SeisR

Jonathan M. Lees
University of North Carolina, Chapel Hill
jonathan.lees@unc.edu

July 12, 2012

Contents

Chapter 1

Introduction

1.1 Introduction

1.2 Getting Started

This set of routines is intended to show show how to make something **R** .

R can be downloaded and installed on linux, windows or macOS machines by extracting files from the website and following the instructions.

Once the code is installed you may add on a variety of packages that enhance the base installation. Which packages you install among the thousands that are available depends on your needs and what you are doing. I do not recommend downloading all the packages. In some cases a function on one package may interfere with the same named function in another, unrelated package.

Install **R** from: `http://www.r-project.org/`

The **R** foundation updates the base **R** software about every 6 months. I recommend keeping the most current version available on your system. In some cases if you upgrade to a more recent version you may have to also re-install packages.

1.3 Installing Packages

If you have administrator access on your computer you may install packages and update them. This is easily done by starting **R** as administrator and telling **R** to exxtract a package and install it. In many cases one package may depend on others so the install process will also extract those packages that the original package depends on. You may extract several packages at once.

Once the packages are installed on a computer they do not need to be installed again. However, many packages are constantly being upgraded and I recommend also installing the updates.

To find out which version of **R** you are using you may query by,

```
vers = R.Version()
print(paste("I am using", vers$version.string))
```

```
[1] "I am using R version 2.15.1 (2012-06-22)"
```

```
Repository="http://lib.stat.cmu.edu/R/CRAN"
install.packages("akima")
packageStatus()
update.packages()
```

To see all the packages available go to the CRAN web site and see the link the PACAKGES.

In **R** you can download the list of packages by executing:

```
tennREPOS = "http://mirrors.nics.utk.edu/cran"
options(repos=tennREPOS)
AV =  available.packages(contriburl = contrib.url(getOption("repos")),
                         method, fields = NULL)
LAV = length(AV[,1])
```

There are 3902 packages in CRAN at the time of the creation of this document.

After a package is installed it can be invoked in an **R** session by calling the library function.

```
library(GEOmap)
```

If you are constantly using the same libraries you can put these commands in a file that is executed (sourced) every time **R** starts on your system.

1.4 Data Storage in R

All data are stored locally in the directory where R-was started, unless it is not specified. Data in **R** are usually organized as objects in one of the following kinds:

scalars numbers,

vectors sequences of numbers

matrices arrays of numbers

lists combinations of several objects

dataframe matrix of mixed mode objects (must have same length)

array higher dimensional matrix or array

Data are accessed by typing the name of the variable or by issuing a print command.

If you read in a vector of character strings that should be numeric you must convert the vector to mode numeric.

Here are some examples of setting these types of variables:

1.4.1 Scalar Examples

```
y = 3
x = 10
z = pi
w = x*y*z
h = x-y^z
```

1.4.2 Vector Examples

```
y = 1:10
h = rep(6.6, times=10)
x = y/30
z = sqrt(y)
w = z* y
w
```

```
[1]  1.000000  2.828427  5.196152  8.000000 11.180340
[6] 14.696938 18.520259 22.627417 27.000000 31.622777
```

Note: the bracketed numbers on the left are the index values of vector (or matrix).

Note: the colon operator creates a sequence of numbers.

Note: if you multiply 2 vectors they elements are multiplied element-by-element and a vector is returned. If you want the "dot" product you must sum the results

```
dotyz = sum(w)
print(dotyz)
```

```
[1] 142.6723
```

Other ways to create vectors:

```
x= runif(10)
y = seq(from=12, to=50, by=3)
```

1.4.3 Matrix Examples

```
y = matrix(1:20, ncol=4, nrow=5)
y
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

```
x =  matrix(runif(20) , ncol=4, nrow=5)
x
```

```
          [,1]       [,2]      [,3]       [,4]
[1,] 0.2644211 0.04856993 0.6225951 0.91758671
[2,] 0.5941977 0.65803509 0.9121923 0.64520937
[3,] 0.3460684 0.14155841 0.8941195 0.10508868
[4,] 0.7104925 0.11026551 0.1785283 0.00329403
[5,] 0.6454219 0.54138437 0.2002159 0.22526565
```

Note: the bracketed numbers on the left and top are the row and column index values of the matrix.

1.4.4 List Examples

Lists are heavily used in **R** . The are mixed vectors of several objects, perhaps scalars, vectors, matrices and other lists. They are very useful for conglomerating a lot of information into one object. Each member of a list may have a different mode (character, vector, numeric, etc.)

```
tel = 9995552456
ssn = "666-77-9898"
street = "555 Pensylvania Ave."
data1 = rnorm(5)
data2 = trunc(runif(10, 10, 100))
L1 = list(tel=tel, ssn=ssn, d1 =data1, d2=data2)
print(L1)
```

```
$tel
[1] 9995552456

$ssn
```

```
[1] "666-77-9898"

$d1
[1]  3.1387802 -0.8173817 -2.1770847  2.1484234  0.8920154

$d2
 [1] 69 32 70 83 76 52 54 96 11 71
```

1.4.5 Data Frame

A dataframe is a matrix (or a list) where each row has the same number of elements. A dataframe is basically a table, like a spreadsheet, with numbers, characters mixed together.

for example, we start with a list,

```
 N = 5
 names = vector(length=N, mode="character")
 for(i in 1:N) { k=trunc(runif(1, 5, 10)); names[i]= paste(sample(letters, k), collapse="") }
 Area = rep(919, 5)
 Pref = rep(555, 5)
 tel = trunc( runif(5, 1000, 9000) )
 data.frame(list(Name=names, Area=Area, Pref=Pref, tel=tel))
```

```
       Name Area Pref  tel
1   tyizpxa  919  555 6096
2 cmqstzwou  919  555 1075
3    ezauqy  919  555 1616
4    sgxthj  919  555 7289
5 mzwcfajdo  919  555 5325
```

1.5 Functions in **R**

All functions are called by using parentheses.

1.5.1 Files, IO

Getting data into an R session and saving data from a current R session are operations that often cause frustration with beginners. There are several ways to store data and this can be confusing. Here is a small bit of advice about data storage.

Data is usually stored as either ASCII (text) or a binary files. If the data is in a binary format not rcreated by **R** you will need to use a specialized program to read it in and you will need detailed information on how the data was written on the original computer where it was created.

If the data is in ASCII (text) format, there are a variety of ways to read it into **R** , depending on how it is organized in the file. There are specialized functions in **R** for reading files that are comma separated tables (CSV files). These are common for the output of spread sheet programs like MS-Excel.

read.table read a table of text

read.csv read a comma separated table

read.delim read a delimited table

readLines read a file line by line

scan low level read a file (see below)

1.5.2 Working Directory

After installation you should choose a working directory for **R** .

On a windows installation of **R** you will have a shortcut to Rgui.exe on your desktop and/or somewhere on the Start menu file tree, and perhaps also in the Quick Launch part of the taskbar (Vista and earlier). Right-click each shortcut, select Properties... and change the 'Start in' field to your working directory. *Windows I*

Two **R** commands can be used to get and set the directory where the user is working:

```
getwd()
setwd()
```

You may wish to always work in the default directory that you get when you start **R** . I do not recommend this approach.

Rather, for each project create a directory and work exclusively in that folder for the duration of that project. Keep your data and output files in that folder or other sub-folders there. When you start an **R** session, depending on which project you plan on working on, switch to that folder and load what you need there. Or, you may create a ".first" file and execute it so that it loads the appropriate libraries, data files, etc.

1.6 Important Commands

There are a few very important commands in R.

```
ls()
help(ls)
save(file="R_mystuff")
history()
help.start()
set.seed(2)
rep()
seq(from=1, to=20, by=2)
```

You may write all your **R** code into a file, save the file and execute it with the source command

```
source("filename.R")
```

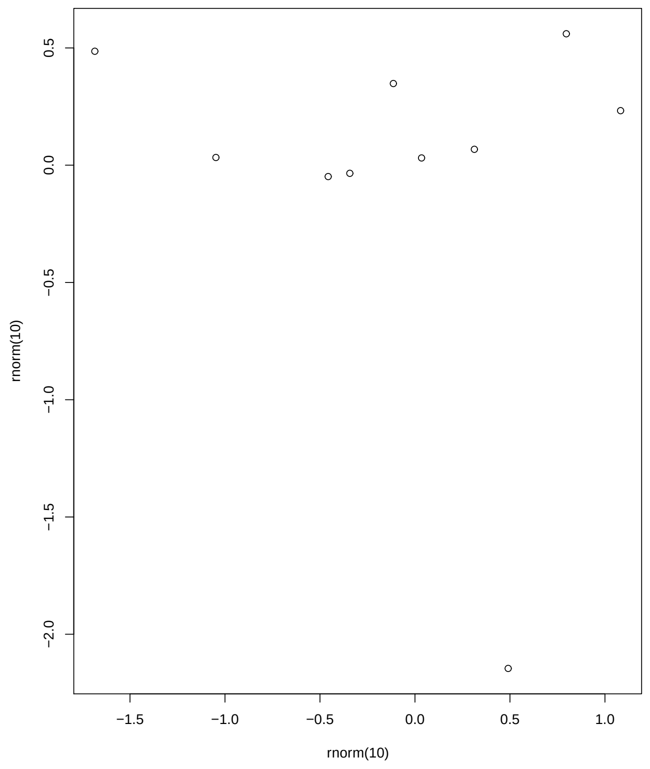

Figure 1.1: GEOmap Example **R**

1.7 Plotting and Graphics

Here is a plot:

```
JPOST(file="/home/lees/Mss/SEIS_BOOK/Intro/FIGS/STUBBER.eps", width = 8,   height = 10)
## par(mai=c(.2, .2, .2,.2))
plot(rnorm(10), rnorm(10))
dev.off()
```

1.7.1 Plots: par()

Chapter 2

Input/Output

2.1 I-O

2.2 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.

2.2.1 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.

2.2.2 SEGY format

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

2.2.3 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** .

2.2.4 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.

2.2.5 IRIS DMC data

Teleseismic data

2.2.6 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 3.10) 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.

2.2.7 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)
```

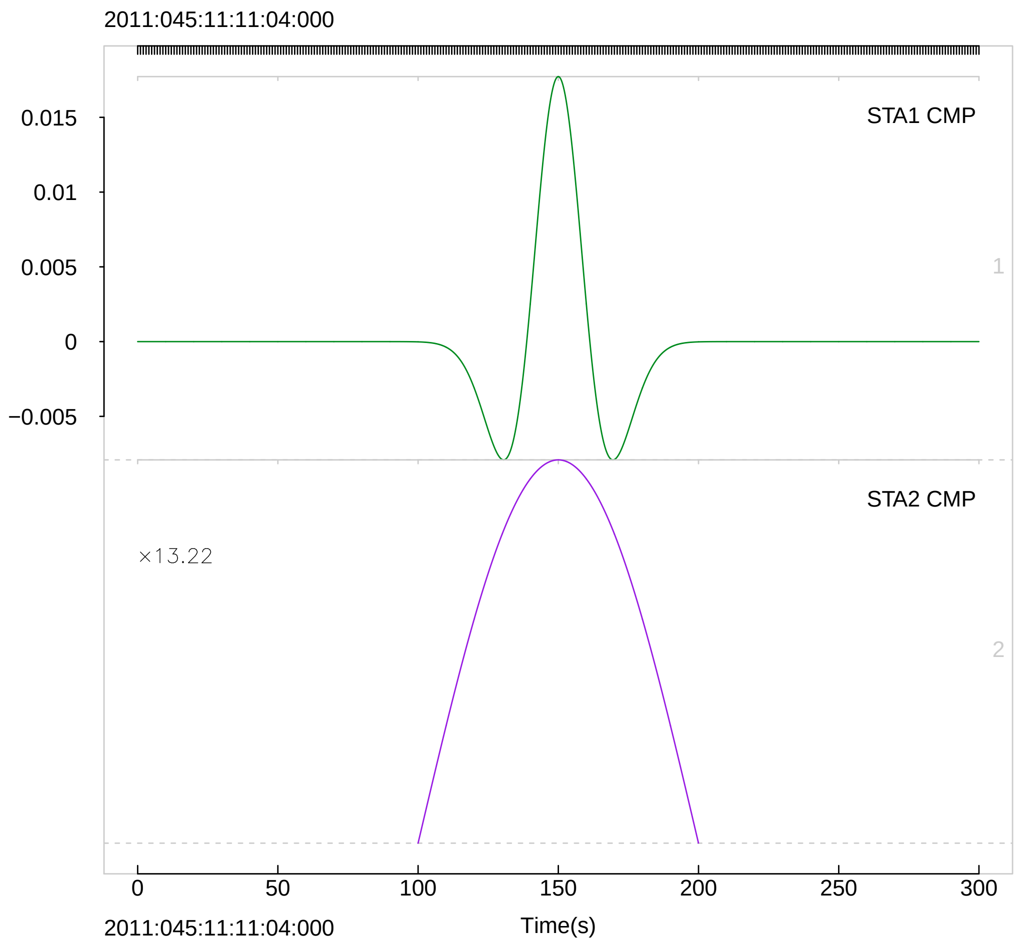

Figure 2.1: PLot showing swig window with signals 1 and 2.

2.3 makeDB

2.4 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).

2.5 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
/home/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
/home/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.

2.6 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 = '/home/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

2.6.1 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.

2.6.2 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** Ṗlatform 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.
```

2.6.3 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.

2.7 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).

Chapter 3

RSEIS

3.1 RSEIS

3.2 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.

3.3 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.

3.4 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)
```

3.4.1 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)
```

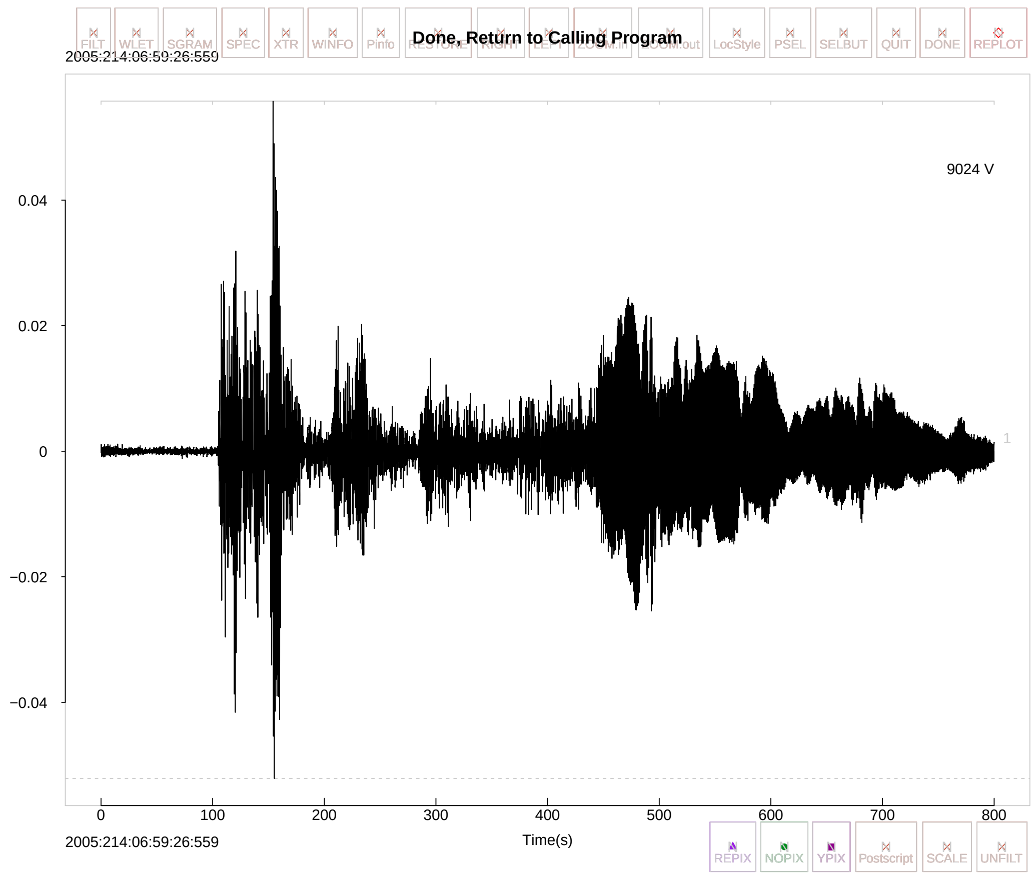

Figure 3.1: swig example with Reventador Data

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 4.1). 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 **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".

3.5 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 3.7.

3.5.1 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 3.2), 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", "Save
"PickWin", "XTR", "SPEC", "SGRAM" ,"WLET", "FILT", "Pinfo", "WINFO", "PTS", "YPIX", "WPIX")
swig(GH,  sel=verts,STDLAB =STDLAB,  SHOWONLY=TRUE)




dev.off()
```

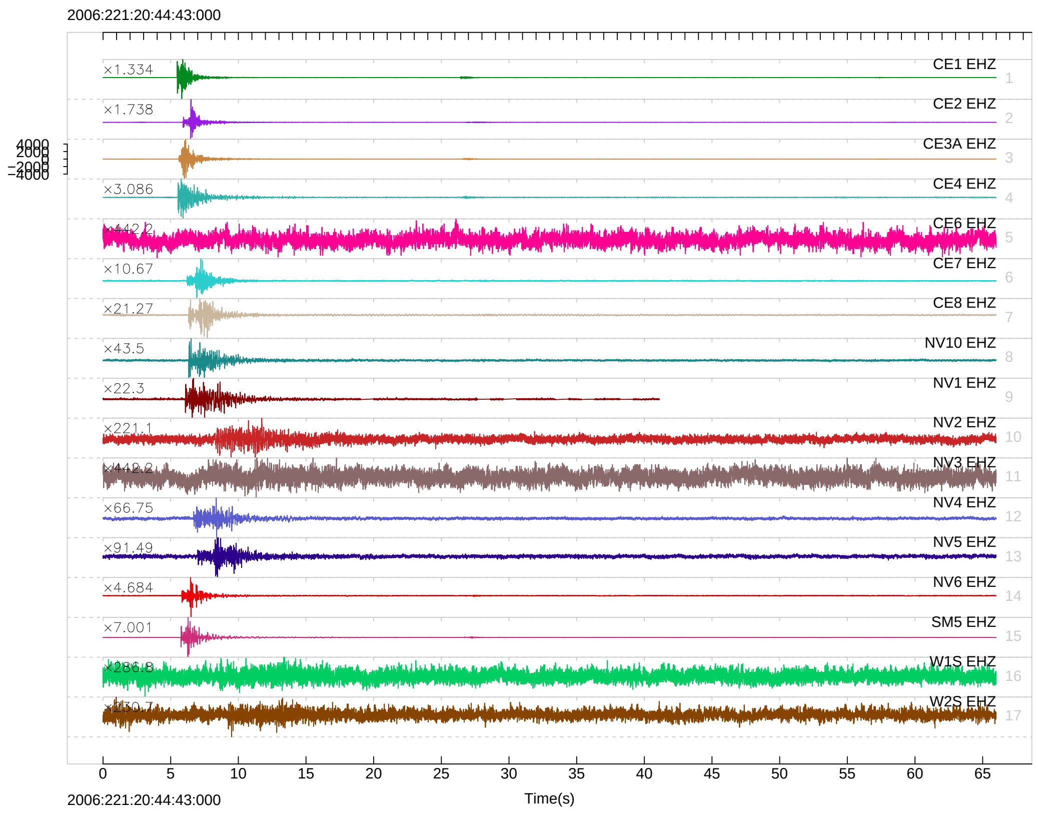

Figure 3.2: Example of Swig

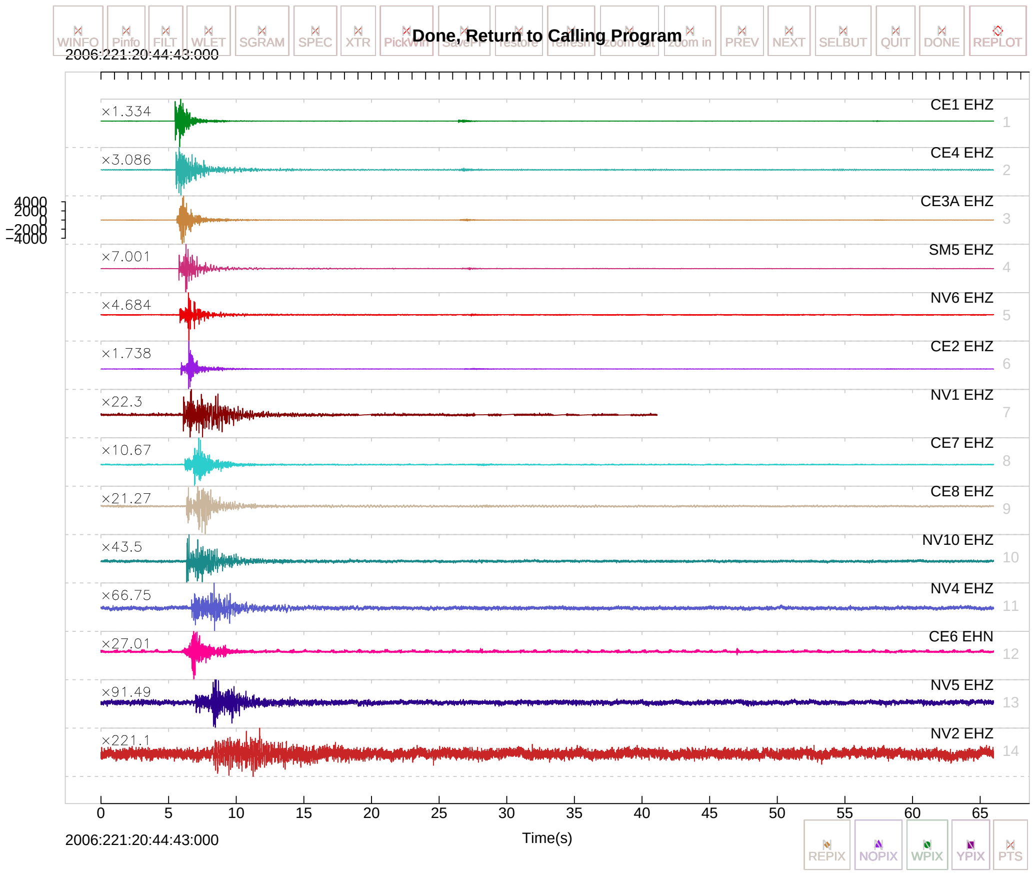

Figure 3.3: Coso vertical components ordered

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

```
##############    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$pickfile$STA
```

```
   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 3.4):

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



dev.off()
```

Brief documentation for buttons (see 3.7) in the **swig** program can be seen by calling the documentation function, either for a specific button, as in:

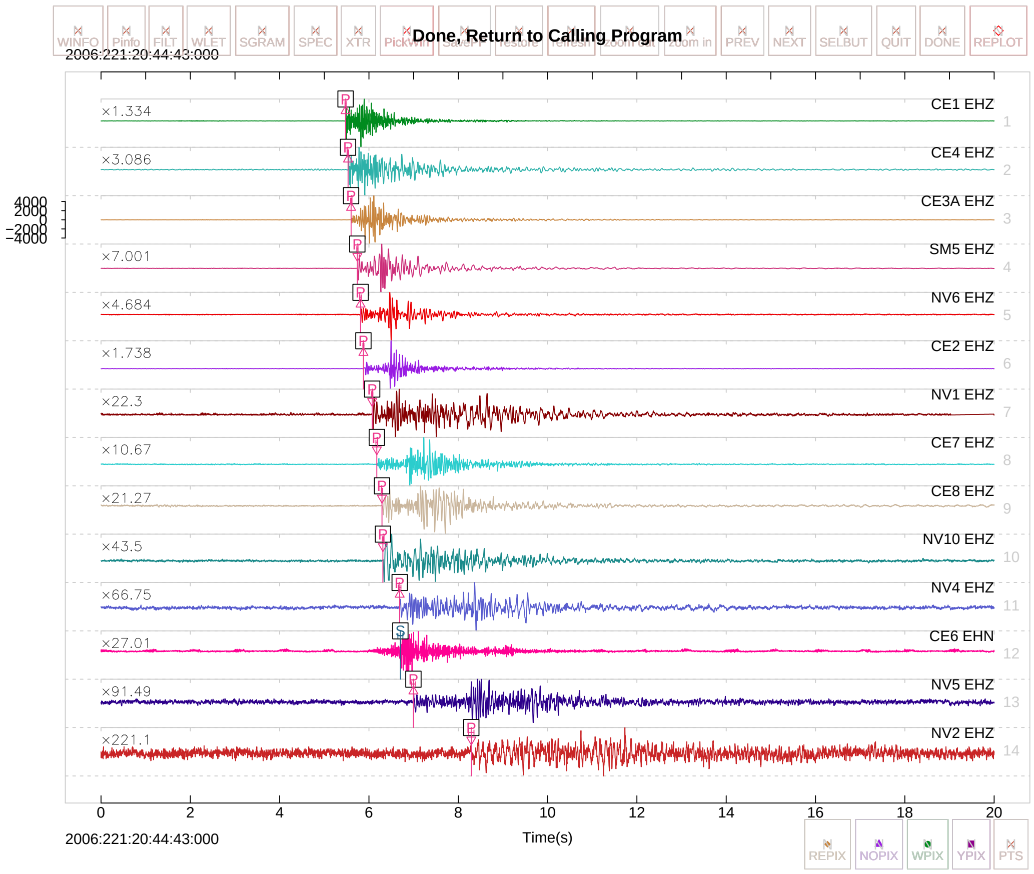

Figure 3.4: Coso vertical components with arrival picks

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

```
DONE = wavelet analysis
```

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

```
 PICK.DOC()
```

3.6 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.

