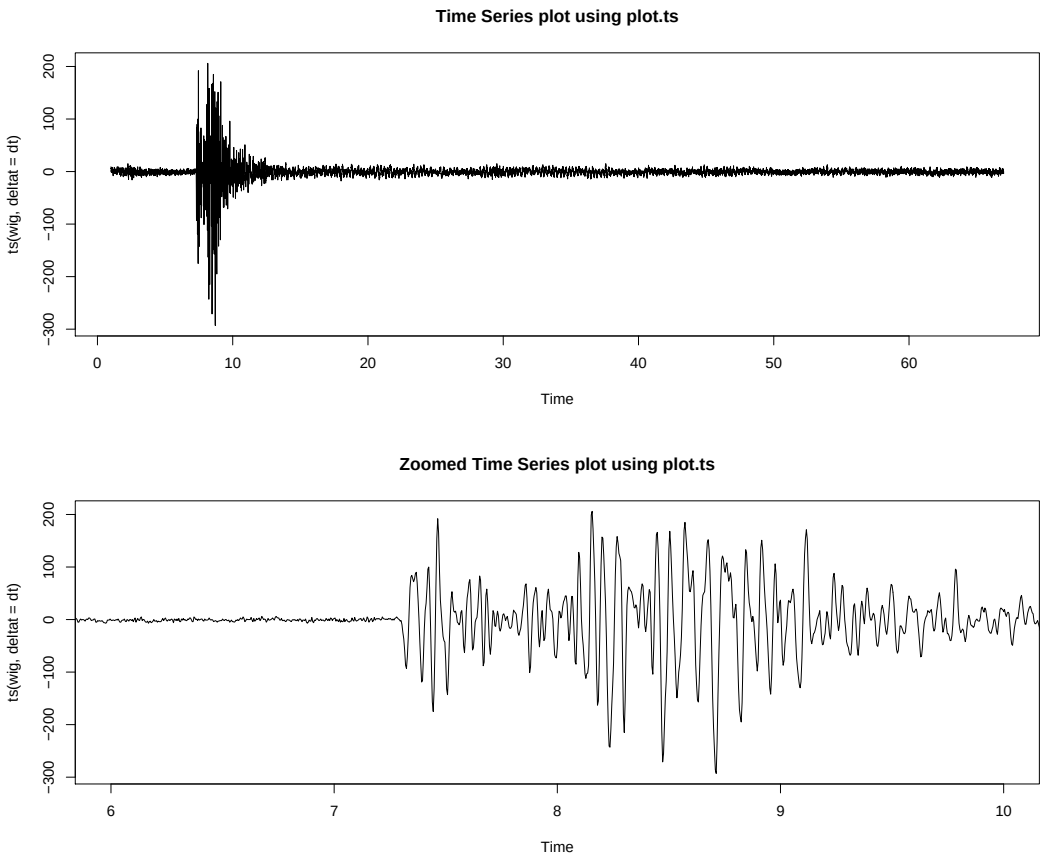

Figure 7: Swig Example Plot

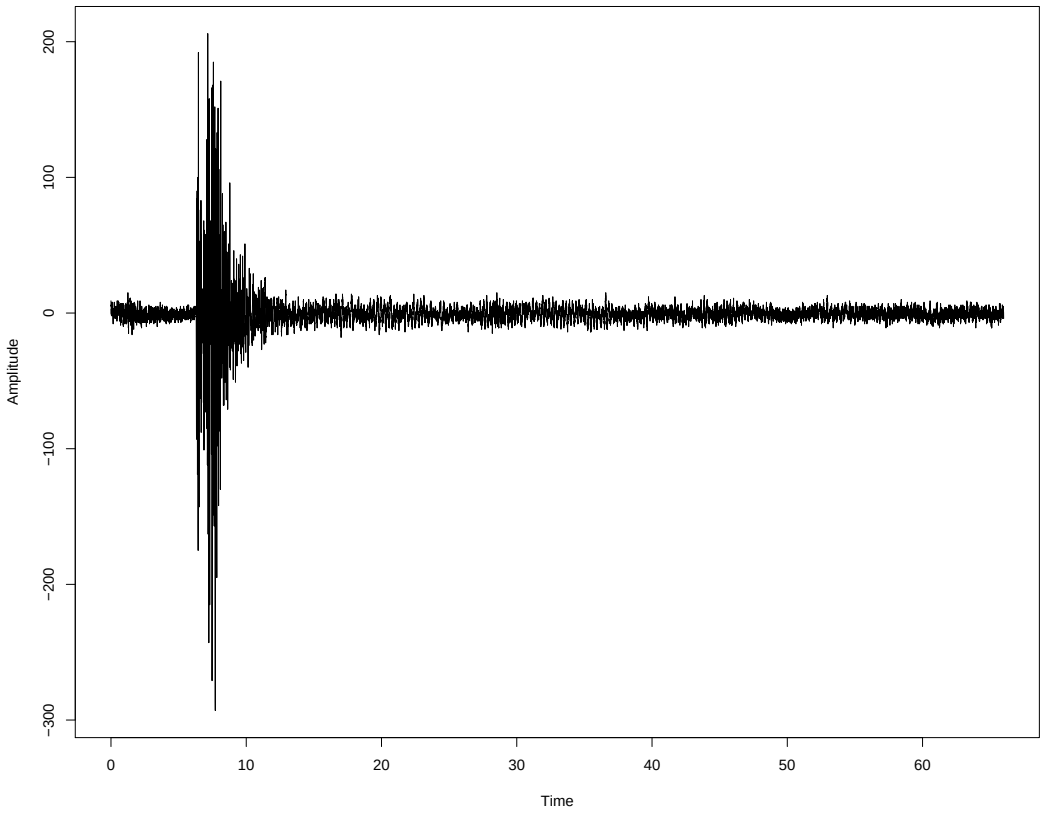

Figure 8: Create a wiggle and plot

Or a specific x-y time series can be created explicitly: The X-Y values would be:

```
x = seq(from=0, by = dt, length=length(wig))
y = wig
plot(x,y, type='l', xlab="Time", ylab="Amplitude" )
```

```
dev.off()
```