3.6.1 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.

3.6.2 SEGY format

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

3.6.3 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** .

3.6.4 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.

3.6.5 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 3.10) 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.

3.6.6 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)
```

```
dev.off()
```

3.7 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.

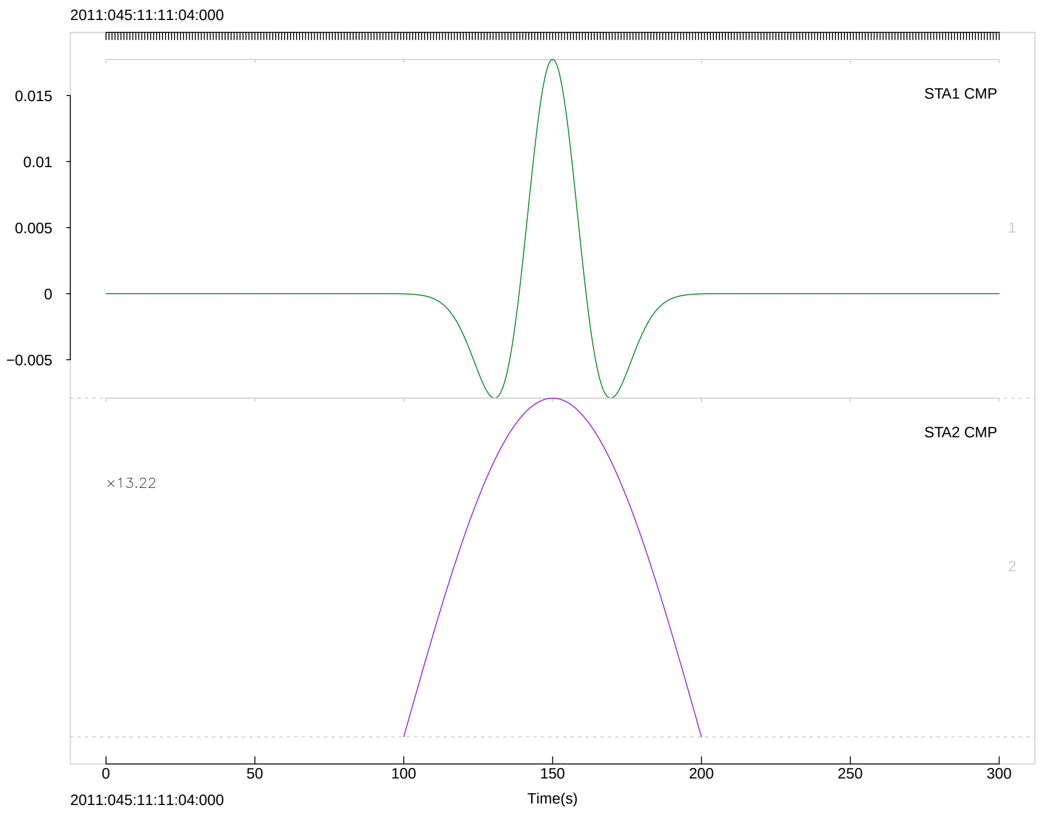

Figure 3.5: swig example from prepSEIS

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.

3.7.1 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)
```

3.7.2 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.

3.8 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.

3.9 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.

3.10 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.

3.10.1 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 = '/home/lees/Site/Santiaguito/SG09'
pattern = "R0*"

###   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)
```

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()
```

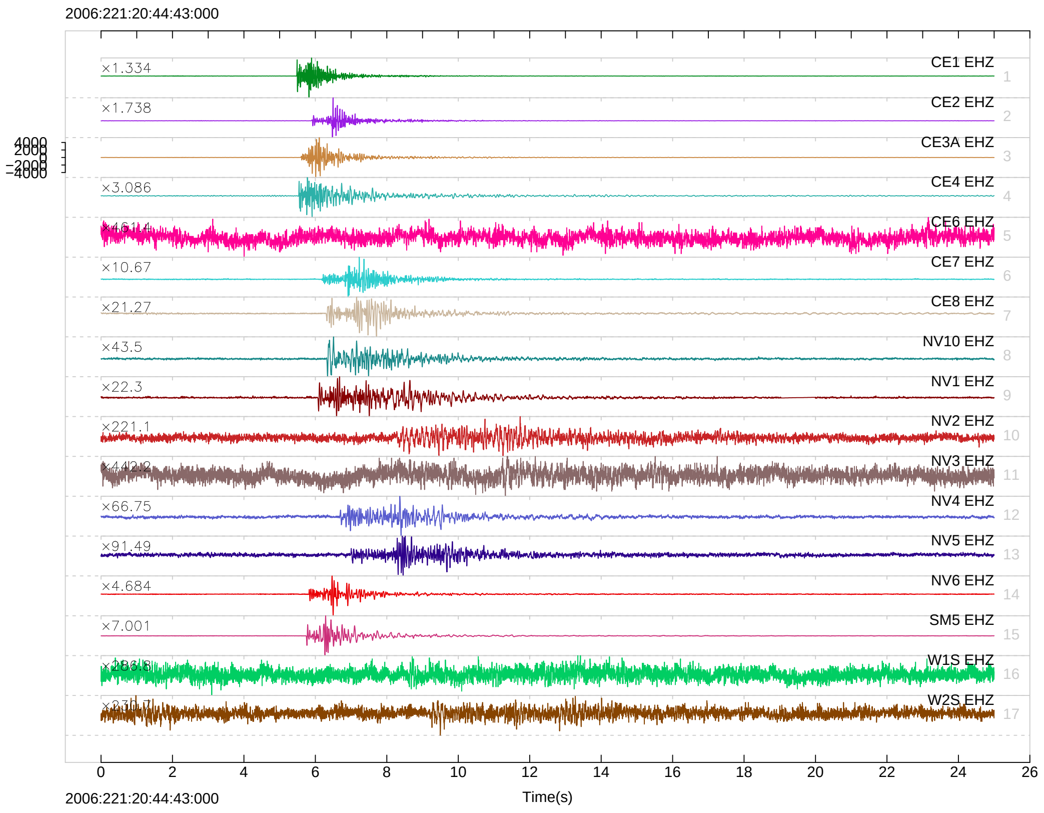

Figure 3.6: Swig Example Plot

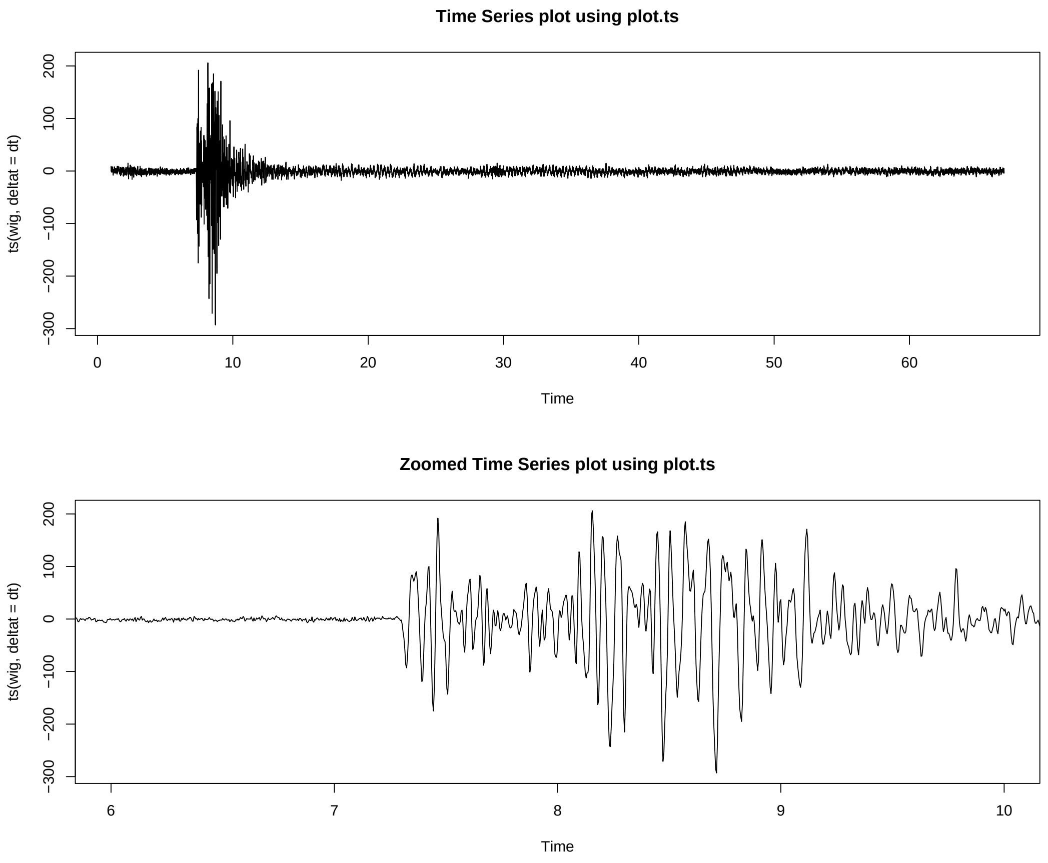

Figure 3.7: Swig Example Plot

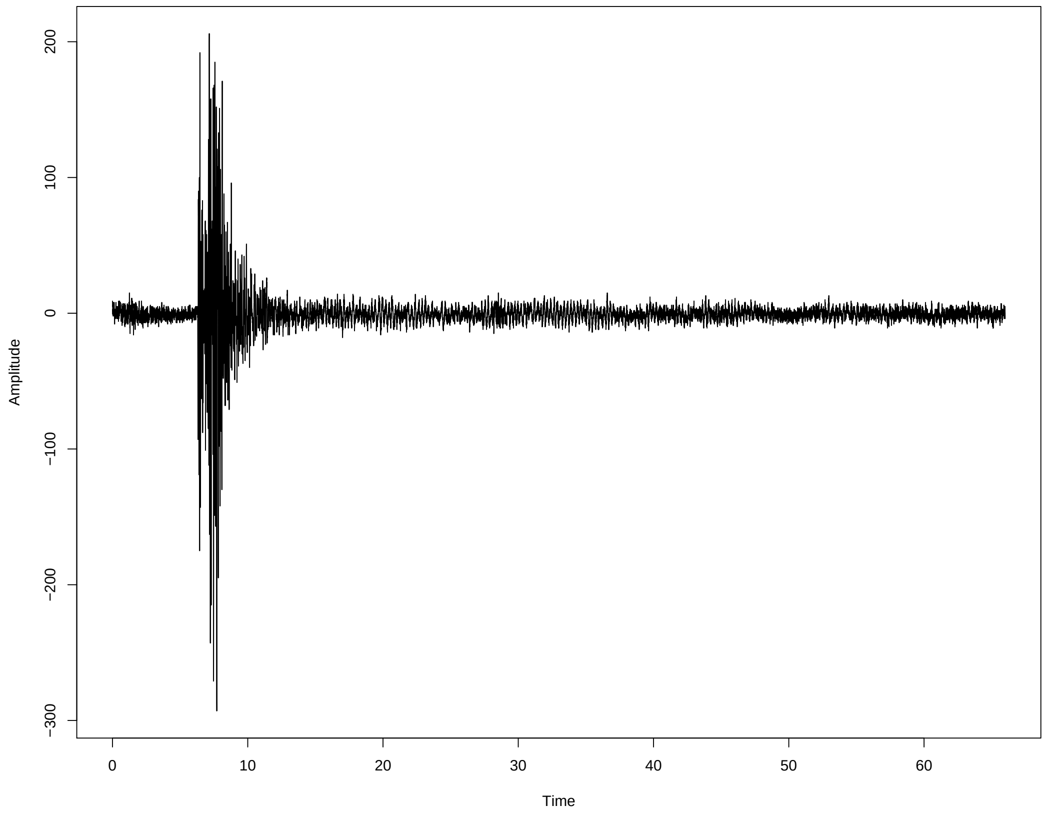

Figure 3.8: Create a wiggle and plot

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()
```

3.11 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 3.9).

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 3.10. .

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 3.11.

```
dev.off()
```

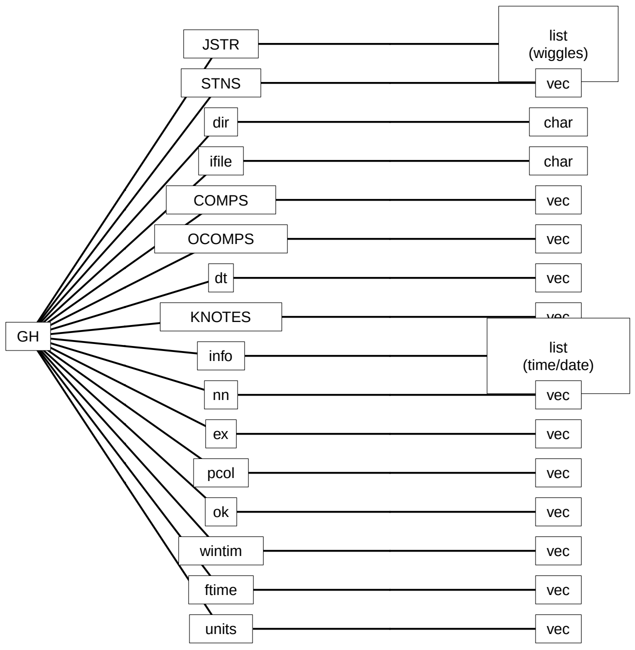

Figure 3.9: Documentation for RSEIS seismic data list

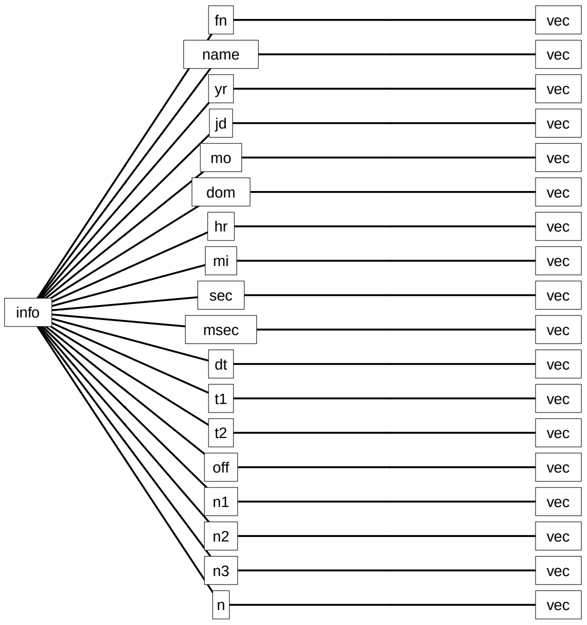

Figure 3.10: Documentation for info list

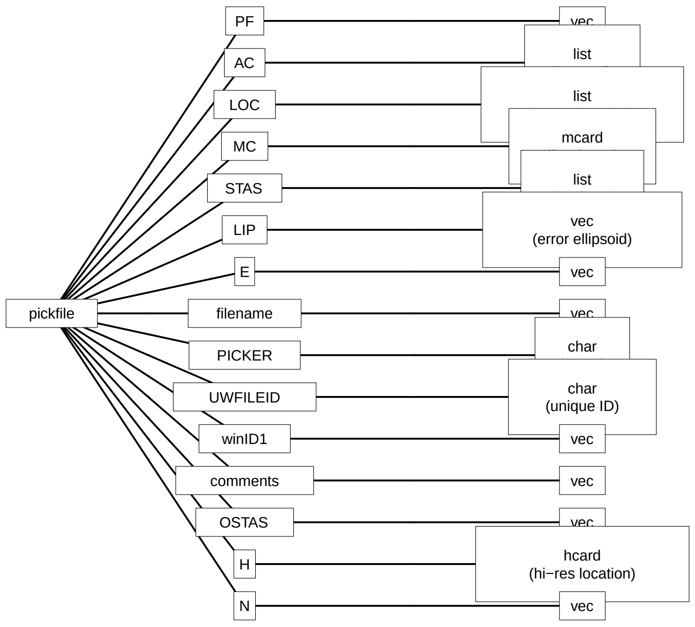

Figure 3.11: Documentation for pickfile list

Chapter 4

RQUAKE

4.1 RQUAKE

4.2 Introduction

Earthquake location is a nonlinear process: Usually it is accomplished by performing a sequence of linear inversions to converge to a location that minimizes the residual misfit. Event determination requires several pieces of information to successfully locate an earthquake: observed arrival time estimates, station locations, a velocity model and, in some cases, a set of station corrections.

Station locations should be provided in terms of Name, Latitude, Longitude and Depth. In some cases several stations may be located at the same location but at different depths (elevation) so these are usually distinguished by some naming convention. Location programs may involve projection of the datainto cartesion coordinates or, in some programs, the processing is done with lat-lon pairs instead. Usually the station locations are stored in a file on disk, as a table or in some other form that can be accessed and stored in memory.

The velocity model is commonly a one-dimensional, layered model, although recently three-dimnesional models are used. Since the location procedure must estimate the travel time from the event to the stations ray-tracing or some other method must be used to make the travel time predictions. In the case of one-dimensional velocity model estimating travel times is relatively simple and the RSEIS package provides routines to accomplish this. The velocity model is typically a table with depth and associated P-wave velocity for a given layer. Often S-wave velocities are provided, although if they are not available a standard approximation is $V_s = V_p/\sqrt{3}$ which is known as a *Poisson* solid. Package Rquake has several built in velocity models that can be used initially if a derived one is not available.

Phase arrival times are typically determined from displays of seismic data. Essentially all that is needed is the arrival time, in seconds relative to some origin of each station for each phase. Software in RSEIS can be used to extract arrival times. Additional information may include the polarity of the P-wave arrival used for focal mechanism determination.

4.3 Rquake

In this document I will illustrate how to Rquake, a non-linear earthquake location program.

4.4 Data Structures and Lists

4.4.1 Station File

Station location information can be stored in memory (in a list) or in a text file on disk. The station file is a table, with name, lat, lon, and elevation.

For example:

```
fsta = "/home/lees/Mss/SEIS_BOOK/RQUAKE/data/staLLZ.txt"
###  system(paste(sep=" ", "cat", fsta), intern = TRUE )
```

```
CHAC0   -0.39377412    -78.15369741 3588
CHAC1   -0.366526404   -78.16962049  3606
CHAC2   -0.42485567    -78.2710065   4020
CHAC3   -0.4524493    -78.18676153    4328
CHAC4   -0.461317213   -78.21783387   4412
CHAC5   -0.351938598   -78.21809574   4000
CHAC6   -0.408928292   -78.20667762    3860
CHAC7   -0.39837847    -78.22075601    4109
```

```
CHAC8   -0.382639731  -78.2023599    3767
CHAC9   -0.323852103  -78.15061344  3762
```

These can be scanned in **R** with a simple command.

See **REIS** for more details on stations.

If the stations are in UTM coordinates, you may convert to Lat-Lon using the GEOmap package.

```
  stas = scan(file=fsta,what=list(name="", lat=0, lon=0, z=0))
  stas$z = stas$z/1000
```

Units in Rquake are in km, so the meters are converted.

REIS has a function for reading in the stations:

```
stas = setstas("stas")
```

4.4.2 Velocity Structure

The one-dimensional velocity model is also stored in file (or stored in memory in an **R** session). See **REIS** for details.

Sample velocity model stored on disk. In this case no estimates of error are provided, so they are set to zero. If S-wave velocity is not available, can use $V_s = V_p/\sqrt{3}$.

```
#MODEL WU COSO REGINAL FINE LAYERS REGIONAL VELOCITY MODEL
#P DEPTH   P VEL      PERR      S DEPTH    S VEL    SERR
   0.00     4.50      0.00      0.00        2.43    0.00
   0.50     4.51      0.00      0.50        2.59    0.00
   1.00     4.92      0.00      1.00        2.97    0.00
   1.50     4.92      0.00      1.50        2.97    0.00
   2.00     5.46      0.00      2.00        3.15    0.00
```

```
  2.50     5.46     0.00     2.50        3.15    0.00
  3.00     5.54     0.00     3.00        3.27    0.00
  3.50     5.54     0.00     3.50        3.27    0.00
  4.00     5.58     0.00     4.00        3.42    0.00
  5.50     5.58     0.00     5.50        3.42    0.00
 12.00     6.05     0.00    12.00        3.49    0.00
 20.00     7.20     0.00    20.00        4.15    0.00
```

The following is a constructor for making a 1D velocity model suitable for use in RSEIS and Rquake:

```
VEL=list()
    VEL$'zp'=c(0,0.25,0.5,0.75,1,2,4,5,10,12)
    VEL$'vp'=c(1.1,2.15,3.2,4.25,5.3,6.25,6.7,6.9,7,7.2)
    VEL$'ep'=c(0,0,0,0,0,0,0,0,0,0)
    VEL$'zs'=c(0,0.25,0.5,0.75,1,2,4,5,10,12)
    VEL$'vs'=c(0.62,1.21,1.8,2.39,2.98,3.51,3.76,3.88,3.93,4.04)
    VEL$'es'=c(0,0,0,0,0,0,0,0,0,0)
    VEL$'name'='/data/wadati/lees/Site/Hengil/krafla.vel'
```

There are several default velocity models available in **REIS** . Function defaultVEL(i) will return one of 6 “standard” models used for different purposes.

If you have a velocity model on disk, you can read it in with **REIS** function, Get1Dvel.

To compare a set of different velocity models visually, try,

```
data(ASW.vel)
     data(wu_coso.vel)
     data(fuj1.vel)
     data(LITHOS.vel)
     Comp1Dvels(c("ASW.vel","wu_coso.vel",  "fuj1.vel" , "LITHOS.vel"  ))



dev.off()
```

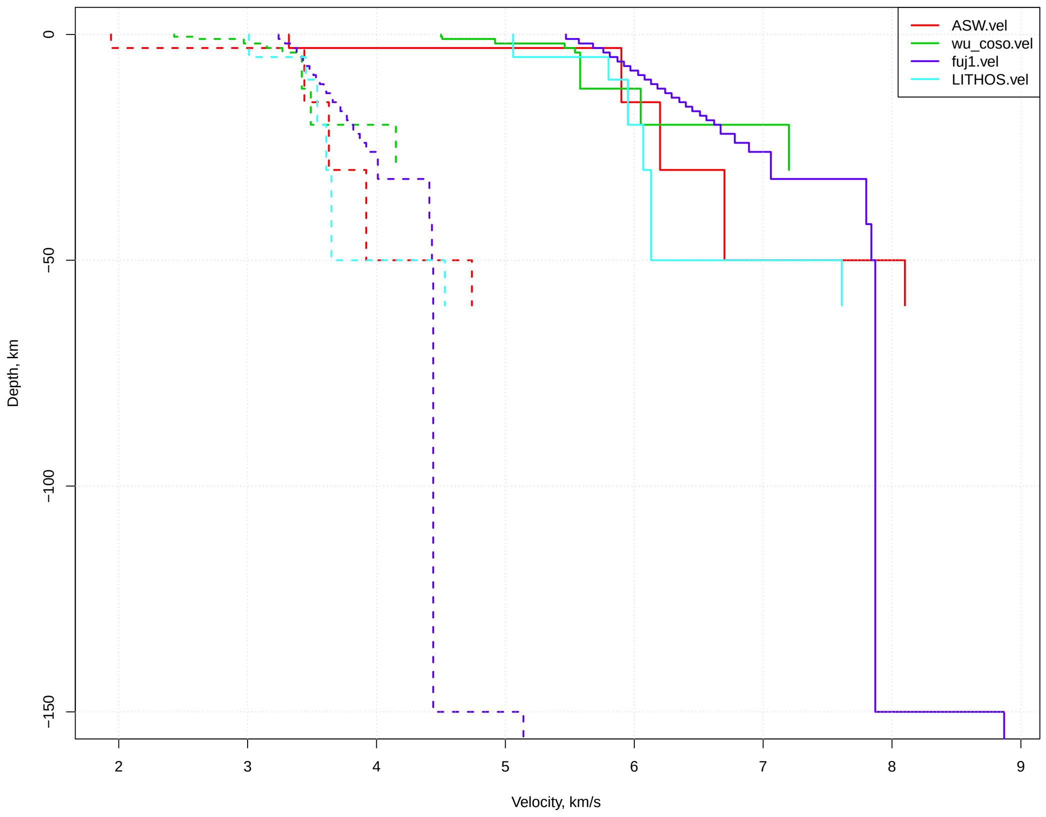

Figure 4.1: swig example with Reventador Data

4.4.3 Arrival Time List

At the most basic level, all that is required to estimate a hypocenter is relative time of arrival at each station. The arrival times, or the picks are stored in in list mode, i.e. a list of vectors each with attributes relating to the arrival time pick.

These vectors are described as:

tag character tag the should be unique

name character, station name

comp character, component name

c3 character, three-component station id sta.hhh.BHZ

phase character, phase name

err numeric, error

pol character polarity, U, D, 0

flg numeric, flag, used in location

res numeric, travel time residual relative to model

dur numeric, duration

yr numeric, year

mo numeric, month

dom numeric, day-of-month

jd numeric, julian day

hr numeric, hour

mi numeric, minute

sec numeric, second

col numeric, or character, color for plotting in RSEIS

onoff numeric, less than 0 means do not use

A constructor for creating an empty pick list is cleanWPX. For many of the functions in RSEIS and Rquake the list must contain filled vectors for each element. use function repairWPX to fill out list elements that are deficient.

The arrival time list has one attribute, the “ID”. This can be used to identify earthquake with a unique tag or identifaication number or name.

For **Rquake** , the elements that are absolutely *required* are: name, phase, err, sec. One can construct the input list from these elements by putting in arbitrary information for the other pasts of the list. Once the event and relative time shift is estiamted, these can be added or subtracted from guess origin time.

There are many different ways to store arrival time picks. It does not matter how these are stored, as long as they are read into R and formatted properly. By disassociating the input format from the analysis, we can simply write a short input, or conversion, routine to use all the codes as is.

We can thus store the data in any format we desire, perhaps for use in other non-R software.

Native (binary) R

The output of **swig** is binary R file, so the data can simply be loaded automatically.

UW format Pickfiles

loadUWpickfiles is a function that reads in a list of pickfiles stored on disk and returns a list of picked events.

Since UW pickfiles store the times relative to a common minute mark, and station information is not stored in the pickfile, this information is filled out in the code:

```
KF = vector(mode="list")
   for(i in 1:length(LF))
     {
       g1 = getpfile(LF[i])
       m1 = match(g1$STAS$name, stas$name)
       g1$STAS$lat = stas$lat[m1]
```

```
        g1$STAS$lon = stas$lon[m1]
        g1$STAS$z = stas$z[m1]
        w1 = which(!is.na(g1$STAS$lat))
        sec = g1$STAS$sec[w1]
        N = length(sec)

        Ldat = list(name = g1$STAS$name[w1],
          sec = g1$STAS$sec[w1],
          phase = g1$STAS$phase[w1],
          lat = g1$STAS$lat[w1],
          lon = g1$STAS$lon[w1],
          z = g1$STAS$z[w1],
          err = g1$STAS$err[w1],
          yr = rep(g1$LOC$yr, times = N),
          jd = rep(g1$LOC$jd, times = N),
          mo = rep(g1$LOC$mo, times = N),
          dom = rep(g1$LOC$dom, times = N),
          hr = rep(g1$LOC$hr, times = N),
          mi = rep(g1$LOC$mi, times = N))

        Ldat$err[Ldat$err <= 0] = 0.05
        Ksta = length(unique(Ldat$name))
       ###  cat(paste("#################      ", i, Ksta), sep = "\n")
        Ldat = LeftjustTime(Ldat)

        KF[[i]] = Ldat

      }
```

CSV Pickfiles

An example comma-separated-value file (csv), might look like this:

```
> cat 2011_11_21_12_14_20_683433.csv
"","tag","name","comp","c3","phase","err","pol","flg","res","dur","yr","mo","dom","jd","hr","mi","
"1","CHAC1","CHAC1","V",0,"G",0,"_",0,0,0,2011,11,21,325,12,14,20.6834335327148,"#0000FF",1
"2","CHAC2","CHAC2","V",0,"G",0,"_",0,0,0,2011,11,21,325,12,14,50.691351890564,"#0000FF",1
"3","CHAC6","CHAC6","V",0,"G",0,"_",0,0,0,2011,11,21,325,12,15,14.6926865577698,"#0000FF",1
"4","CHAC8","CHAC8","V",0,"G",0,"_",0,0,0,2011,11,21,325,12,15,44.7056050300598,"#0000FF",1
```

4.5 Example

Suppose you have a set of arrival at the stations of a network.

```
H = read.csv(file='/home/lees/Mss/SEIS_BOOK/RQUAKE/data/2011_11_21_12_14_20_683433.csv')
```

Chapter 5

Filter and Decon

5.1 Filter

5.2 Signal Package

There is a package in R called signal that replicates the functionality of the signal processing toolbox in MATLAB. There are many features in the signal package that are useful and can be applied in slightly different ways than the implementation presented in RSEIS.

For example, consider the creation and application of a butterworth filter. In signal,

Figure 5.1:

```
library(RSEIS)
library(signal)
    bf <- butter(3, 0.1)                        # 10 Hz low-pass filter
     t <- seq(0, 1, len = 100)                  # 1 second sample
     x <- sin(2*pi*t*2.3) + 0.25*rnorm(length(t))# 2.3 Hz sinusoid+noise
     y <- filtfilt(bf, x)
     z <- filter(bf, x) # apply filter
 zz = butfilt(x, fl=0, fh=10,  deltat=1/100,  type="LP" ,  proto="BU")
     plot(t, x, type='l')
     lines(t, y, col="red")
     lines(t, z, col="blue")
   lines(t, zz, col="purple")
   legend("bottomleft", legend = c("data", "filtfilt", "filter", "butfilt"),
            pch = 1, col = c("black", "red", "blue", "purple"), bty = "n")
```

Figure 5.2:

```
library(signal)
    bf <- butter(2, 0.1)                        # 10 Hz low-pass filter
     t <- seq(0, 1, len = 100)                   # 1 second sample
     x <- rep(0, times=length(t))
     x[floor(length(x)/2)] = 1
     y <- filtfilt(bf, x)
     z <- filter(bf, x) # apply filter
 zz = butfilt(x, fl=0, fh=10,  deltat=1/100,  type="LP" ,  proto="BU", npoles=2)
     plot(t, x, type='l')
     lines(t, y, col="red")
     lines(t, z, col="blue")
   lines(t, zz, col="purple")
   legend("bottomleft", legend = c("data", "filtfilt", "filter", "butfilt"),
            pch = 1, col = c("black", "red", "blue", "purple"), bty = "n")
```

Figure 5.3:

```
library(signal)
                    # 10 Hz low-pass filter
     t <- seq(0, 1, len = 100)                   # 1 second sample
     x <- rep(0, times=length(t))
     x[floor(length(x)/2)] = 1
         bf2 <- butter(2, 0.1)
y2 <- filtfilt(bf2, x)
         bf3 <- butter(3, 0.1)
y3 <- filtfilt(bf3, x)
         bf4 <- butter(4, 0.1)
```

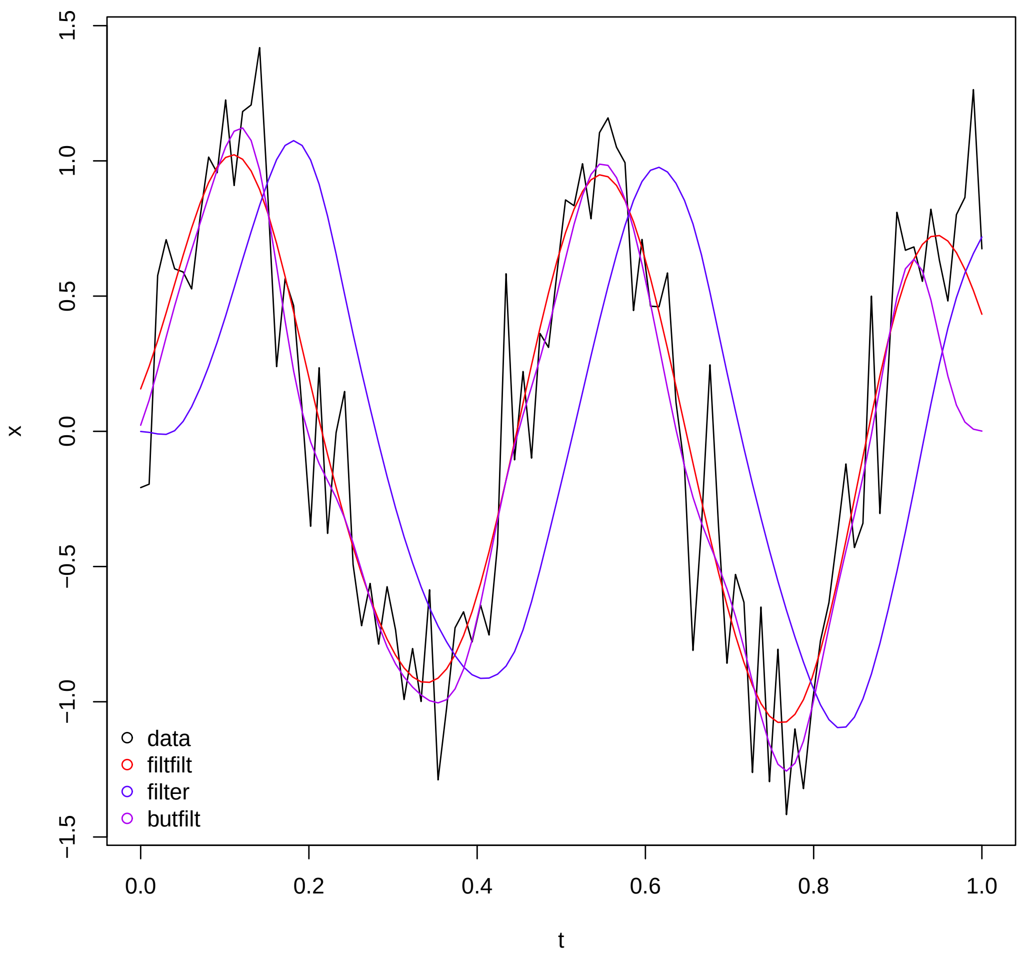

Figure 5.1: Butterworth filter applied with filtfilt, filter and butfilt.

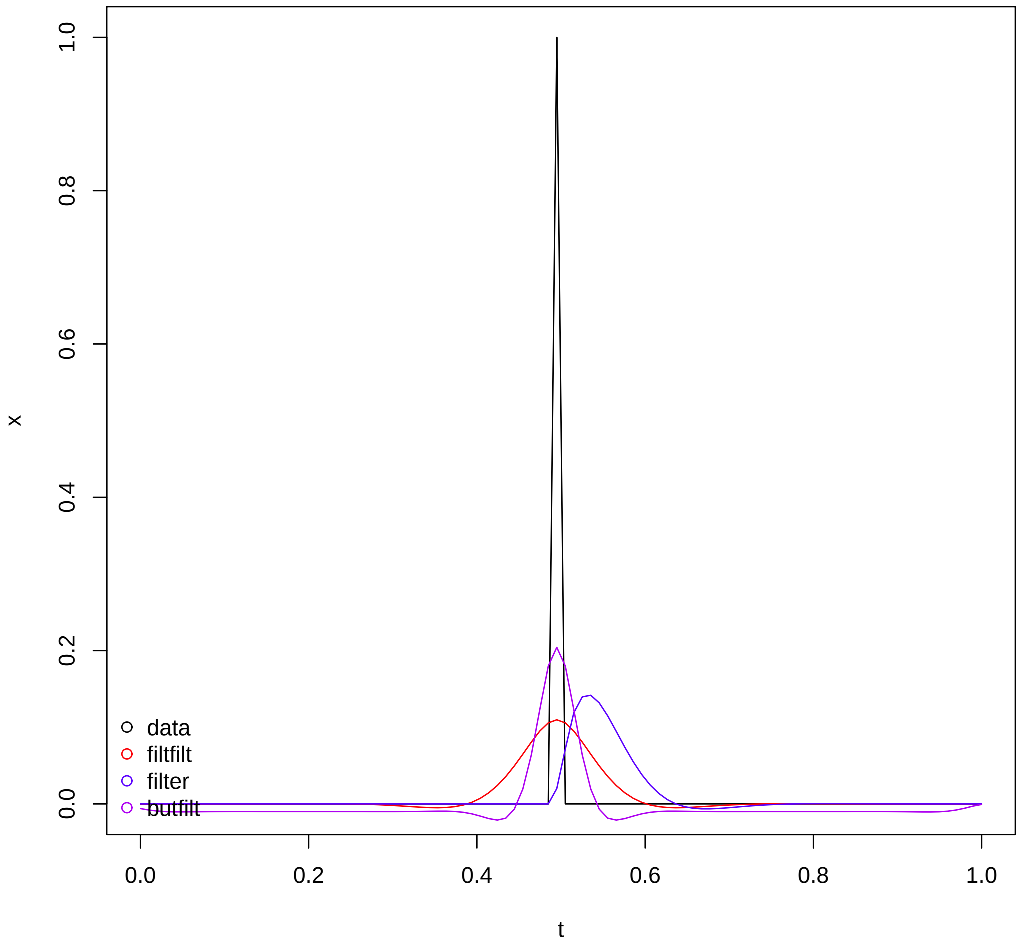

Figure 5.2: Butterworth filter applied to an impulse: Impulse response functions.

```
y4 <- filtfilt(bf4, x)
     plot(t, x, type='l')
     lines(t, y2, col="red")
     lines(t, y3, col="blue")
   lines(t, y4, col="purple")
   legend("bottomleft", legend = c("data", "2-poles", "3-poles", "4-poles"),
            pch = 1, col = c("black", "red", "blue", "purple"), bty = "n")
```

Figure 5.3:

```
library(signal)
                    # 10 Hz low-pass filter
     t <- seq(0, 1, len = 100)                   # 1 second sample
     x <- rep(0, times=length(t))
     x[(floor(0.25*length(x))):(floor(0.75*length(x)))] = 1
         bf2 <- butter(2, 0.1)
y2 <- filtfilt(bf2, x)
         bf3 <- butter(3, 0.1)
y3 <- filtfilt(bf3, x)
         bf4 <- butter(10, 0.1)
y4 <- filtfilt(bf4, x)
     plot(range(t), range(x, y2, y3, y4) , type='n')
lines(t, x, type='l')
     lines(t, y2, col="red")
     lines(t, y3, col="blue")
   lines(t, y4, col="purple")
   legend("bottomleft", legend = c("data", "2-poles", "3-poles", "4-poles"),
            pch = 1, col = c("black", "red", "blue", "purple"), bty = "n")
```

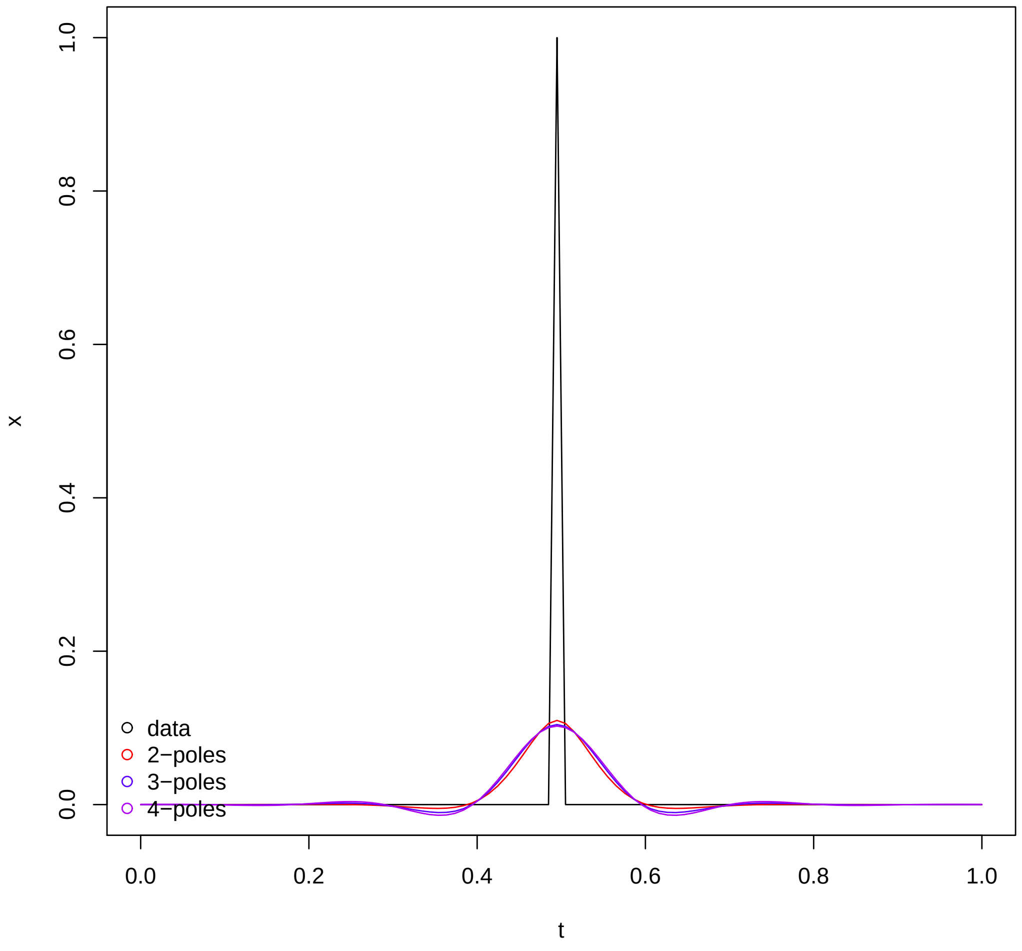

Figure 5.3: Butterworth filter applied to an impulse: Impulse response functions.

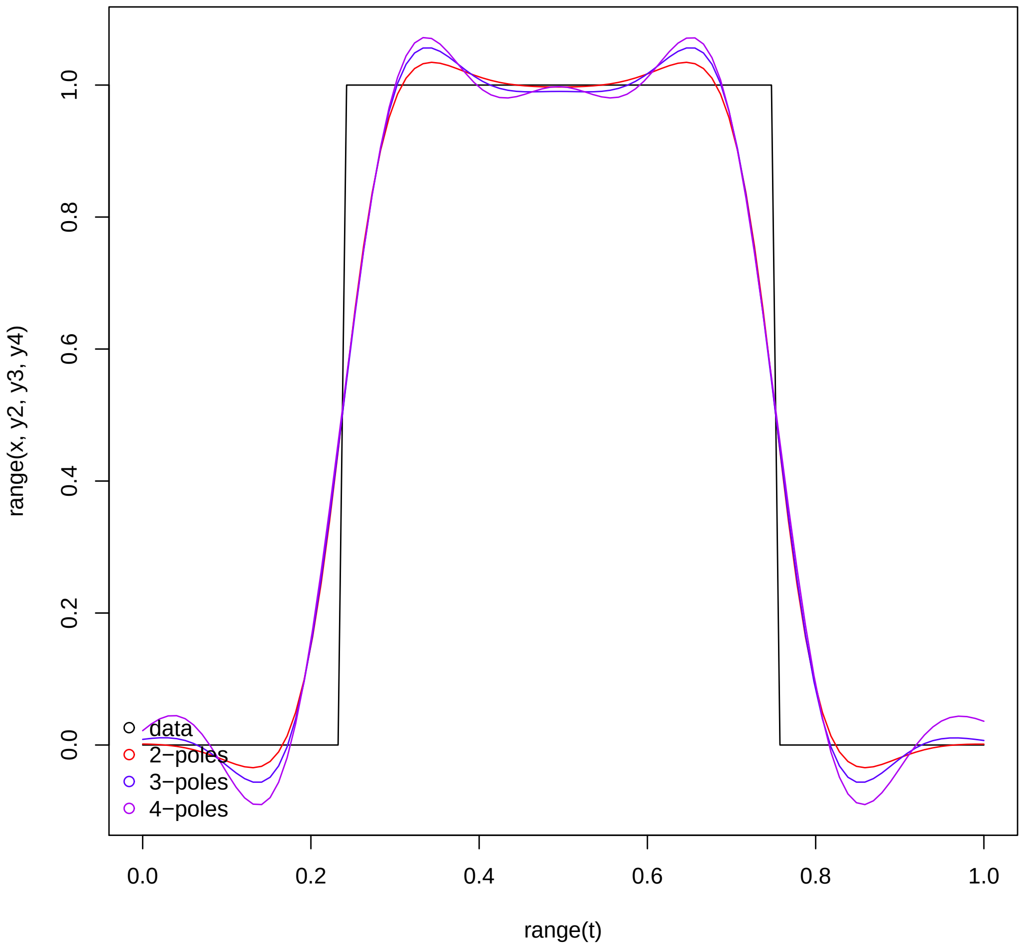

Figure 5.4: Butterworth filter applied to an impulse: Impulse response functions.

Chapter 6

Focal Mechanisms

6.1 Focal Mech 1

6.1.1 Directional Data: Circles

Statistics using circular or directional data

Since data in the earth sciences often involve spatially distributed information on the directionality of a particular property it is important for researchers in the geological sciences to have a firm grasp on the differences these data have with other scalar data. Advanced treatments of circular statistics can be found in [**?**] and [**?**].

Directions in the earth sciences arise in analysis of spatially oriented observations such as fault strikes, striations, mineral deformation or other data concerned with bearing. It is common for the uninitiated to make the mistake that angles can be treated like other Cartesian parameters. Angles are commonly recorded as data points in units of degrees, although these are not very useful for typical calculation. As a first step angles should transformed into radians using the conversion factor $\pi/180$ radians/degree. Once angles have been converted, they can be transformed to Cartesian coordinates using

$$\begin{aligned} x &= A\cos(\alpha) \\ y &= A\sin(\alpha) \end{aligned} \tag{6.1.1}$$

where the angle α increases from the x (bottom) axis counter-clockwise and A is the radius of the circle, here set to one. In many geological situations, however, angles are measured from the North clockwise. In that case, given data in degrees, we must reverse the direction and rotate by 90°.

$$\begin{aligned} x &= A\cos(90-\alpha^\circ) \\ y &= A\sin(90-\alpha^\circ) \end{aligned} \tag{6.1.2}$$

Again, angles presented in degrees and must be converted to radians prior to using trigonometric functions in R.

For example, if we have a set of angles a, provided as an example in Mardia(1972, p.22),

```
A = 1
##  a=c(41.9,31.7,48.2,42.4,32.8,36.0,28.6,33.2,34.3,32.5)
###   data from Mardia(1972) p 22
a = c(43,45,52,61,75,88,88,279,357)
x = A*cos((a)*pi/180)
y = A*sin((a)*pi/180)
n = length(a)
Mx = mean(x)
My = mean(y)
Mdir = (atan2(My, Mx))*180/pi
Rbar = sqrt(Mx^2+ My^2)
##  draw a circle
i=pi*(0:360)/180
cx =  A*cos(i);
cy =  A*sin(i);
```

which can be plotted with the code:

```
plot(cx, cy, type='n', asp=1, ann=FALSE, axes=FALSE)
lines(cx,cy)
points(x,y)
segments(0,0, x,y, col=grey(.7))
arrows(0,0, Rbar*cos(Mdir*pi/180  ) , Rbar*sin(Mdir*pi/180  ), col=grey(0))
```

For directional data a summary direction can be extracted by adding the x and y coordinates because they are Cartesian. If the directions are uniformly scattered, the resultant direction will be small. On the other hand if the directions cluster in a dominant orientation, the summed vectors will add constructively and the resultant will be large. As an example we use the data for a set of glacial striation directions from Finland (Davis, 2003, p.317). The average direction will be oriented towards the mean x and y values.

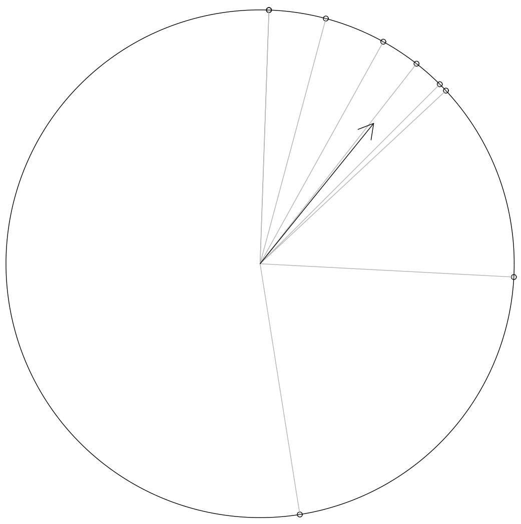

Figure 6.1: Circular plot of directions for FINLAND data set