14 Appendix

The RSEIS package uses a set of lists that have specific components useful for plotting and manipulating seismic data. The main seismic record is a list that consists of time series and meta data (figure 9).

One component of the seismic record is the information on the time, sample rate and lengthof each trace. These are stored as vectors in the list info, see figure 10. .

A pickfile is a list of data structured with a broad range of meta-data associated with a seismic event. The event has a location, an error elliposid, arrival time information, a focal mechanism, etc... See figure 11.

```
dev.off()
```

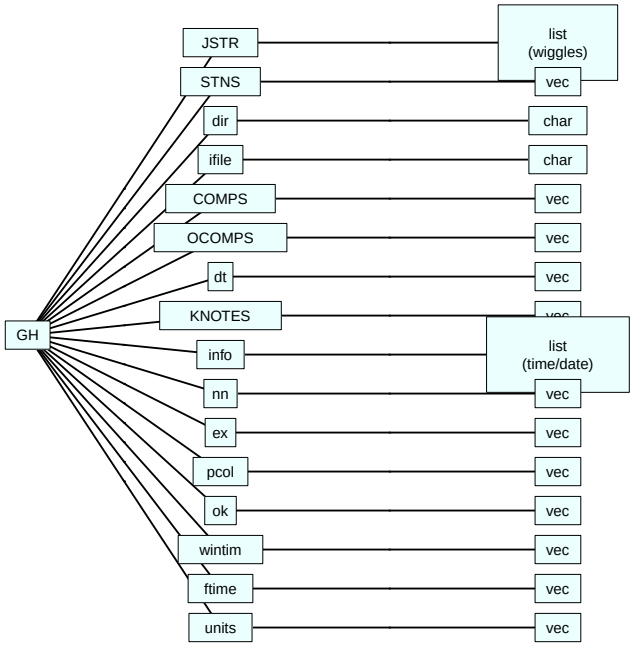

Figure 9: Documentation for RSEIS seismic data list

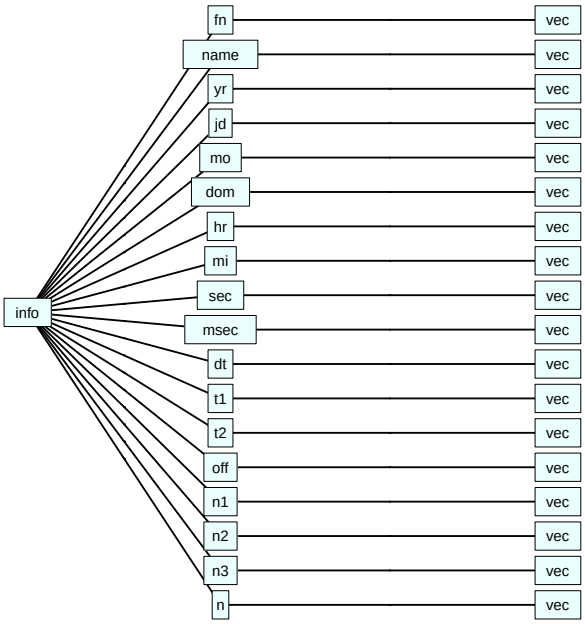

Figure 10: Documentation for info list

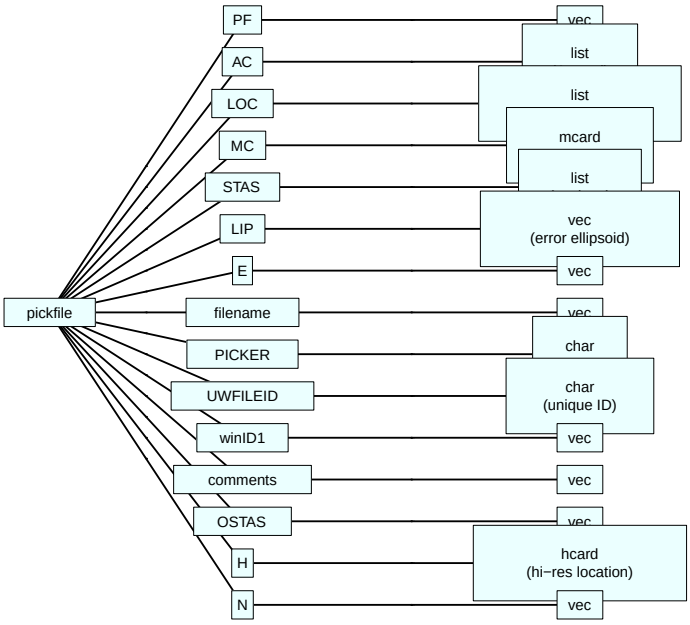

Figure 11: Documentation for pickfile list

References Cited