```
a=c(23,27,53,58,64,83,85,88,93,99,100,105,113,113,114,117,121,123,125,
126,126,126,127,127,128,128,129,132,132,132,134,135,137,144,145,145,146,153,155,155,
155,157,163,165,171,172,179,181,186,190,212)
x = A*cos((90-a)*pi/180)
y = A*sin((90-a)*pi/180)
Mx = mean(x)
My = mean(y)
Mdir = (pi/2-atan2(My, Mx))*180/pi
```

The mean direction of this data is thus α =129°. This can be added simply to the plot of the original data,

```
plot(cx, cy, type='n', asp=1, ann=FALSE, axes=FALSE)
lines(cx,cy)
points(x,y)
segments(0,0, x,y, col=grey(.8))
arrows(0, 0, Mx, My, lwd=2)
```

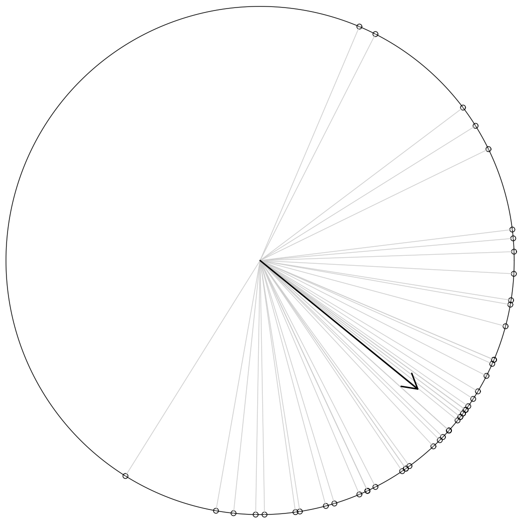

Figure 6.2: FINLAND data set with mean direction shown

The mean resultant length is often denoted $\bar{R}$ and is used in a variety of quantitative measures of the circular data set. for example, the circular variance is estimated by

$$S = 1 - \frac{R}{n} = 1 - \bar{R} \tag{6.1.3}$$

which in our case is,

```
R = sqrt(sum(x)^2+sum(y)^2)
Rbar = sqrt(Mx^2+ My^2)
S = 1 - Rbar
```

To perform hypothesis testing with circular data we use the von Mises distribution, or the circular normal distribution. This distribution is characterized by two parameters, κ and μ and is given by,

$$f(x \mid \mu, \kappa) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)} \tag{6.1.4}$$

To illustrate this distribution with different levels of compaction κ we can generate distributions using R,

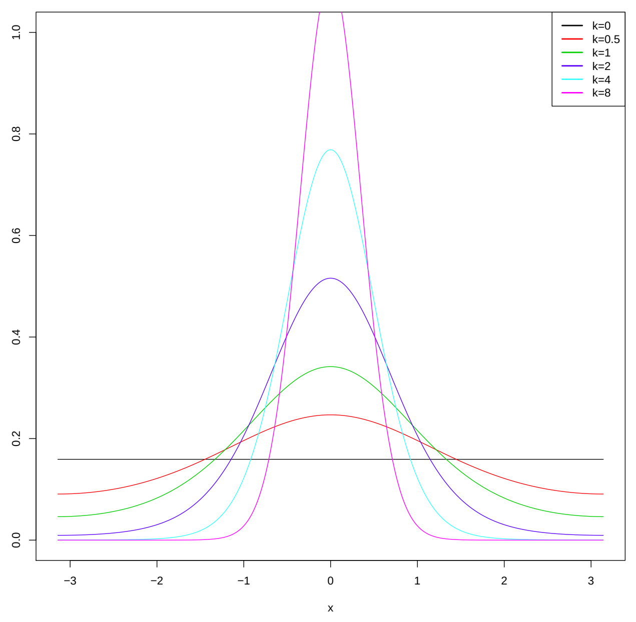

Figure 6.3: Probability Density Function for von Mises Distribution

The CRAN distribution site has a package called CircStats that has many of the functions used in analyzing circular data. Given a calculated $\bar{R}$ value one can get an estimate of the κ by loading the CircStats package and calling the est.kappa function.

```
library(CircStats)
kappa = est.kappa(a*pi/180)
mu = Mdir*pi/180
```

If the data were extracted from a uniform distribution we would expect the value of $\bar{R}$ to be small. A test due to Lord Rayleigh can be accomplished be comparing $\bar{R}$ to critical values for different P-values. A table of P-values can be found in Davis (Table A10) or Mardia (Appendix 2.5). Here we provide an approximation using a method devised by Wilkie(1983). The table data are recreated using R:

```
pees = c(0.1, 0.05, 0.025, 0.01)
n = c(4:25, 30, 35, 40, 45, 50)
On = 1/n
TAB = matrix(ncol=length(pees)+1, nrow=length(n))
TAB[,1] = n
```

```
for(i in 1:length(pees))
{
P = pees[i]
K = -log(P) - ( (2*log(P) + (log(P)^2) )/(4*n))
TAB[,i+1] = sqrt(K*On)
}
colnames(TAB)<-c("n",pees)
print(TAB)
```

```
       n       0.1      0.05     0.025      0.01
 [1,]  4 0.7515051 0.8380480 0.9082263 0.9817512
 [2,]  5 0.6734610 0.7545310 0.8218730 0.8950204
 [3,]  6 0.6155695 0.6917903 0.7560085 0.8271724
 [4,]  7 0.5704262 0.6424505 0.7037021 0.7724448
 [5,]  8 0.5339490 0.6023408 0.6608886 0.7271740
 [6,]  9 0.5036789 0.5689047 0.6250175 0.6889536
 [7,] 10 0.4780342 0.5404774 0.5944012 0.6561432
 [8,] 11 0.4559456 0.5159227 0.5678738 0.6275859
 [9,] 12 0.4366606 0.4944347 0.5446013 0.6024414
[10,] 13 0.4196323 0.4754243 0.5239693 0.5800831
[11,] 14 0.4044523 0.4584497 0.5055145 0.5600340
[12,] 15 0.3908088 0.4431716 0.4888793 0.5419235
[13,] 16 0.3784589 0.4293253 0.4737836 0.5254589
[14,] 17 0.3672104 0.4167003 0.4600039 0.5104058
[15,] 18 0.3569087 0.4051270 0.4473596 0.4965739
[16,] 19 0.3474280 0.3944672 0.4357030 0.4838068
[17,] 20 0.3386647 0.3846066 0.4249120 0.4719748
[18,] 21 0.3305328 0.3754503 0.4148845 0.4609693
[19,] 22 0.3229599 0.3669181 0.4055346 0.4506983
[20,] 23 0.3158847 0.3589422 0.3967894 0.4410839
[21,] 24 0.3092550 0.3514648 0.3885863 0.4320590
[22,] 25 0.3030259 0.3444359 0.3808716 0.4235657
[23,] 30 0.2766935 0.3146890 0.3481840 0.3875217
[24,] 35 0.2562147 0.2915194 0.3226842 0.3593437
[25,] 40 0.2396993 0.2728135 0.3020737 0.3365333
[26,] 45 0.2260145 0.2573005 0.2849666 0.3175783
[27,] 50 0.2144342 0.2441646 0.2704710 0.3015024
```

Once a significance level has been chosen, it can be compared to values in the table to determine the critical points for rejection. As shown in Davis, since $\bar{R}$ =0.8 is greater than the critical value from the table $\bar{R}_{50\,5\%}$ =0.244. Thus at 5% significance we reject the null hypothesis that the concentration parameter is zero and conclude that the data are not uniformly distributed.

Rose Diagrams

Rose diagrams are often used to represent graphically circular distributions of data in a way that is analogous to histograms. Serious problems can arise resulting in misleading readers if rose diagrams are done improperly. If the petals of a rose diagram are plotted with radii proportional to the *number* of samples in the bin, as opposed to the *area* of the petal plotted proportional to the number or percentage of the elements between two angles, the rose diagram will appear biased, or weighted improperly. To illustrate this we use a set of data from Davis called FINLAND.txt using the program rose.R supplied in the appendix.

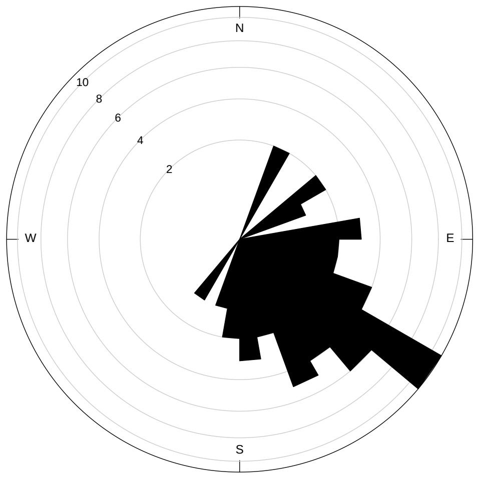

Figure 6.4: Simple Rose Diagram

Note that the scale is not linear. In the example and code provided here it is worthwhile noting that the rose diagram can be scaled and plotted anywhere on an existing plot made in R. For example, if a map is plotted and circular rose diagrams are used to illustrate the spatial variation of orientations it is easy to create a complex plot of both map and roses together.

Here one can easily see the spatial variation of directional data across a field site. For each location a statistical test can be performed and confidence can be used to modify the roses by color or other method.

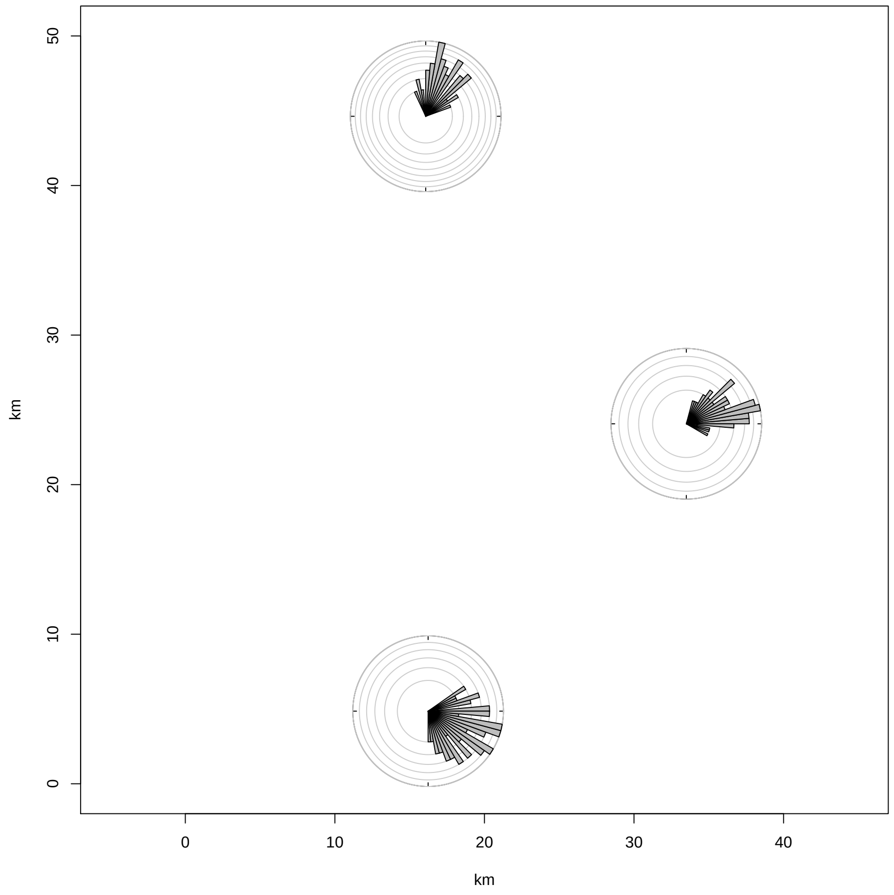

Figure 6.5: Geographic distribution of Rose diagrams

6.2 Focal Mech 2

6.2.1 Data distributed on Spheres

Analysis of data on the sphere is critical to geological sciences for an obvious reason: the Earth can be approximated simply as a sphere to first order. Beyond this important motivation it spheres are used in geological sciences to represent the distribution of three-dimensional orientations and directions, the distribution of planar surfaces and, of course, as simplification of tensors (focal mechanisms).

This chapter includes analysis of upper and lower hemispheric projections of fault information, focal mechanisms, stress distributions and associated statistical tests. Examples from paleomagnetic data will be presented and rotations on the sphere will be discussed. This chapter ties in well with the section on Matrices. An excellent advanced book on this topic is [**?**] and [**?**].

6.2.2 Spherical Coordinate Systems

Geometry on the sphere is critical for Earth Science data analysis for obvious reasons, namely that the Earth is approximated to first order by a sphere. Locations on the planet are referenced via latitude (Lat) and longitude (Lon), and calculations related to positioning or distribution of properties on the globe are essential. Furthermore, many applications involving spherical distributions are common in structural geology and geophysics, where faults are represented as poles on and earthquake ruptures are modeled as double couple focal mechanisms using spherical relations between fault and auxiliary planes.

We first establish the relationship between Cartesian coordinate system (x, y, z) and spherical coordinate system (r, ϕ, θ). For many applications the mathematical definition stems from taking the x-axis on the page to the right, the y-axis is up, and the z-axis is out of the page. In that case r is the radius of the vector, ϕ is the angle from the zenith along the z-axis, and θ is the angle of the projection of the vector in the x-y plane with the x axis. The relationship between the Cartesian and the spherical coordinates is expressed as a transformation,

$$\begin{aligned} x &= r\cos(\theta)\sin(\phi) \\ y &= r\sin(\theta)\sin(\phi) \\ z &= r\cos(\phi) \end{aligned} \tag{6.2.1}$$

where the inverse relationships are,

$$\begin{aligned} r &= \sqrt{x^2+y^2+z^2} \\ \phi &= \cos^{-1}(z/r) \\ \theta &= \tan^{-1}(y/x) \end{aligned} \tag{6.2.2}$$

These are illustrated in the following diagram where the angles and the Cartesian axes are laid out. As noted earlier, it is conventional in geographical applications to use North facing up on the page. In that case the θ should be adjusted to represent increasing angle clockwise, or $\theta\prime = 90 - \theta$ when θ is in degrees. Also, the angle ϕ is co-latitude and if latitude is given it must be converted similarly, $\phi\prime = 90 - \phi$.

Once these relations are established and coded properly, converting coordinate systems in R is easy. Given a set of LAT-LON pairs, one converts to Cartesian coordinates, performs a calculation and converts back for further analysis. we saw this earlier in chapter 4 when we found the distance between Chicago and Paris using the cross product of two Cartesian vectors.

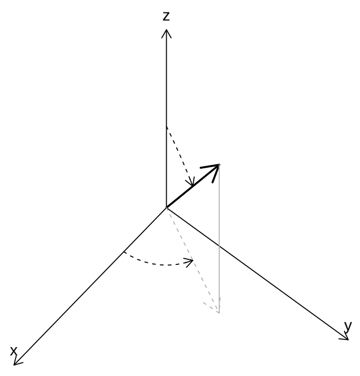

Figure 6.6: Coordinate system for geographic vectors and angles

6.2.3 Fisher Statistics

Data distributed on a sphere is quite common in Earth science. Problems that deal with fault distributions, focal mechanisms, and paleomagnetic poles are just a few examples. It is useful for a researcher in geology to have at hand a set of tools ready for attacking such problem with ease and flexibility. While canned programs exist that allow users to dump in data sets, plot see summary statistics it is much more beneficial to be able to access data sets in minute detail while also having higher level graphical and analytical tools. The R platform is excellent for achieving this goal. As an example consider a set of paleomagentic poles observed from laboratory measurements of rocks collected across a large continental region. The research needs to know what the mean orientation of the magnetic poles are to see if the continent has rotated and move large distances in the north south direction.

From a set of N measurements the researcher has pairs of inclination and declination estimated in the lab. The first step is to plot these on a stereo net, either equal area (Schmidt net) or equal angle (Wulff net). There is no set function in R available for this but contributed packages have already solved this problem.

For a Schmidt net we load the RFOC package and call a function,

```
JPOST(file="/home/lees/Mss/SEIS_BOOK/Fmech/FIGS/nets.eps" , width=10, height=6)
library('RFOC')
par(mfrow=c(1,2))
net()
title(sub='Schmidt Equal Area Net')
Wnet()
title(sub='Wulff Equal Angle Net')
dev.off()
```

From here on out we will use the equal area stereonet, although all the following can be accomplished by Wulff nets. To plot a set of poles from the experiment, we invoke a few calls to built in functions,

```
az=c(-149,-154,-125,-142,-128,-123,-127,-137,-130,-130,-127,-150,
-138,-128,-141,-128,-143,-122,-130,-129,-158,-131,-128,-109,-134)
dip=c(54,62,52,53,53,45,43,50,54,52,54,53,54,53,56,52,54,46,54,48,51,
53,65,60,53)
```

In this case dip angles are apparently measured from the zenith, as one would for an upper hemispherical projection.

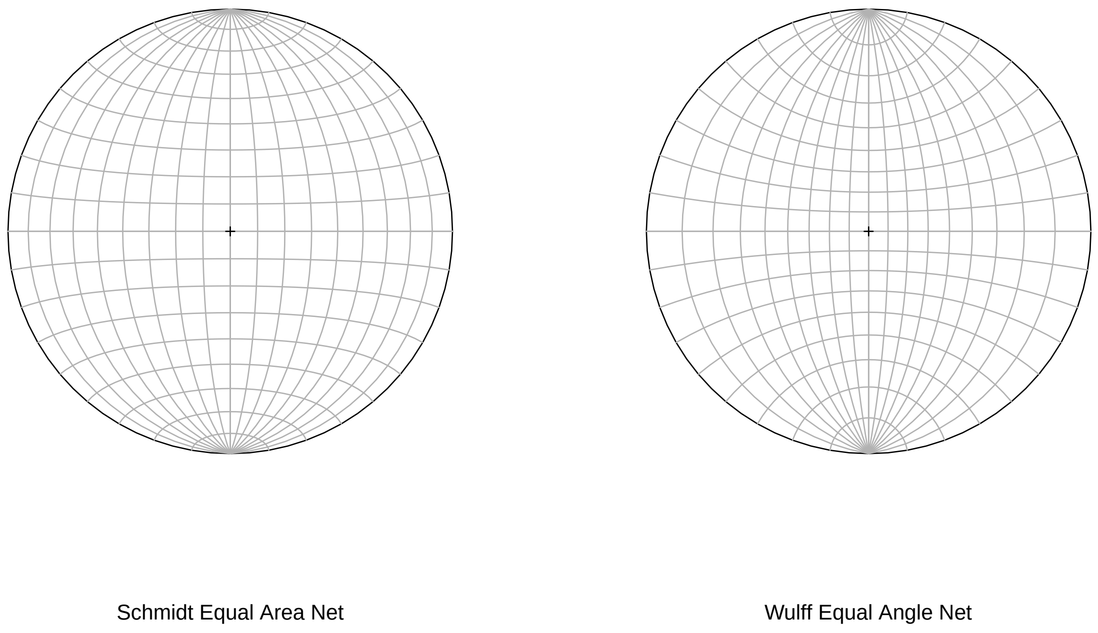

Figure 6.7: Schmidt (Equal Area) and Wulff (Equal Angle) Nets

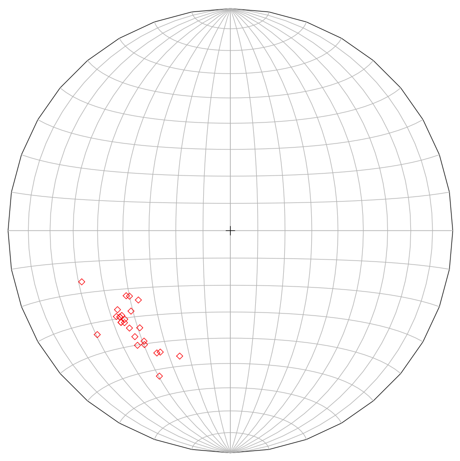

Figure 6.8: Schmidt Net with observations plotted

If we want to know the mean vector summarizing these points we cannot simply average the inclination and declination angles. This would lead to serious errors in many applications. Rather the poles should be converted to Cartesian coordinates, then averaged and, if necessary, the resultant vector is converted back to polar form. The formulas derived above are applied to all the vectors, such that, if the angles are given in degrees, we can form a matrix of Cartesian vectors,

```
DEG2RAD = pi/180
   a = dip * DEG2RAD
   b = (az) * DEG2RAD
   x = sin(a) * cos(b)
   y = sin(a) * sin(b)
   z = cos(a)
   v = cbind(x, y, z)
```

The matrix v consists of three columns, representing the x,y,z coordinates of each pole. N is the number of poles in our data set. To get the summary vectors we average the N Cartesian coordinates individually, first by taking the sums

$$R_e = \sum x_i \tag{6.2.3}$$

$$R_n = \sum y_i \tag{6.2.4}$$

$$R_d = \sum z_i \tag{6.2.5}$$

The length of this vector is R,

$$R = \sqrt{R_e^2 + R_n^2 + R_d^2} \tag{6.2.6}$$

and, of course the average direction is $R_e/N, R_n/N, R_d/N$. If R is is close to 1 the vectors are dispersed randomly around the sphere. On the other hand if R is close to N the vectors are nearly all aligned in the same direction. The precision parameter (K) can be thus formed by taking the

$$K = \frac{(N-1)}{(N-R)} \tag{6.2.7}$$

The precision parameter provides an estimate of the standard deviation (in degrees) of the vector poles via the approximation,

$$S \approx \frac{81^\circ}{\sqrt{K}} \tag{6.2.8}$$

In R the sum of the coordinates is the mean vector, and the inverse coordinate transformation is applied to get the resultant pole.

```
 Rn = sum(y)
    Re = sum(x)
    Rd = sum(z)
    N = length(x)
    Ir = 180 * atan2( sqrt(Rn^2 + Re^2), Rd)/pi
    Dr = 180 * atan2(Re, Rn)/pi
```

We can now plot the full data set with the resultant mean orientation (blue triangle),

```
JPOST(file="/home/lees/Mss/SEIS_BOOK/Fmech/FIGS/netpoint2.eps" , width=16, height=10)
####  X11(w=16, h=10)
pnet(MN)
PTS = qpoint(az, dip, UP=FALSE,  col=2)
PTS = qpoint(Dr, Ir, UP=FALSE,  col='blue', pch =2, cex=1.5)
dev.off()
```

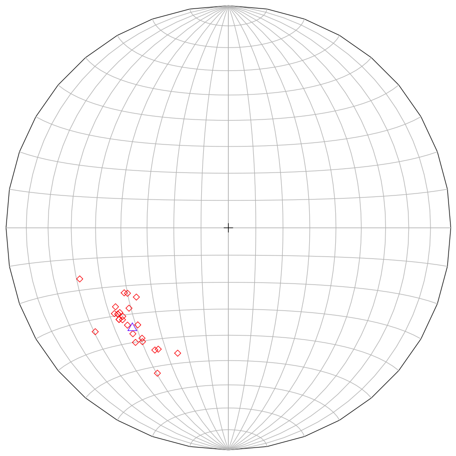

Figure 6.9: Schmidt Net, observations and mean vector plotted

The next step is to provide an estimate of the precision. This approximation is good if $R/n < 0.095$

```
R = sqrt(Rn^2 + Re^2 + Rd^2)
    K = (N - 1)/(N - R)
    S = 81/sqrt(K)
```

We can see that R is much greater than 1, close to N so the precision parameter K is large and there is high precision. The standard deviation in this case is about 10°.

Finally an estimate of the 95% confidence bounds may be estimated

$$\alpha_{95} = \arccos\left(1 - (N - R)\, 20^{\frac{1}{N-1}}\right) \tag{6.2.9}$$

```
alpha=1-0.95
Alpha95 = 180 * acos(1 - ((N - R) * (((1/alpha)^(1/(N - 1))) - 1)/R))/pi
JPOST(file="/home/lees/Mss/SEIS_BOOK/Fmech/FIGS/netpoint3.eps" , width=16, height=10)
####  X11(w=16, h=10)
pnet(MN)
jj = qpoint(az, dip, UP=FALSE,  col=2)
jj =  qpoint(Dr, Ir, UP=FALSE,  col='black', pch=6)
jj = addsmallcirc(Dr, Ir, Alpha95, lty=1, lwd=2)
dev.off()
#####  acos(1+(log(0.05/(K*R))  ))/DEG2RAD
```

The last step is to provide a quantitative estimate of the clustering of the points. If the points are show a girded distribution versus a very pointed distribution this can be represented by replacing the distribution by an equivalent ellipse that captures the essence of the distribution on the sphere. If the ellipse is very elongate, the points are clustered tightly in a a narrow area. If they are randomly distributed, however, the ellipse will be nearly spherical. This can be summarized in a single coefficient κ which is derived from the eigenvalue decomposition of the variance-covariance matrix of Cartesian vectors. The matrix is a measure of the distance each point is from a (special) axis, much like the moment of inertia of a body in physics. If the Cartesian coordinates are stored in an N by 3 matrix v, this is achieved by calculating,

$$B = \begin{bmatrix} N & 0 & 0 \\ 0 & N & 0 \\ 0 & 0 & N \end{bmatrix} - \begin{bmatrix} \sum x_i x_i & \sum x_i y_j & \sum x_i z_j \\ \sum y_i x_j & \sum y_i y_i & \sum y_i z_j \\ \sum z_i x_j & \sum z_i y_j & \sum z_i z_i \end{bmatrix} \tag{6.2.10}$$

Eigenvectors and eigenvalues can extracted for this matrix, although most authors simply calculate the Eigenvalues of the second matrix on the right hand side, the variance-covariance matrix Ψ. (We note that

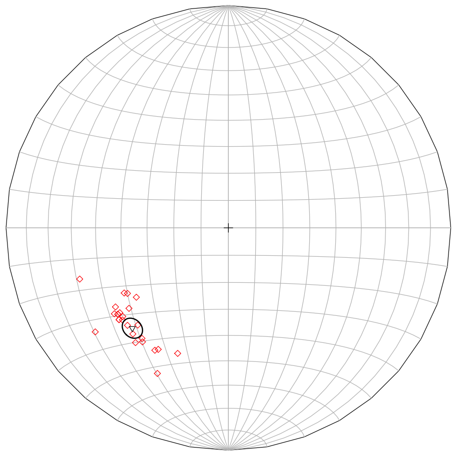

Figure 6.10: 95 % confidence bound for data

[?] does not mention this discrepancy. See [?]) The eigenvectors represent directions along which variance is maximized or minimized. In R these are obtained with the function *eigen*(). The cluster coefficient is the log eigenvalue ratios,

$$\kappa = \frac{\log\left(\frac{\epsilon_1}{\epsilon_2}\right)}{\log\left(\frac{\epsilon_2}{\epsilon_3}\right)} \tag{6.2.11}$$

```
KapT = t(v) %*% v
B = length(x) * diag(3) - KapT
E1 = eigen(B)
E = eigen(KapT)
Kappa = log(E$values[1]/E$values[2])/log(E$values[2]/E$values[3])
```

The value of κ =2.76 shows that the data is clustered near a central vector. We can see this by calculating $\kappa =$ for a variety of sample distributions (Figure 6.11). Depending on the value of κ

say spatially distributed, one can give a measure of the concentration distributed in space.

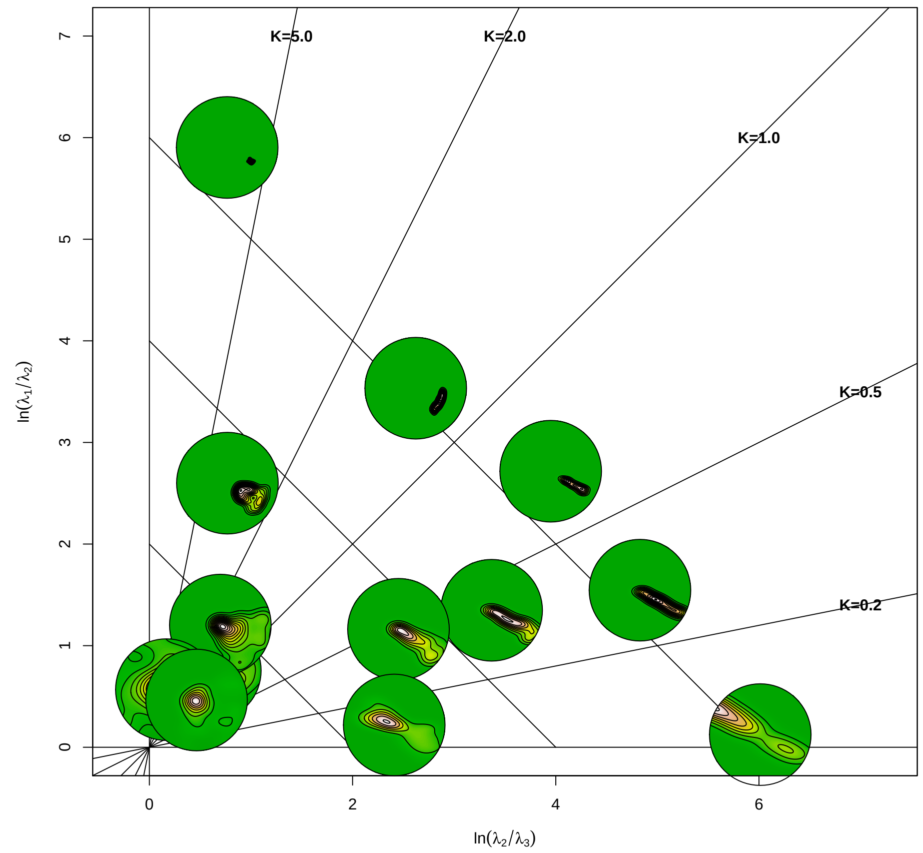

Figure 6.11: Kappa values and eigen value ratio classification

6.3 Focal Mech 3

6.3.1 Example: Earthquake Focal Mechanisms

The application of stereonets can be particularly useful for earthquake data where thousands of P and T-axes (pressure and tension) are distributed in laterally and in depth. The focal mechanism is a way geologists plot tensor quantities in space.

When earthquakes occur below the surface they radiate energy along two sets of orthogonal directions forming two sets of force couples called a double couple. (Landslides can radiate single couple patterns and explosions have an isotropic component.) The double solutions can be derived from the first motions at seismographic stations distributed at the surface by projecting the ray-paths of the waves back to the hypocenter and plotting the directions of the first motions on a stereographic projection of the focal sphere. The resulting solution is a set of two planes, one representing the actual fault plane where the earth ruptured, the other is called the auxiliary plane that also radiates energy towards the surface. The data consists of points on the focal sphere indicating whether the motion was away or towards the earthquake hypocenter. These are plotted and a set of best fitting orthogonal planes are determined, usually by grid search methods. An example is shown in Figure 1 where the actual seismograms are displayed for illustration. A set of poles (points on the sphere) are derived that relate geophysical information about the nature of the earthquake orientation and slip vector and the compressional and tensional radiation axes. The slip vectors correspond possibly to striations one might observe on the fault surface as the earth scrapes the two planes during rupture.

Strike slip, normal and reverse faults each have characteristic shapes and can represented in a visually revealing way be introducing the associated *beachball displays.*

An example of a typical representation of a focal mechanism for a fault strike-dip-rake of (65, 32, -34) is presented in Figure 6.14. Several of the relevant parameters are marked on this figure, although generally one only plots the colored hemispheres. The hemispheres show the investigator the orientation of the fault and auxiliary planes as seen from above projected on a lower hemisphere bowl. The colored region shows information about the radiation pattern of the seismic waves after rupture.

6.4 Map Views and Summary Representations

Once focal mechanisms are determined for each earthquake they can be plotted in map view using standard projections as a set of beach balls (Figure 6.15). The beach balls relate the nature of changing stress in

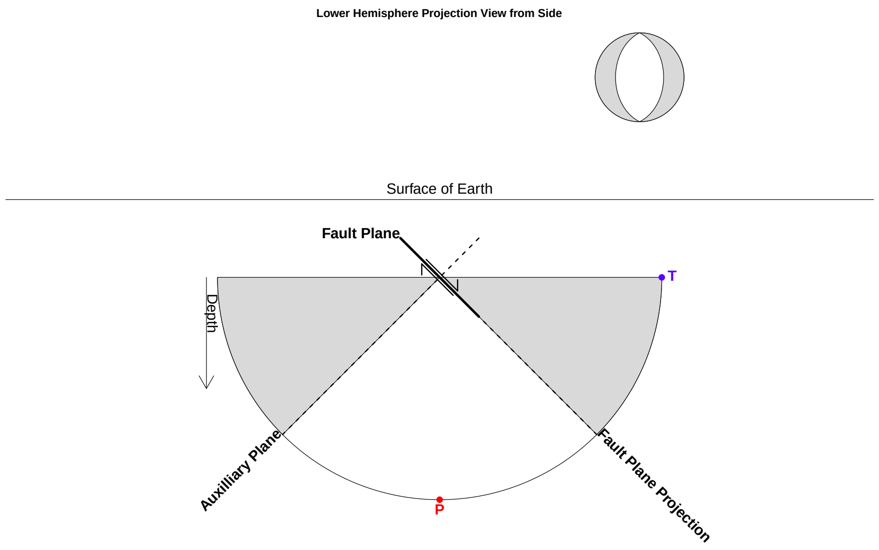

Figure 6.12: Earthquake Focal Mechanism

the earth during earthquake swarms. In the Kamchatka example, there are thousands of beach-balls, some covering others which makes it difficult to discern important patterns.

In this case we can extract the P and T-axes from each focal mechanism and plot them all on one stereonet graph. Each P and T-axis is a vector in space derived from the focal mechanism. These are gathered together in one plot and statistical testing my be applied to determine if the distributions are uniform random, or clusters.

But really we are interested in the spatial distribution of P and T-axis across the Aleutian-Kamchatka Arc. To see how we might visualize this, we can break down the data (Figure 6.15) into small spatially distributed subsets. In each subset we project the P-T axes and contour or image the results, plotting them in their respective positions on the map. This provides a way to investigate variations in orientation of stress across a wide geologic region. As a further reduction of the data, one could calculate, plot and contour κ (concentration parameter) for each region to see how stress is focused in some regions and disbursed in others. I will leave that exercise to the reader.

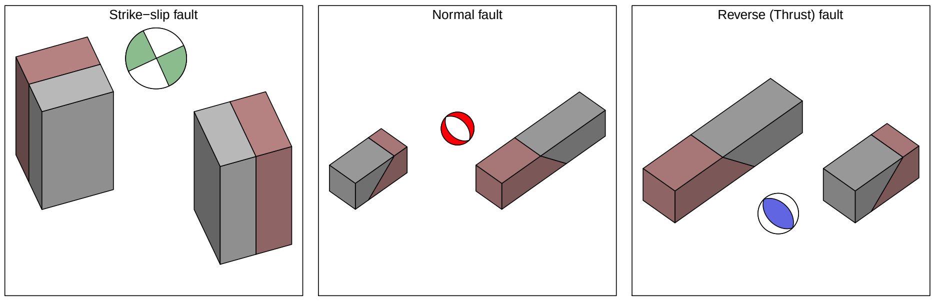

Figure 6.13: Examples of Faults and Focal Mechanisms

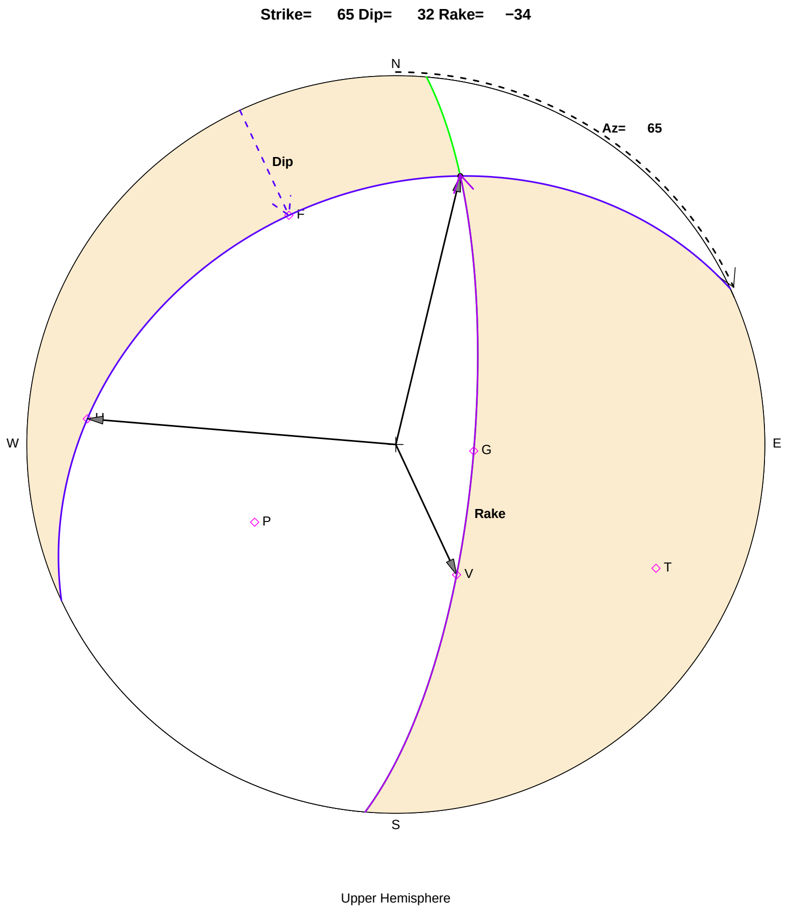

Figure 6.14: Earthquake Focal Mechanism

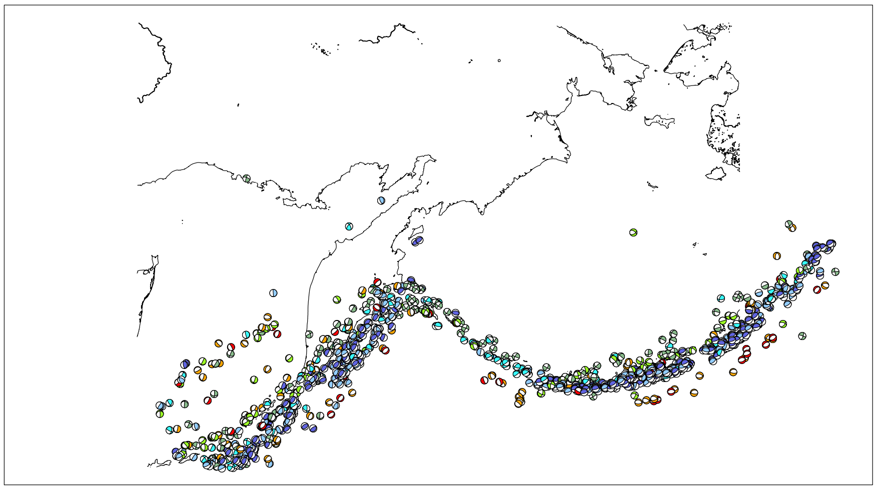

Figure 6.15: Kamchatka focal mechanisms

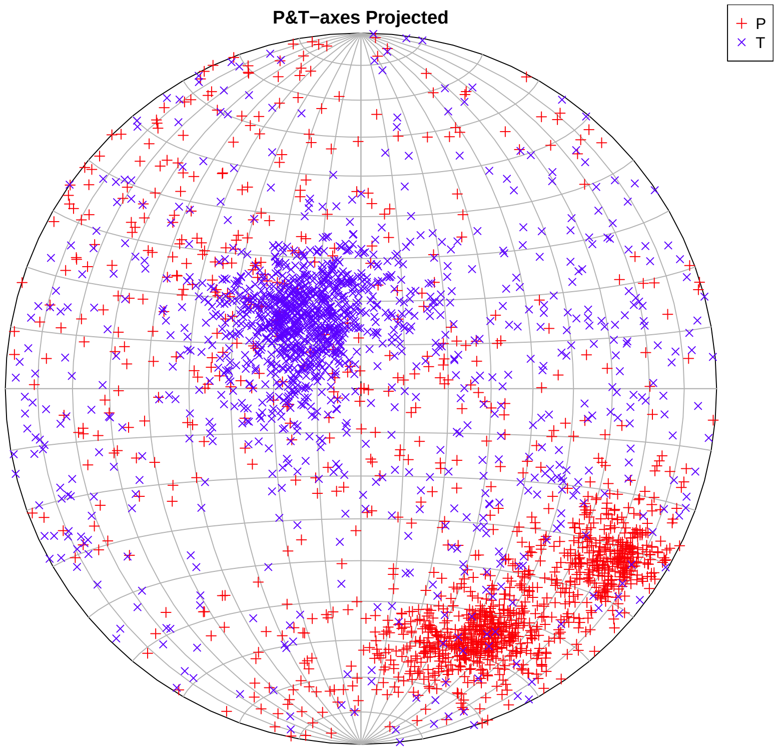

Figure 6.16: Kamchatka P-T axes

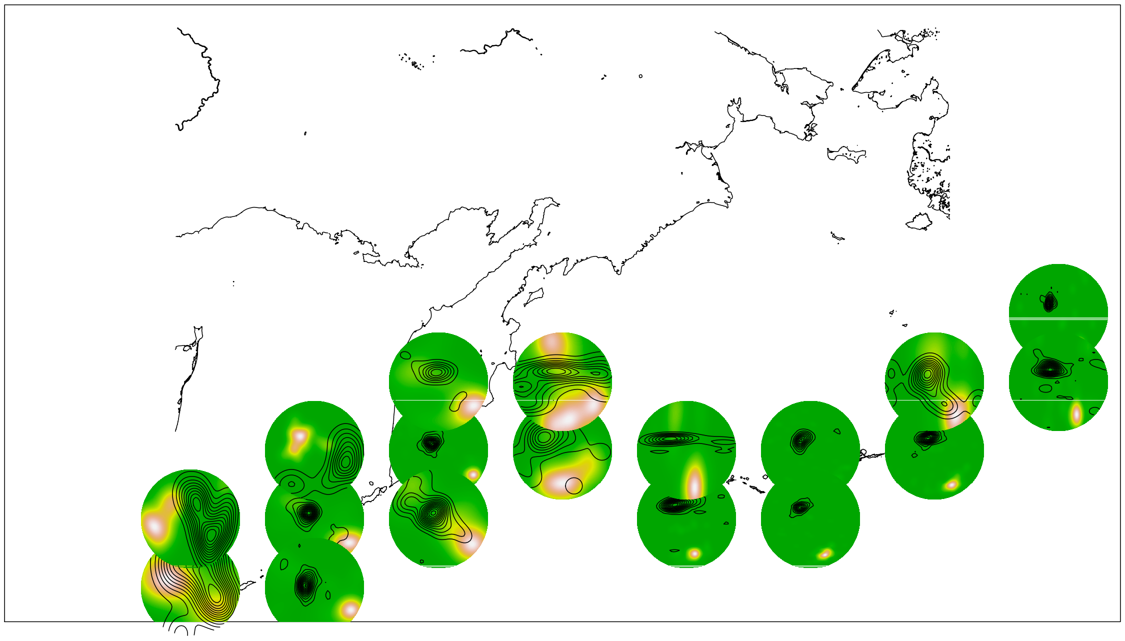

Figure 6.17: Geographic plot of Aleutian-Kamchatka P-T axes

Chapter 7

Particle Motion

7.1 Hodograms

7.2 Particle Motion Analysis

Start out by calling the RSEIS library,

```
library(RSEIS)
```

Then load some data and plot. This data is from the Coso Geothermal field in Coso Californmia. The data is sampled at 250 sampled/sec and there are numerous stations recorded on three components. Some stations are too noise for particle motion analysis.

```
data(GH)
# swig(GH, SHOWONLY=TRUE)
```

In this case the GH structure holds the phase arrival information as well as the station locations and event location.

```
print(GH$pickfile$STAS$name)
```

```
 [1] "CE1"  "CE4"  "CE3A" "SM5"  "NV6"  "CE2"  "NV1"  "CE7"
 [9] "NV10" "CE8"  "NV4"  "NV5"  "NV2"  "CE1"  "CE4"  "CE3A"
[17] "SM5"  "NV6"  "CE2"  "CE6"  "CE7"  "CE8"  "NV4"  "NV5"
```

We will choose one station and do the hodogram analysis there, but our analysis could easily be put in a loop to cover all the stations.

```
thesta = "CE1"
iwv  = which(GH$STNS==thesta & GH$COMPS=="V")
iwn  = which(GH$STNS==thesta & GH$COMPS=="N")
iwe  = which(GH$STNS==thesta & GH$COMPS=="E")
data = cbind(GH$JSTR[[iwv]], GH$JSTR[[iwn]], GH$JSTR[[iwe]])
```

Next we get the station back azimuth to the seismic event, which has been located previously. The information on the location of the seismic event is stored also in the pickfile.

```
ipphase = which(GH$pickfile$STAS$name==thesta & GH$pickfile$STAS$phase=="P" )
isphase = which(GH$pickfile$STAS$name==thesta & GH$pickfile$STAS$phase=="S" )
lat=GH$pickfile$STAS$lat[ipphase]
lon = GH$pickfile$STAS$lon[ipphase]
DAZ = rdistaz(lat, lon,   GH$pickfile$LOC$lat,GH$pickfile$LOC$lon )
 rbaz = grotseis(DAZ$baz, flip=FALSE)
```

To illustrate the approach we start by plotting a map view of the stations and the earthquake source,

```
plot( c(GH$pickfile$STAS$lon, GH$pickfile$LOC$lon) ,
     c(GH$pickfile$STAS$lat, GH$pickfile$LOC$lat), type='n', xlab="LON", ylab="LAT")
points(GH$pickfile$STAS$lon, GH$pickfile$STAS$lat, pch=6)
points(GH$pickfile$LOC$lon, GH$pickfile$LOC$lat, pch=8)
text(GH$pickfile$STAS$lon, GH$pickfile$STAS$lat,  GH$pickfile$STAS$name, pos=3)
```

```
##  plot( c(lon, GH$pickfile$LOC$lon) , c(GH$pickfile$STAS$lat, GH$pickfile$LOC$lat))
##  text(lon, lat, labels="station")
```

These can be plotted in km rather than LAT-LON by either projection or by simply using the distances to the stations and azimuths to get the approximate flat orientation. We show the orientation of the horizontal components after the seismogram gets rotated by the designated angle. The red arrow is the radial component and the blue is the transverse.

```
x = DAZ$dist*sin(DAZ$baz*pi/180)
y = DAZ$dist*cos(DAZ$baz*pi/180)
plot(c(0,1.3*x), c(0,1.3*y), type='n', asp=1, xlab="E-W, km", ylab="N-S, km")
points(c(0,x), c(0,y), pch=c(3,6))
text(x,y, labels="station", pos=1)
text(x,y, labels=GH$pickfile$STAS$name[ipphase], pos=2)
text(0,0, labels="source", pos=3)
vecs  = rbind(c(0,0,1), c(0,1,0))
bvec  = vecs %*%  rbaz
bvec = .1*DAZ$dist*bvec
arrows(x,y, x+bvec[,2], y+bvec[,3], col=c("red", "blue"))
```

We first plot the data in its original Vertical-North-East orientation,

```
## data = cbind(GH$JSTR[[iwv]], GH$JSTR[[iwn]], GH$JSTR[[iwe]])

    vnelabs=c("Vertical", "North", "East")
rotlabs=c("Vertical", "Radial(away)", "Transvers(right)")
 xt=seq(from=0, by=GH$dt[iwv], length=length(GH$JSTR[[iwv]]))
 PLOT.MATN(data, tim=xt, dt=GH$dt[iwv], notes=vnelabs)
```

And then we rotate the seismograms so that they are oriented Vertical-Radial-Transverse, is is often done in seismic analysis:

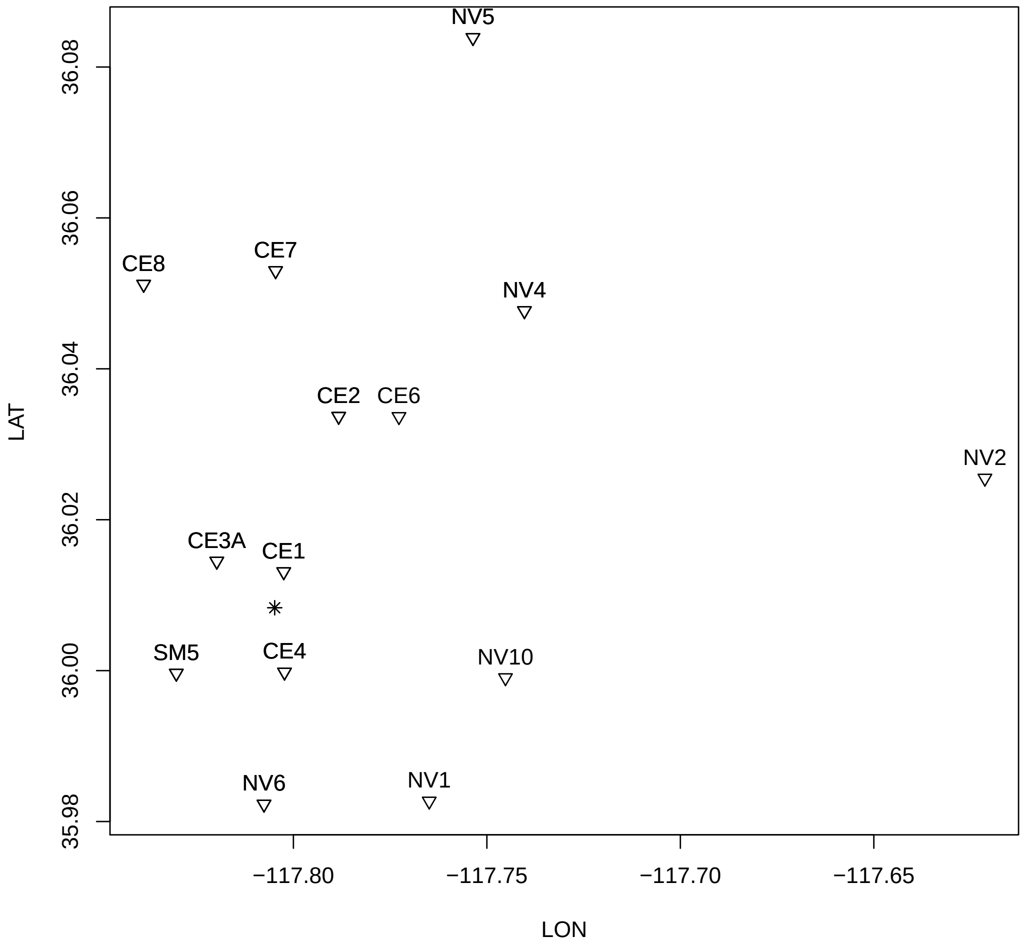

Figure 7.1: Map view of the stations and earthquake source.

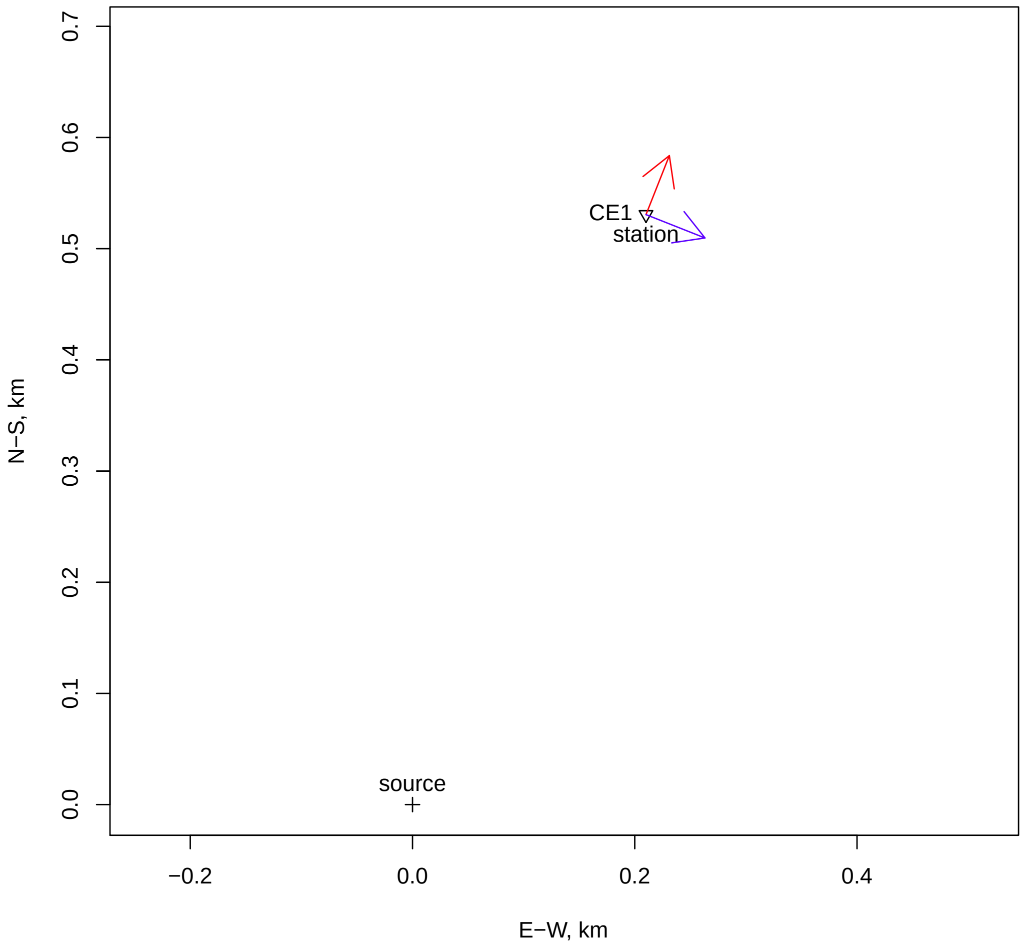

Figure 7.2: map2

```
## data = cbind(GH$JSTR[[iwv]], GH$JSTR[[iwn]], GH$JSTR[[iwe]])
btemp  = data  %*%  rbaz
 PLOT.MATN(btemp, tim=xt, dt=GH$dt[iwv], notes=rotlabs)
```

Next we extract information from the seismic structure that tells us when the P and S-arrival were estimated. This information is stored in the pickfile structure, but we want to know how many seconds paste the start of the trace the arrivals came in, so we can window that portion of the trace for hodogram analysis.

```
i1 = match(GH$STNS[iwv],  GH$pickfile$STAS$name)
reft = list(jd=GH$info$jd[iwv], hr=GH$info$hr[iwv], mi=GH$info$mi[iwv], sec=GH$info$sec[iwv] )
ptim = list(jd=GH$pickfile$LOC$jd, hr=GH$pickfile$LOC$hr, mi=GH$pickfile$LOC$mi, sec=GH$pickfi
stim = list(jd=GH$pickfile$LOC$jd, hr=GH$pickfile$LOC$hr, mi=GH$pickfile$LOC$mi, sec=GH$pickfi
t1 = secdifL( reft, ptim)
t2 = secdifL( reft, stim)
 PLOT.MATN(btemp, WIN=c(5,8) , tim=xt, dt=GH$dt[iwv], notes=rotlabs)
abline(v=t1, col='red', lty=2)
abline(v=t2, col='blue', lty=2)
mtext(side=3, at=t1, line=.1, text="Pwave", col='red')
mtext(side=3, at=t2, line=.1, text="Swave", col='blue')
pwin = c(t1-.02, t1+.09)
swin = c(t2-.01, t2+.1)
abline(v=pwin, col='red', lty=2)
abline(v=swin, col='blue', lty=2)
```

So we now extract that portion and apply the hodogram program on the P-wave arrival

```
rbow=rainbow(140)[1:100]
atemp = btemp[xt>pwin[1]&xt<pwin[2]  ,]
##  PLOT.MATN(atemp,  tim=xt[xt>pwin[1]&xt<pwin[2]], dt=GH$dt[iwv], notes=rotlabs)

 hodogram(atemp, dt=GH$dt[iwv]  ,labs=rotlabs,  STAMP=thesta,  COL=rbow )
```

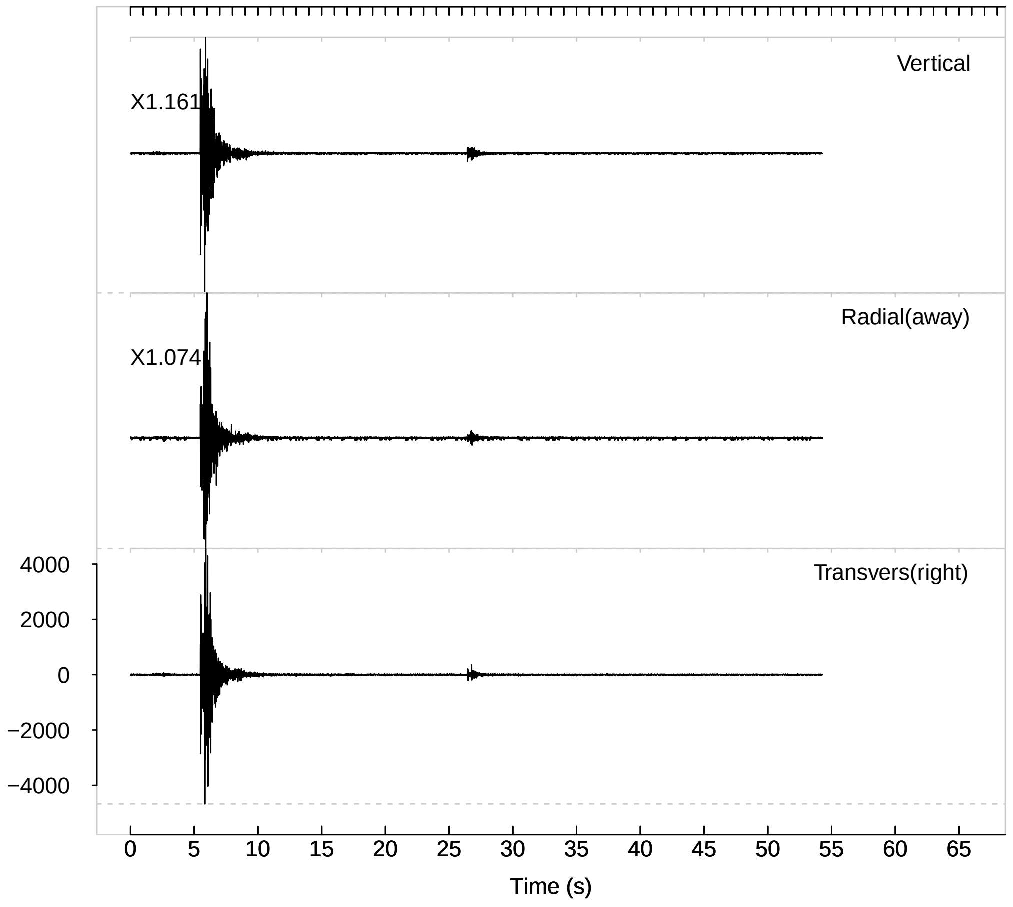

Figure 7.3: fig4

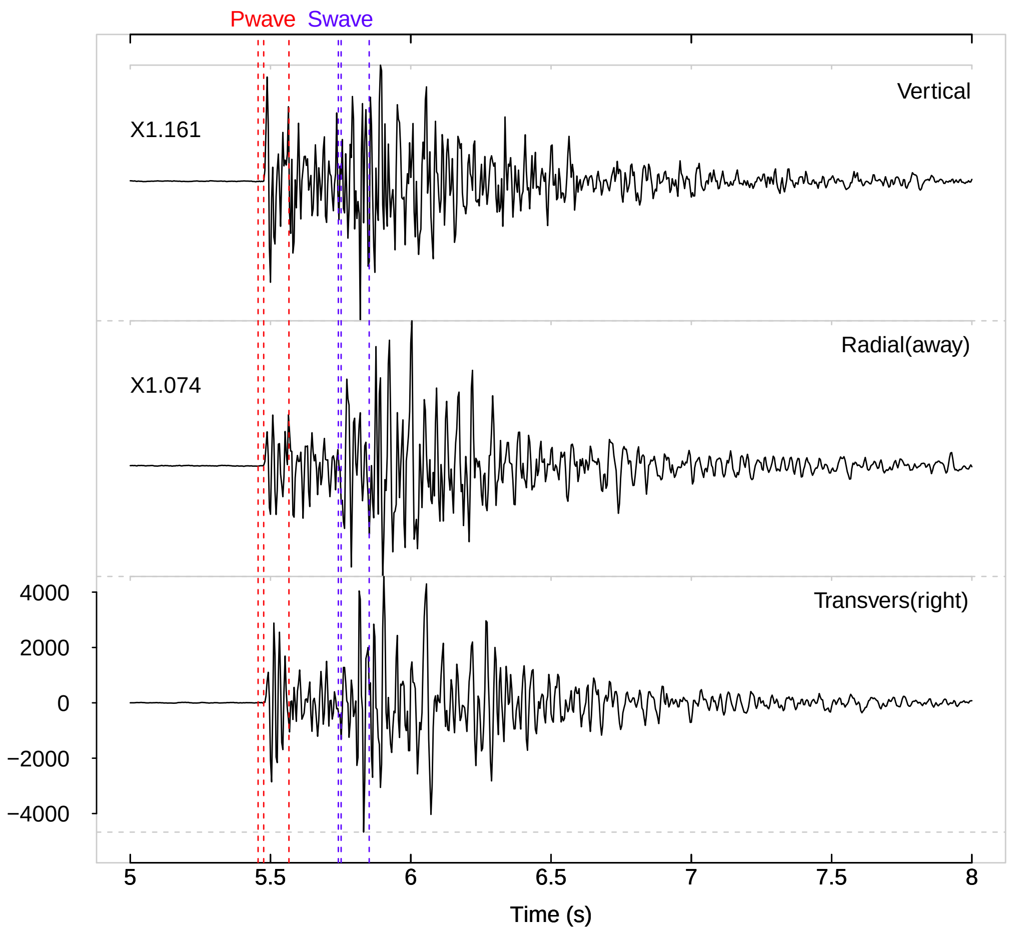

Figure 7.4: fig5

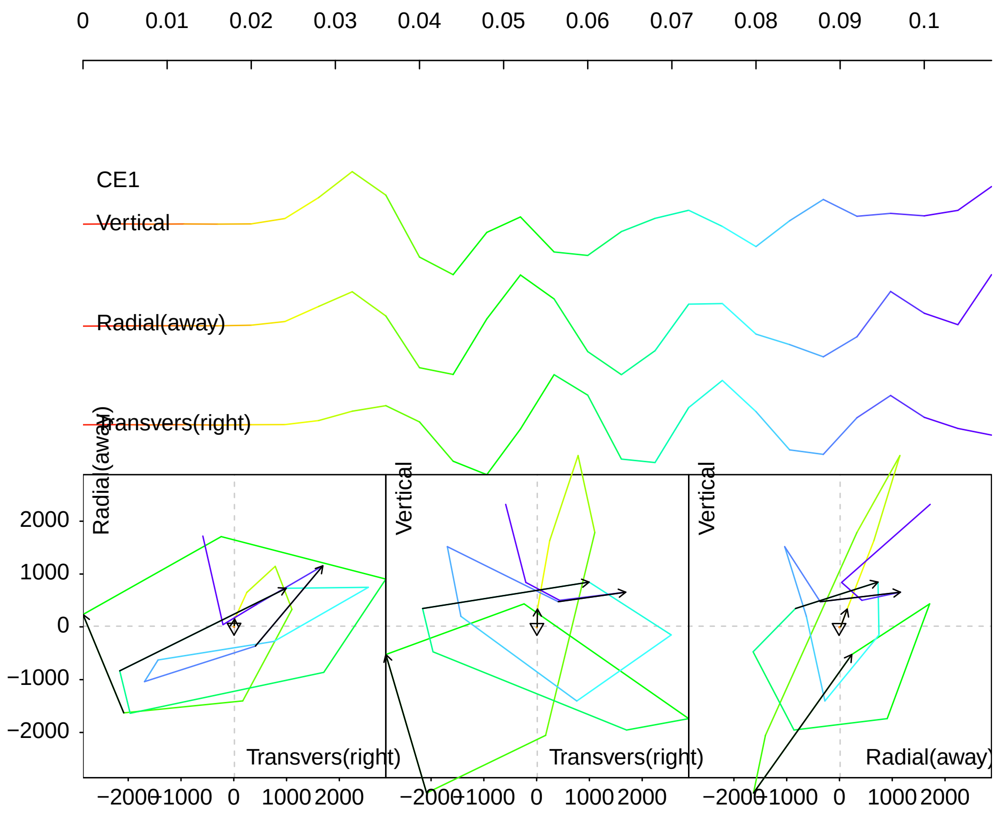

Figure 7.5: fig6

and on the S-wave arrival

```
 atemp = btemp[xt>swin[1]&xt<swin[2]  ,]
 ###  PLOT.MATN(atemp,  tim=xt[xt>swin[1]&xt<swin[2]], dt=GH$dt[iwv], notes=rotlabs)

  hodogram(atemp, dt=GH$dt[iwv]  ,labs=rotlabs,  STAMP=thesta,  COL=rbow )
```

Clearly, a loop can be programed so that all the stations are examined for the particle motion and specific patterns will be revealed.

For example, we may wish to differentiate between direct arrival of body waves and later arrivals of surface waves. The Raleigh wave has retrograde motion int he vertical-radial components, so we expect to see the first motions moving opposite the direction of wave propagation (i.e. towards the source) as the motion is decomposed.

Hodograms can be used to estimate the arrival of a "split shear wave" often used to detect anisotropy in the geologic structures in the subsurface. In some cases anisotropy indicates crack orientation and in the mantle it may refer to fluid flow as olivene crystals align along the direction of flow.

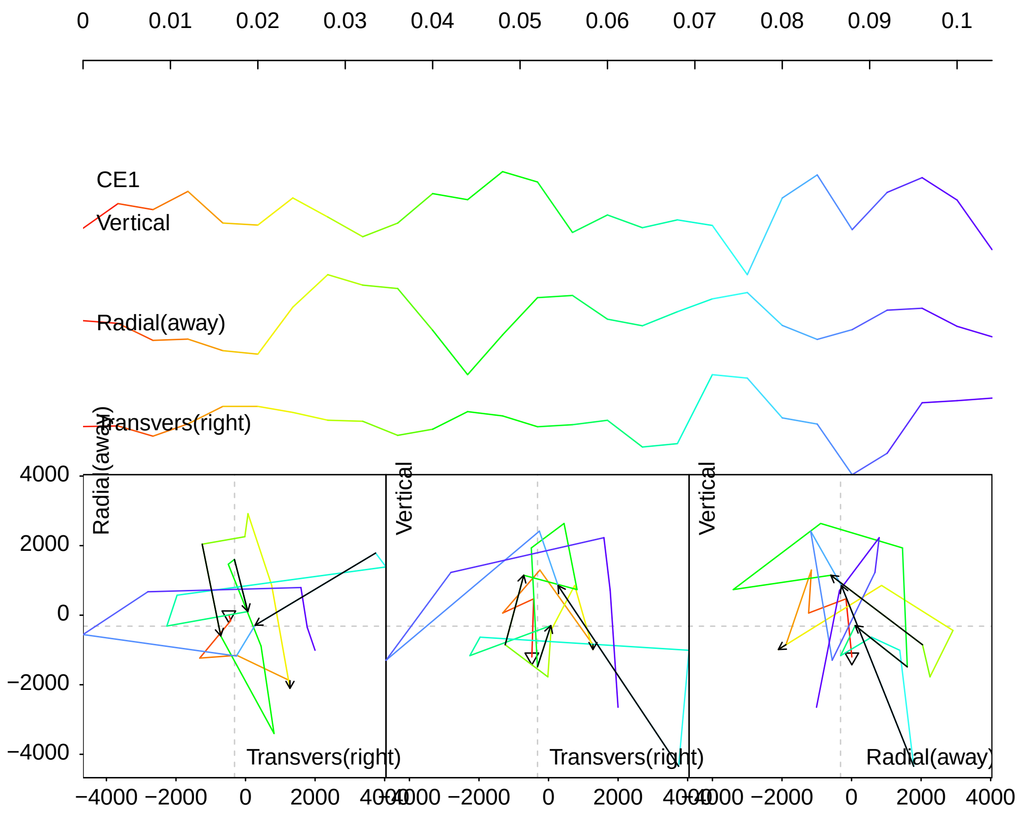

Figure 7.6: fig7

Chapter 8

Inverse Theory

8.1 Tomography

8.2 Tomographic Inversion: Introduction

Seismic tomography uses the travel time of elastic waves to probe the internal structure of the earth. It differs from traditional medical tomography in four major aspects:

1. acoustic signals travel in highly curved raypaths in media that vary in 3-dimensions,

2. the travel time is a non-linear function of the velocity field ("velocity" field in the seismic sense is the scalar wave speed),

3. when the sources are earthquakes, the distribution of rays covering the target cannot be controlled and is often highly inhomogeneous and

4. uncertainties in the travel time exist because the source location and origin time must be determined from the observations themselves. These differences indicate that special care must be taken when techniques borrowed from the medical field are applied to seismic data. Specifically, due to the non-uniform distribution of sources and receivers, the convolutional techniques of inversion, common in medical tomography, are inapplicable in the seismic case and iterative approaches are used instead.

In this demonstration we show how to set up a synthetic 2D tomographic modeling experiment and perform the inversion via matrix inversion.

8.3 History

Tomography (literally, 'slice picture') originated in radio astronomy as a method to image aspects of remote regions of the universe. Later physicists and bio-physicists collaborated to create the first methodology and instrumentation that led to the first tomographic analysis of live tissue, especially human bodies. This approach was called 'computer aided tomography' or CAT scans. Researchers who pioneered these methods received the Nobel Prize in physiology and medicine in 1979 (Allan Cormack and Godfrey Hounsfield). At the same time seismologists recognized that similar methodology could be applied to imaging the earth. Early papers on these approaches were not called tomography, but simply 'three-dimensional analysis'. It was not until the early 1980's that data sets large enough to actually mimic an approach similar to medical tomography emerged.

8.4 Basic Idea

The basic idea is illustrated in cartoon form in Figure 8.1 . Earthquakes emit seismic energy that travels out to the stations at the surface. At first, we assume an intervening velocity structure, typically one dimensional, and use that to predict traveltimes to each station. If the model is correct the difference between predicted and observed arrivals will be small. If waves pass through anomalous structures, however, travel times will be perturbed and the differences will become significant. Seismic tomography often involves using the travel time residuals to reconstruct anomalies where large numbers of raypaths overlap at varying angles. It can be shown that with complete coverage from all angles, the anomalous body can be reconstructed perfectly. This ideal situation is never achieved in real analyses, of course.

8.5 Inversion approach

Nearly all seismic tomography is founded on a linearization of the highly non-linear seismic inversion problem. In equation (1) it was noted that the raypath depended on the velocity model which also depends on the raypaths in the inversion. Furthermore, earthquake locations are also derived using the velocity models, so this introduces an additional non-linearity. To circumvent the nonlinearity we usually introduce a linearization of the inversion and iterate a sequence of linear inversion in the hopes that we

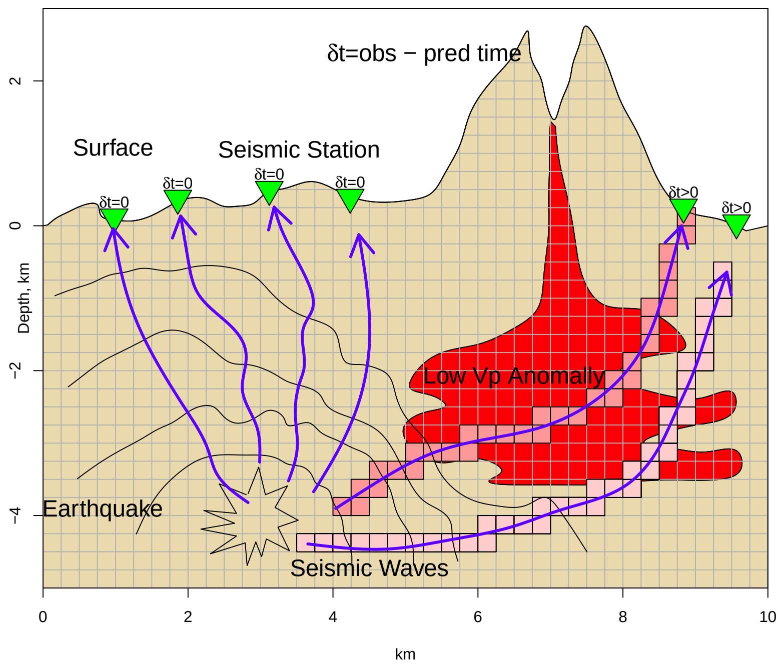

Figure 8.1: Tomographic inversion of a Magma Chamber. The grid in the subsurface represents the grid used for the tomographic inversion. The two rays on the right traverse through the anomalous magma body. those on the left do not, so they do not contribute changing the model. The blocks highlighted are the ones that are perturbed sequentially in row-action methods of inversion.

will converge to the correct non-linear solution. This is accomplished by assuming we have a reasonably close initial guess to the correct solution and then seeking a small perturbation which will drive the model closer to the correct, final model. The sequence of calculations involves making an initial (typically 1D) model, finding the raypaths in that model, perturbing the (3D) model so the travel times are minimized, adding new perturbations to the first model and iterating. Once the new model is derived, new earthquake locations are calculated and the process is repeated. These iterations usually converge in a few to several steps. Convergence is determined when the models change by a very small amount between iterations and travel time differentials get small.

The inversion described above is typically discretized (digitized) by assuming the earth is composed of blocks, or nodes of points, where the velocity is defined. The integrals in Equation (1) can be discretized and cast in matrix form. The matrices involved include a data design matrix which describes the way the raypaths intersect the earth model and the data (travel time perturbations), i.e. the differences between the observed travel times and calculated times through the current model. New, upgraded, models are derived by least-squares methods. Since the matrices are very sparse, specialized methods have been developed to store the data and arrive at an inverse solution with great speed.

The inversion described above is typically discretized (digitized) by assuming the earth is composed of blocks, or nodes of points, where the velocity is defined. The integrals in Equation (1) can be discretized and cast in matrix form. The matrices involved include a data design matrix which describes the way the raypaths intersect the earth model and the data (travel time perturbations), i.e. the differences between the observed travel times and calculated times through the current model. New, upgraded, models are derived by least-squares methods. Since the matrices are very sparse, specialized methods have been developed to store the data and arrive at an inverse solution with great speed.

8.6 Methodology and Inversion

Once all the data are collected and the relationship of the intersecting ray paths with the target region is discretized and digitized, a set of matrices is typically set up to solve the so called inverse problem: what earth model would give rise to the travel time residuals observed. If we represent the earth model as a vector of perturbations (anomalies), x, the interaction of the raypaths with the earth as A, the array of travel time residuals, Δt, one can relate the earth to the residuals by the simple linear relationship,

$$\vec{A}\vec{x} = \vec{b} \tag{8.6.1}$$

This matrix equation is solved using a variety of methods depending on the structure of the matrices, although the specific approach usually does not have a large impact on the resulting images.

8.7 Synthetic Example using R

Set up libraries used in the following:

```
library(sp)
library(splancs)
library(RTOMO)
```

Next we create a synthetic target region and grid where the earthquakes and stations will be distributed.

We set the background velocity to 4.5 km/sec and we create 2 irregular anomalies, one with a 10 percent positive perturbation and one with negative 5 percent anomaly. This is totally arbitrary.

```
NX = 100
NY = 100
xo = seq(from=1, to=NX, by=1)
yo = seq(from=1, to=NY, by=1)
### v (or s) model

V = 4.5
v1 = V+0.1*V
v2 = V-.05*V
rad = 20
```

The travel times are calculated by using the slowness in seismology, or 1/velocity.

```
MOD =  matrix(1/V, ncol=NX, nrow=NY)
PHANT =  matrix(0, ncol=NX, nrow=NY)
M = meshgrid(xo, yo)

## image(xo, yo, MOD, col="grey")
```

Two arbitrary, complex anomalous bodies are created. I made these by clicking on the screen a then saving the points:

```
##  A1 = locator(type='o', col='black')
A1=list()
A1$x=c(21.4380148625385,22.9678921197393,25.2038665725713,29.0874011485427,
  31.5587413332518,40.7380048764569,40.7380048764569,42.0325164017807,
  46.1514167096291,50.0349512856005,45.2099537821209,29.2050840144812,
  29.7934983441739,33.6770329201453,28.6166696847886,21.9087463262926,
  11.6703369896408,8.96363107305464,17.083748822813,15.3185058337351,
  7.08070521803821,11.5526541237022,14.2593600402883,15.7892372974892,
  17.6721631525056,19.0843575437679)
A1$y=c(55.6872037914692,57.8199052132701,59.0047393364929,
  59.0047393364929,55.2132701421801,58.5308056872038,
  61.9668246445498,70.6161137440758,74.7630331753555,79.8578199052133,
  83.175355450237,88.2701421800948,83.5308056872038,81.9905213270142,
  77.4881516587678,77.3696682464455,78.909952606635,71.3270142180095,
  70.1421800947867,66.1137440758294,54.5023696682464,49.4075829383886,
  51.6587677725119,57.1090047393365,55.0947867298578,52.1327014218010)
##  A2 = locator(type='o', col='black')

A2=list()
A2$x=c(90.6355400343922,82.750788016511,72.9831101436132,81.2209107593101,
  78.8672534405396,63.9215294663467,55.8014117165883,47.0928796371373,
  65.6867724554246,70.9825014226583,57.9197033034818,45.9160509777521,
  48.2697082965226,64.9806752597934,72.3946958139206,76.6312789877075,
  81.1032278933716,74.8660359986297,86.7520054584209,92.7538316212857)
A2$y=c(39.5734597156398,50.4739336492891,52.60663507109,49.0521327014218,
  46.0900473933649,46.563981042654,45.260663507109,35.9004739336493,
  37.914691943128,33.175355450237,25.5924170616114,24.2890995260664,
  16.8246445497630,15.7582938388626,11.6113744075829,9.12322274881517,
  19.4312796208531,24.7630331753554,30.2132701421801,31.3981042654028)
```

These bodies are saved as geometric outlines and all the points inside the bodies are given the perturbation assigned. We used the program inout to extract which points of the model were inside each body.

```
mypoly = as.points(as.vector(A1$x) , as.vector(A1$y))
mypoints = as.points(as.vector(M$x),as.vector(M$y))
INTEMP1 = inout(mypoints, mypoly, bound=TRUE )
mypoly = as.points(as.vector(A2$x) , as.vector(A2$y))
```

```
INTEMP2 = inout(mypoints, mypoly, bound=TRUE )
INTEMP = INTEMP1 | INTEMP2
MOD[INTEMP1] = 1/v1
MOD[INTEMP2] = 1/v2
ZEE = t(MOD)
```

```
image(xo, yo, ZEE, col=terrain.colors(100) )

##############   code
```

8.8 Station Distribution

Make the stations:

```
##############   code

stax = runif(20, min=1, max=100)
stay = runif(20, min=1, max=100)
```

```
image(xo, yo, ZEE, col=terrain.colors(100) )
points(stax, stay, pch=6, col='blue')
```

8.9 Event (source) Distribution

Make the randomly distributed earthquakes (sources).

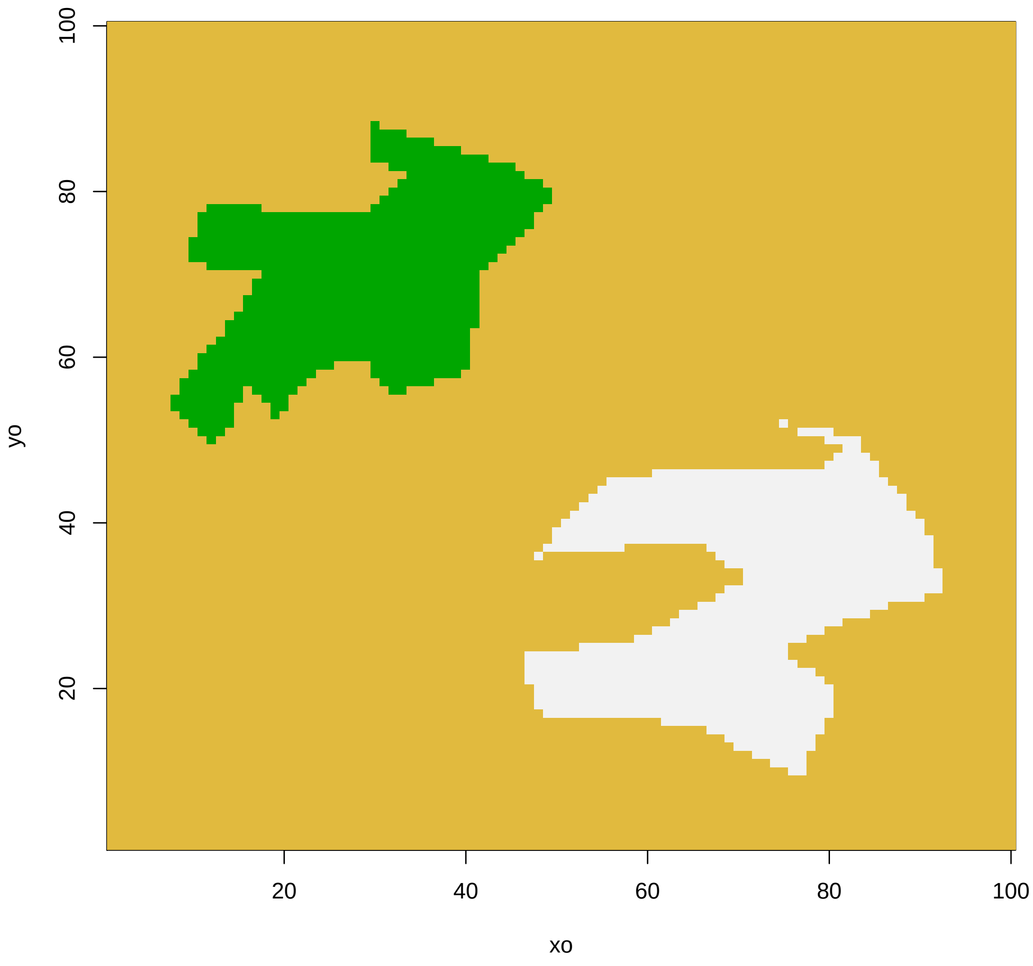

Figure 8.2: Tomographic Phantom (Synthetic model)

```
NEV = 200
evx = runif(NEV, min=1, max=100)
evy = runif(NEV, min=1, max=100)
####   get random radii for earthquake magnitude
rads = rnorm(NEV, m=50, s=10)



image(xo, yo, ZEE, col=terrain.colors(100) )
points(evx, evy, pch=8, col='red')
```

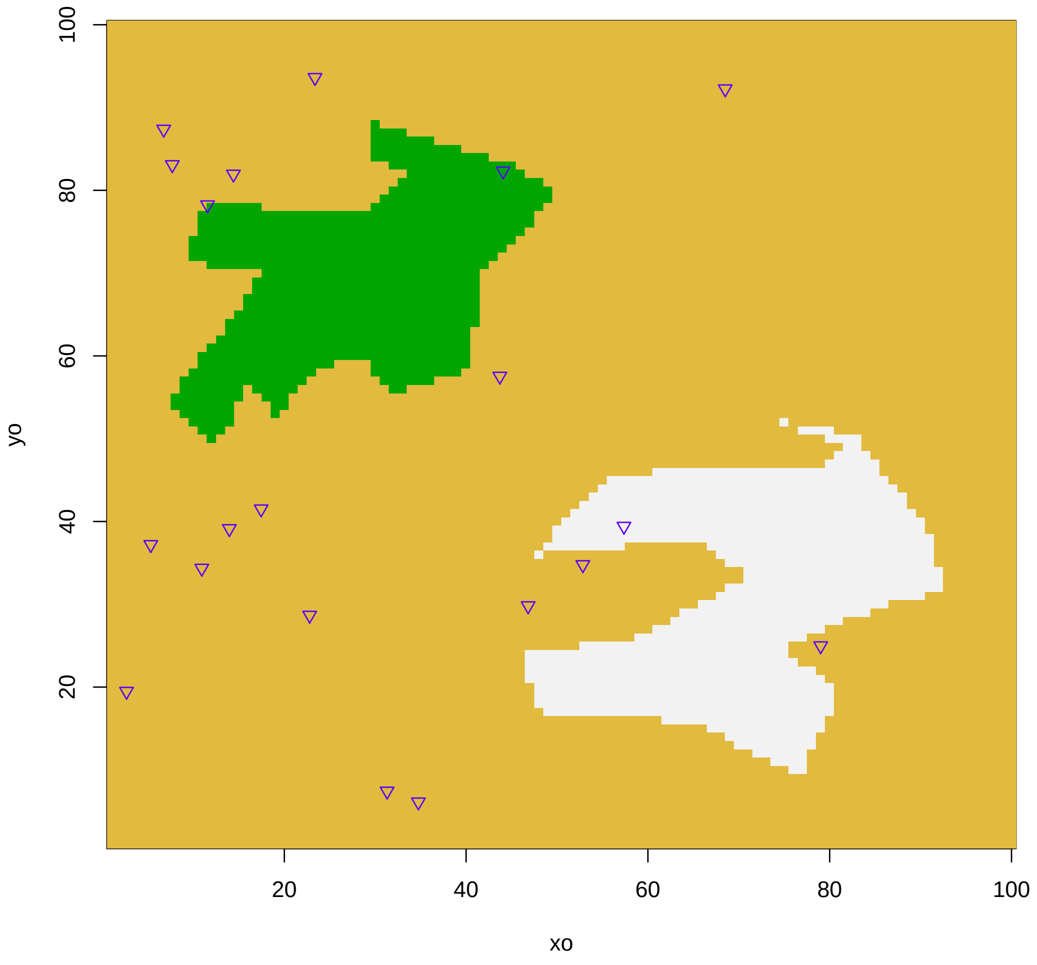

Figure 8.3: (Random) Station Distribution

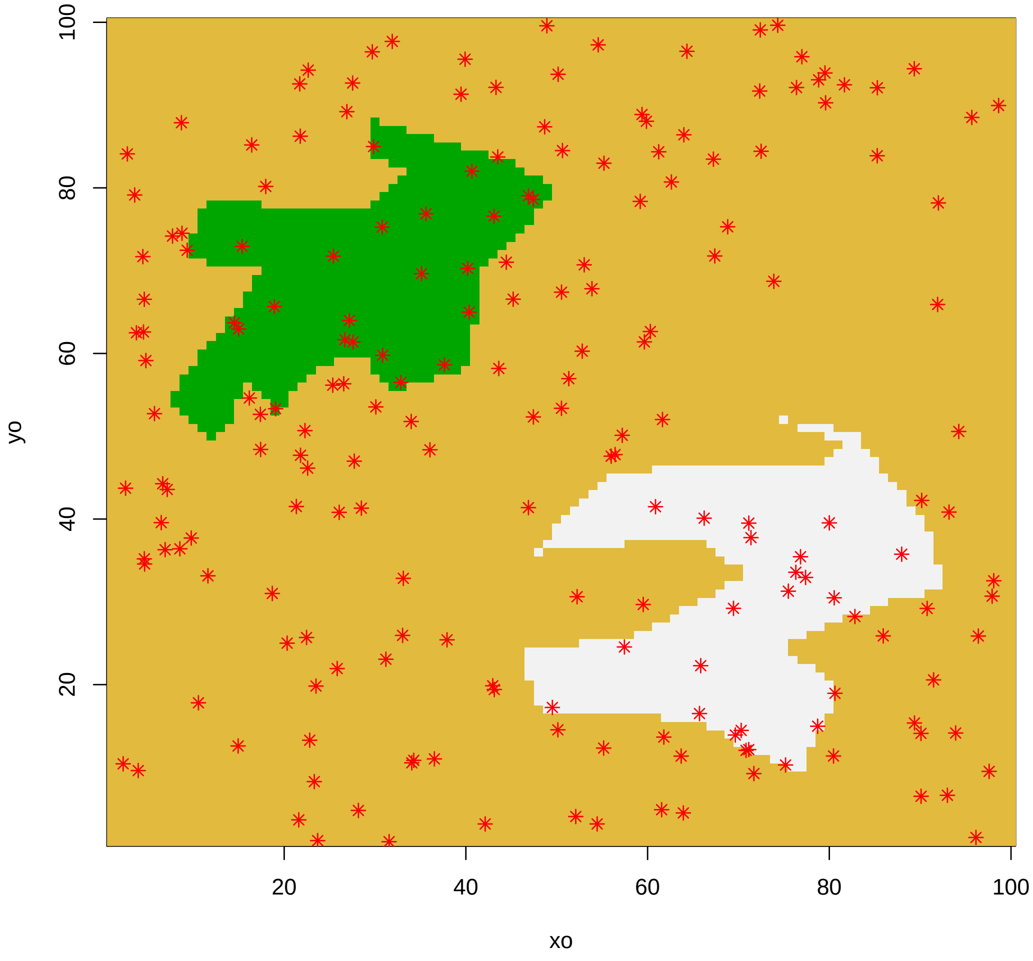

Figure 8.4: (Random) Event Distribution

Here we estimate which stations record each event by using the radius (magnitude) of the event:

```
image(xo, yo, ZEE, col=terrain.colors(100) )
points(evx, evy, pch=8, col='red')
```

```
for(i in 1:length(evx))
  {
    print(paste(sep=' ', "working on ", i))
    dis = sqrt( (evx[i]-stax)^2+(evy[i]-stay)^2)
    w = which(dis<rads[i])
    if(length(w)>1)
      {
        segments(evx[i], evy[i], stax[w],  stay[w])
}

  }
```

8.10 Prepare the Matrix

Determine the intersection of the raypaths with the model. In 2D this is just the length of the raypath in the i^{th} block.

```
COV = list()
k = 0
for(i in 1:length(evx))
  {
    print(paste(sep=' ', "working on ", i))
    dis = sqrt( (evx[i]-stax)^2+(evy[i]-stay)^2)
    w = which(dis<rads[i])
    if(length(w)>1)
      {
        segments(evx[i], evy[i], stax[w],  stay[w])
        for(j in 1:length(w))
          {
        RAP = get2Drayblox(evx[i], evy[i],stax[w[j]] ,stay[w[j]] ,
          xo, yo, NODES=TRUE, PLOT=FALSE)
        slns = MOD[cbind(RAP$ix, RAP$iy)]
        tt = slns*(RAP$lengs)
        k = k+1
        COV[[k]] = list(RAP=RAP, tt=tt)
```

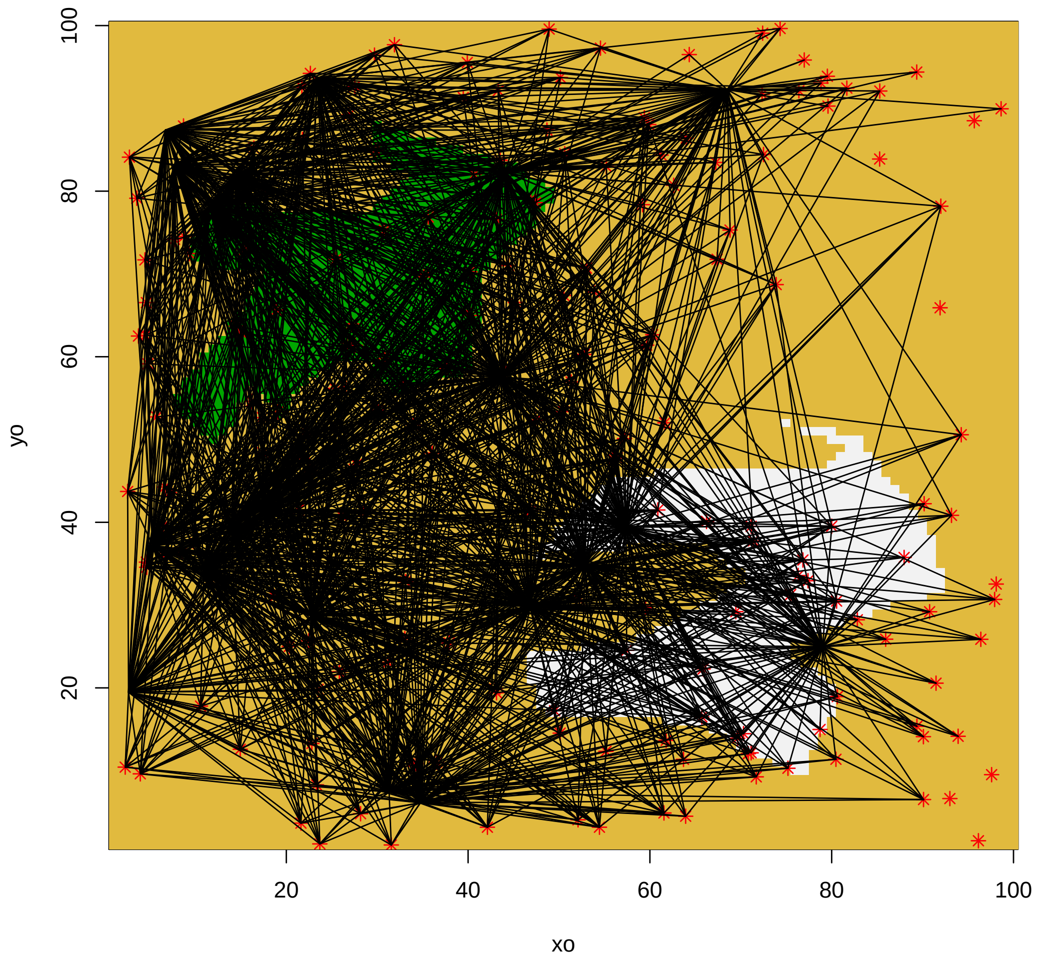

Figure 8.5: Raypath Distribution

```
      }

      }

    }
```

Add Noise to the data:

```
####image(xo, yo, ZEE, col=terrain.colors(100) )

#### points(stax, stay, pch=6, col='blue')

###################################################
###################################################
###################################################

NOISE = 0.001
    for(i in 1:length(COV))
      {

       COV[[i]]$noise = rnorm(n=1, m=0, sd=NOISE)


      }

###################################################
```

Now the synthetic data has been created. The data is stored in memory as lists.

The list named *COV* has the coverage information. That is the travel times and the raypath coverage. The travel times are in a vector called *tt*. these are the ‘true’ arrival times. The raypaths are stored in a second list called *RAP*. It has the coordinated of each (straight) raypath transecting the model.

8.11 Inversion by Backprojection

Here we solve the problem,

$$\Delta\vec{t} = \vec{\vec{A}}\vec{s}$$

Here we solve the tomography problem via iterative backprojection. This is the algorithm that I believe can be made parallel.

8.11.1 Inversion by backprojection

Do 30 iterations, include noise. At each step use the current model and upgrade the velocity.

```
######
PHANT =  matrix(0, ncol=NX, nrow=NY)
for(j in 1:30)
  {
    print(paste("Iteration", j))
    for(i in 1:length(COV))
      {
        ###
        oldsln = as.vector(PHANT[cbind(COV[[i]]$RAP$ix, COV[[i]]$RAP$iy)])
        slns = (1/V)*(1+oldsln)
        t1 = slns*COV[[i]]$RAP$lengs
        DT = (COV[[i]]$tt - t1) + COV[[i]]$noise

        PHANT[cbind(COV[[i]]$RAP$ix, COV[[i]]$RAP$iy)] =
          PHANT[cbind(COV[[i]]$RAP$ix, COV[[i]]$RAP$iy)]+
            DT*COV[[i]]$RAP$lengs/(sum(COV[[i]]$RAP$lengs))


      }

  }
```

To illustrate, here is inversion by backprojection using only 1 iteration:

```
###################################################

PHANT1 =  matrix(0, ncol=NX, nrow=NY)
for(i in 1:length(COV))
  {

    t1 =(1/V)*COV[[i]]$RAP$lengs
    DT = (COV[[i]]$tt - t1) + COV[[i]]$noise

    PHANT1[cbind(COV[[i]]$RAP$ix, COV[[i]]$RAP$iy)] =
      PHANT1[cbind(COV[[i]]$RAP$ix, COV[[i]]$RAP$iy)]+
        DT*COV[[i]]$RAP$lengs/(sum(COV[[i]]$RAP$lengs))


  }


##  screens(2)


##dev.set(dev.next())
```

Display results: This is the original phantom:

```
image(xo, yo, ZEE, col=tomo.colors(100) )
polygon(A1)
polygon(A2)
```

This is the backprojected image with only one iteration:

```
## dev.set(dev.next())

image(xo, yo, t(PHANT1), col=tomo.colors(100) )
polygon(A1)
```

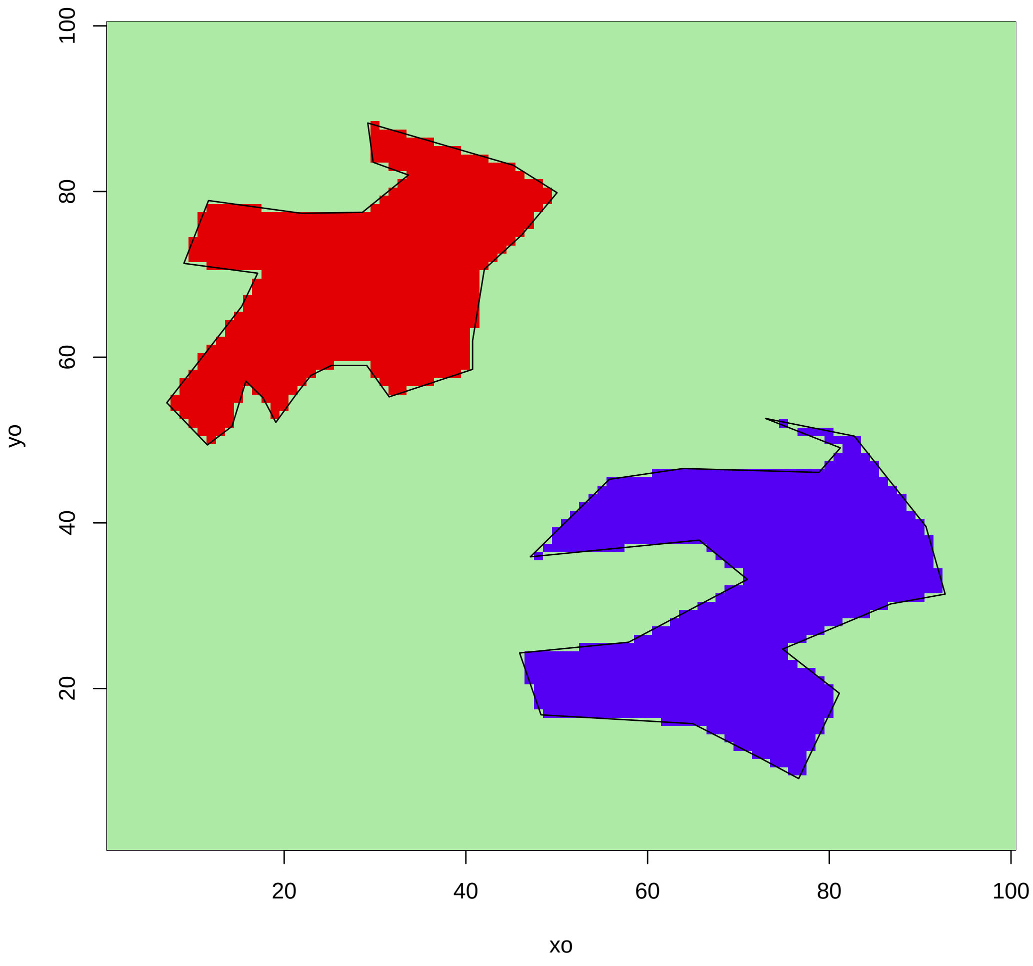

Figure 8.6: Synthetic phantom - this is the "Truth"

```
polygon(A2)
```

This is the backprojected image:

```
## dev.set(dev.next())
image(xo, yo, t(PHANT), col=tomo.colors(100) )
```

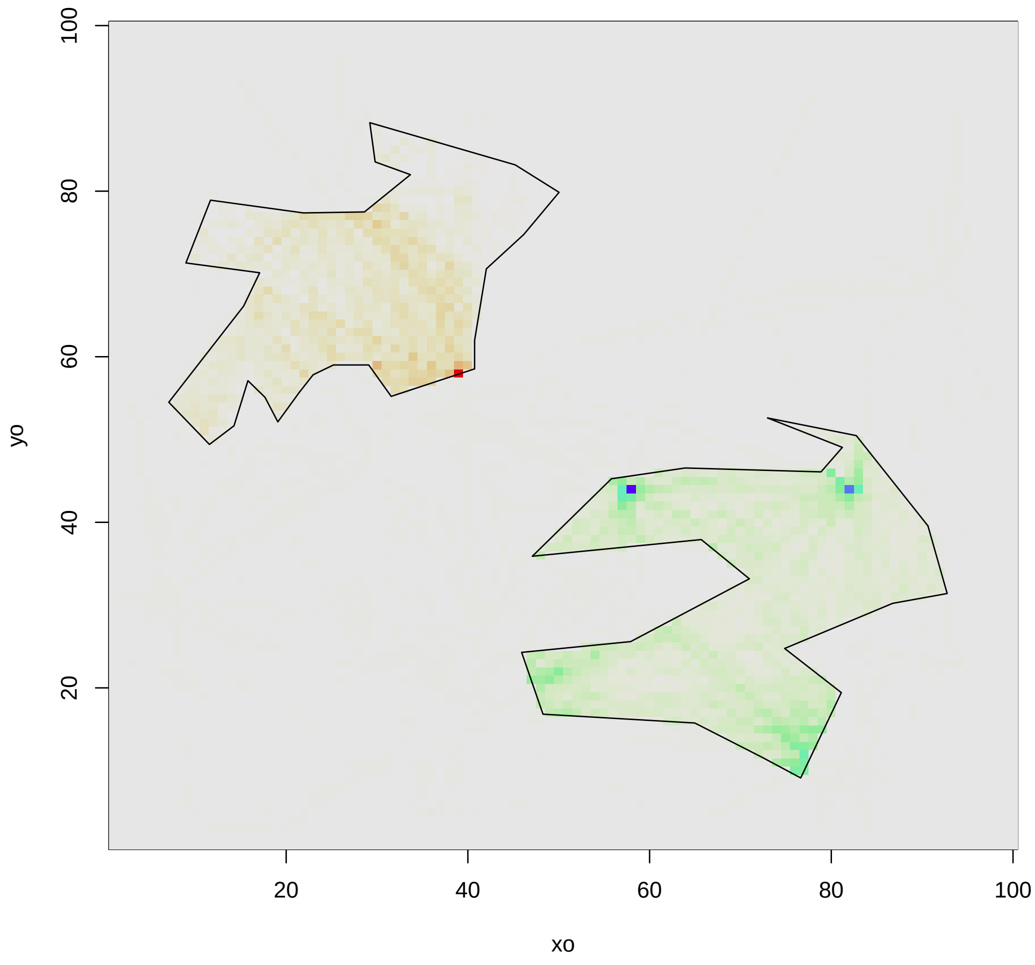

Figure 8.7: Tomographic inversion result with one backprojected iteration.

```
polygon(A1)
polygon(A2)
```

8.12 Modifications and Improvements

8.12.1 Damping or Regularization

The solution achieved through solving the normal equations may work, but it may fail since noise in the data corrupts the solution. Some kind of damping should be applied prior to solving equations in the manner. This is accomplished by adding additional constraint equations to the original matrix inversion problem. The original formula,

$$\vec{A}\vec{x} = \vec{b}$$

becomes

$$\begin{bmatrix} \vec{A} \\ \vec{\Theta} \end{bmatrix} \vec{x} = \begin{bmatrix} \vec{b} \\ \vec{0} \end{bmatrix}$$

Where Θ is a matrix of constraints, as described above in the discussion of damping as

$$\vec{\Theta} = \begin{bmatrix} \theta & 0 & 0 & \cdots \\ 0 & \theta & 0 & \cdots \\ 0 & 0 & \theta & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \theta\vec{I}$$

There is no analytical way to determine the correct damping θ. This should be done by trial and error, or through experience etc. The damping can be thought of as some kind of *a priori* information constraining the inversion. It is saying, in effect, allow perturbations, but do not allow them to be too large. How large is “too large”? There in lies the problem. Since damping is required because of potentially destabalizing effects of noise, it may be possible to estimate a good damping parameter based on estiamted uncertainty in arrival time picks.

8.12.2 Weighting

The equations may (should) not have all the same importance. For example, stations much further away will experience more attenuation and arrival time determination may be more prone to uncertainty. Some

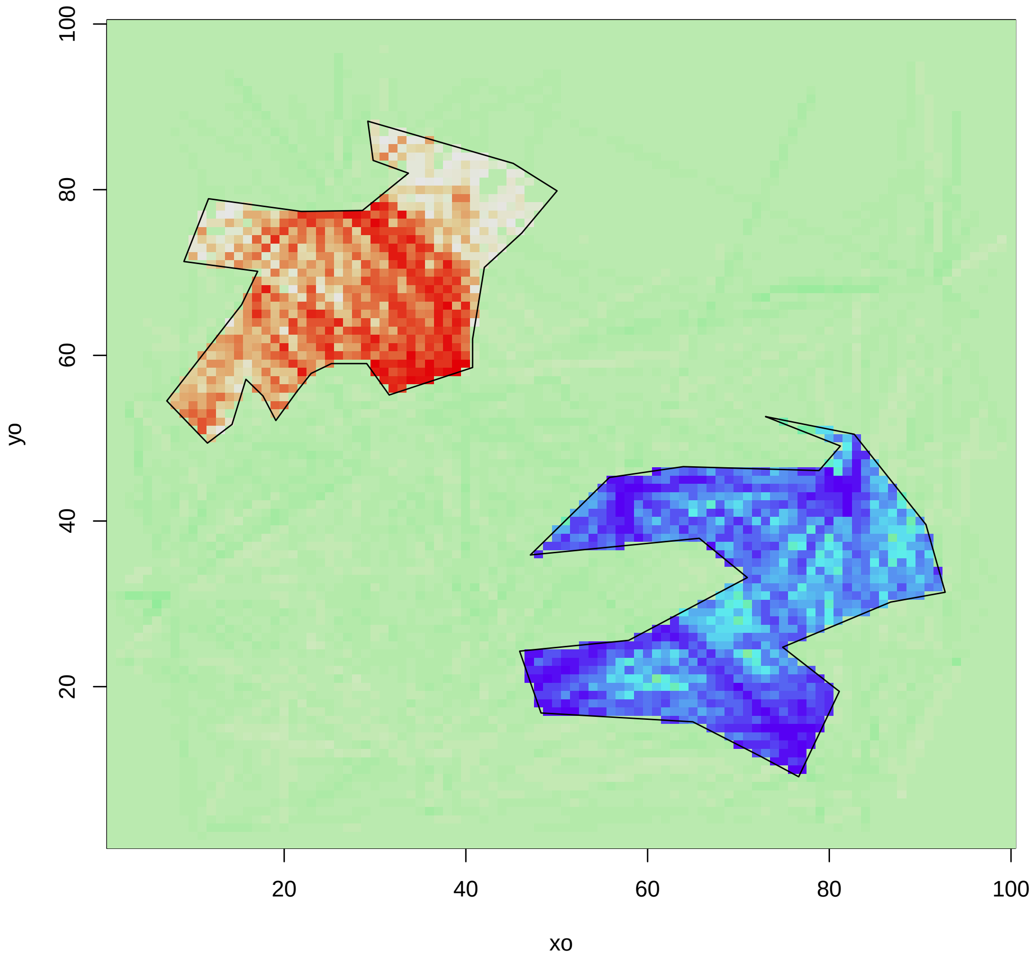

Figure 8.8: Tomographic inversion result, 30 iterations. This is the image seen through the inversion process.

stations may have known problems or may be located on sites that consistently provide poor arrival time picks. A weighting matrix $\vec{W}$ is then introduced and mulitplied to account for variable quality of data:

$$\begin{bmatrix} \vec{W}\vec{A} \\ \vec{\Theta} \end{bmatrix} \vec{x} = \begin{bmatrix} \vec{W}\vec{b} \\ \vec{0} \end{bmatrix}$$

This set of equations may be solved using a variety of methods including QR-Decomposition, cholesky decomposition, singular value decomposition, etc.... Note that the inversion is repeated for each step in the nonlinear convergence process, i.e. each time the event is shifted a new matrix is formed and a new, perturbed location is derived.

Chapter 9

Ray Tracing

9.1 Raytracing

```
> options(width=60)
> options(prompt=" ")
 options(continue=" ")
 options(SweaveHooks=list(fig=function()
 par(mar=c(5.1, 4.1, 1.1, 2.1))))
 JPOST<-function(file="tmp", width = 8, height = 8)
   {
     postscript(file=file, width = width,
            height = height, paper = "special",
         horizontal = FALSE, onefile = TRUE, print.it = FALSE)
 }
 library(RSEIS)
 library(Rquake)
```

9.2 Rquake

In this document I will illustrate how to use **Rquake** , a non-linear earthquake location program.

9.3 Data Structures and Lists

9.3.1 Station File

Station location information can be stored in memory (in a list) or in a text file on disk. The station file is a table, with name, lat, lon, and elevation.

For example:

```
fsta = "/home/lees/Site/CHAC/staLLZ.txt"
###  system(paste(sep=" ", "cat", fsta), intern = TRUE )
```

```
CHAC0   -0.39377412    -78.15369741 3588
CHAC1   -0.366526404   -78.16962049  3606
CHAC2   -0.42485567    -78.2710065   4020
CHAC3   -0.4524493    -78.18676153    4328
CHAC4   -0.461317213   -78.21783387   4412
CHAC5   -0.351938598   -78.21809574   4000
CHAC6   -0.408928292   -78.20667762    3860
CHAC7   -0.39837847    -78.22075601    4109
CHAC8   -0.382639731  -78.2023599    3767
CHAC9   -0.323852103  -78.15061344  3762
```

These can be scanned in **R** with a simple command.

See **REIS** for more details on stations.

If the stations are in UTM coordinates, you may convert to Lat-Lon using the GEOmap package.

```
stas = scan(file=fsta,what=list(name="", lat=0, lon=0, z=0))
stas$z = stas$z/1000
```

Units in Rquake are in km, so the meters are converted.

REIS has a function for reading in the stations:

```
stas = setstas("stas")
```

9.3.2 Velocity Structure

The one-dimensional velocity model is also stored in file (or stored in memory in an **R** session). See **REIS** for details.

Sample velocity model stored on disk. In this case no estimates of error are provided, so they are set to zero. If S-wave velocity is not available, can use $V_s = V_p/\sqrt{3}$.

```
#MODEL WU COSO REGINAL FINE LAYERS REGIONAL VELOCITY MODEL
#P DEPTH   P VEL      PERR      S DEPTH    S VEL    SERR
   0.00     4.50      0.00      0.00        2.43    0.00
   0.50     4.51      0.00      0.50        2.59    0.00
   1.00     4.92      0.00      1.00        2.97    0.00
   1.50     4.92      0.00      1.50        2.97    0.00
   2.00     5.46      0.00      2.00        3.15    0.00
   2.50     5.46      0.00      2.50        3.15    0.00
   3.00     5.54      0.00      3.00        3.27    0.00
   3.50     5.54      0.00      3.50        3.27    0.00
   4.00     5.58      0.00      4.00        3.42    0.00
   5.50     5.58      0.00      5.50        3.42    0.00
  12.00     6.05      0.00     12.00        3.49    0.00
  20.00     7.20      0.00     20.00        4.15    0.00
```

The following is a constructor for making a 1D velocity model suitable for use in RSEIS and Rquake:

```
 VEL=list()
     VEL$'zp'=c(0,0.25,0.5,0.75,1,2,4,5,10,12)
     VEL$'vp'=c(1.1,2.15,3.2,4.25,5.3,6.25,6.7,6.9,7,7.2)
     VEL$'ep'=c(0,0,0,0,0,0,0,0,0,0)
     VEL$'zs'=c(0,0.25,0.5,0.75,1,2,4,5,10,12)
     VEL$'vs'=c(0.62,1.21,1.8,2.39,2.98,3.51,3.76,3.88,3.93,4.04)
     VEL$'es'=c(0,0,0,0,0,0,0,0,0,0)
     VEL$'name'='/data/wadati/lees/Site/Hengil/krafla.vel'
```

There are several default velocity models available in **REIS** . Function defaultVEL(i) will return one of 6 “standard” models used for different purposes.

If you have a velocity model on disk, you can read it in with **REIS** function, Get1Dvel.

To compare a set of different velocity models visually, try,

```
data(ASW.vel)
      data(wu_coso.vel)
      data(fuj1.vel)
      data(LITHOS.vel)
```

These can be plotted with the routine:

```
Comp1Dvels(c("ASW.vel","wu_coso.vel",  "fuj1.vel" , "LITHOS.vel"  ))
```

9.3.3 Arrival Time List

The arrival times, or the picks are stored in in list mode, i.e. a list of vectors each with attributes relating to the arrival time pick.

These vectors are described as:

tag character tag the should be unique

name character, station name

comp character, component name

c3 character, three-component station id sta.hhh.BHZ

phase character, phase name

err numeric, error

pol character polarity, U, D, 0

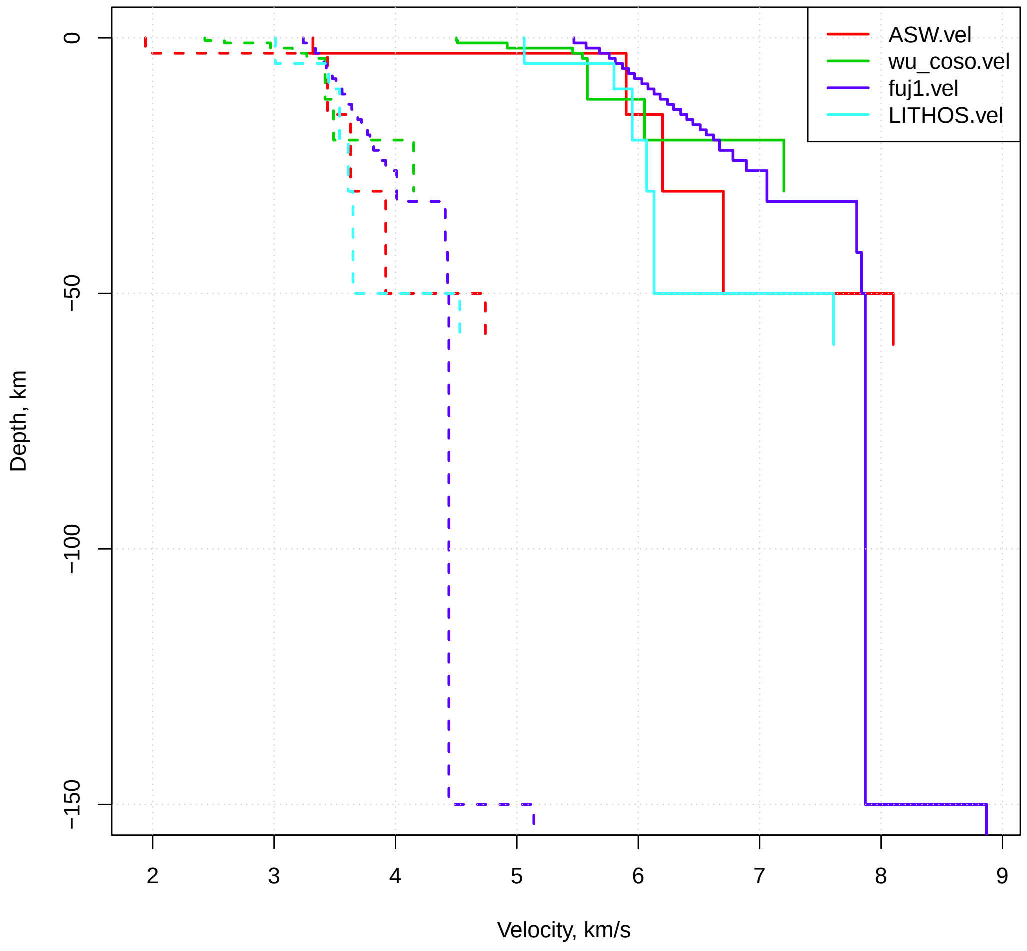

Figure 9.1: Comparison of 4 sample velocity models

flg numeric, flag, used in location

res numeric, travel time residual relative to model

dur numeric, duration

yr numeric, year

mo numeric, month

dom numeric, day-of-month

jd numeric, julian day

hr numeric, hour

mi numeric, minute

sec numeric, second

col numeric, or character, color for plotting in RSEIS

onoff numeric, less than 0 means do not use

A constructor for creating an empty pick list is cleanWPX. For many of the functions in RSEIS and Rquake the list must contain filled vectors for each element. use function repairWPX to fill out list elements that are deficient.

The arrival time list has one attribute, the “ID”. This can be used to identify earthquake with a unique tag or identifaication number or name.

For **Rquake** , the elements that are absolutely *required* are: name, phase, err, sec.

There are many different ways to store arrival time picks. It does not matter how these are stored, as long as they are read into R and formatted properly. By disassociating the input format from the analysis, we can simply write a short input, or conversion, routine to use all the codes as is.

We can thus store the data in any format we desire, perhaps for use in other non-R software.

Native (binary) R

The output of **swig** is binary R file, so the data can simply be loaded automatically.

UW format Pickfiles

loadUWpickfiles is a function that reads in a list of pickfiles stored on disk and returns a list of picked events.

Since UW pickfiles store the times relative to a common minute mark, and station information is not stored in the pickfile, this information is filled out in the code:

```
KF = vector(mode="list")
   for(i in 1:length(LF))
     {
       g1 = getpfile(LF[i])
       m1 = match(g1$STAS$name, stas$name)
       g1$STAS$lat = stas$lat[m1]
       g1$STAS$lon = stas$lon[m1]
       g1$STAS$z = stas$z[m1]
       w1 = which(!is.na(g1$STAS$lat))
       sec = g1$STAS$sec[w1]
       N = length(sec)

       Ldat = list(name = g1$STAS$name[w1],
         sec = g1$STAS$sec[w1],
         phase = g1$STAS$phase[w1],
         lat = g1$STAS$lat[w1],
         lon = g1$STAS$lon[w1],
         z = g1$STAS$z[w1],
         err = g1$STAS$err[w1],
         yr = rep(g1$LOC$yr, times = N),
         jd = rep(g1$LOC$jd, times = N),
         mo = rep(g1$LOC$mo, times = N),
         dom = rep(g1$LOC$dom, times = N),
         hr = rep(g1$LOC$hr, times = N),
         mi = rep(g1$LOC$mi, times = N))

       Ldat$err[Ldat$err <= 0] = 0.05
       Ksta = length(unique(Ldat$name))
      ###  cat(paste("################      ", i, Ksta), sep = "\n")
       Ldat = LeftjustTime(Ldat)

       KF[[i]] = Ldat
```

```
    }
```

CSV Pickfiles

Chapter 10

Scattering

10.1 Zoeppritz Equations

10.2 Zoeppritz Equations

When a seismic wave impinges on an flat interface some of the energy is transmitted into the layer below and sme of the energy is reflected back. The relative proportion of energy converted into different aspects of P and S-wave can be decribed by a a set of equations set forth in the classic text book by Aki and Richards [?]Aki2002). Zoeppritz are also known as Knott's Equations.

To calculate the associated matrix of incoming and outgoing amplitudes with different incident angles, first we set up the velocity model at the interface. These are the α, β, ρ illustrated in figure 10.1 above and below the interface.

```
##############   set up a velocity model

### layer 1:
library(zoeppritz)
alpha1 = 4.98
  beta1 =  2.9
   rho1 = 2.667
##########  layer 2
```

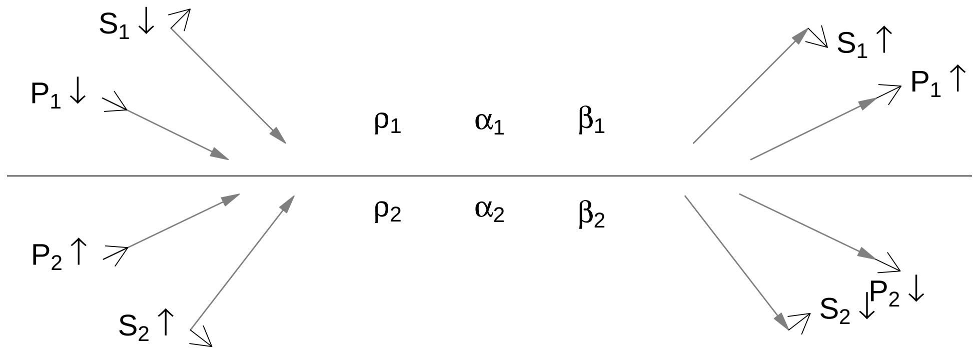

Figure 10.1: Figure from Aki and Richards showing incoming P and SV waves to the interface.

```
alpha2 = 8.0
 beta2 = 4.6
 rho2 = 3.38
```

Then the program is called that calculates the scattering coefficients and plots the result:

```
App =  pzoeppritz( "Amplitude" , alpha1, alpha2, beta1,
  beta2, rho1 ,rho2, "P", "ALL");
```

```
dev.off()
```

Next we change the outgoing waves to S-waves:

Incident wave in high velocity layer

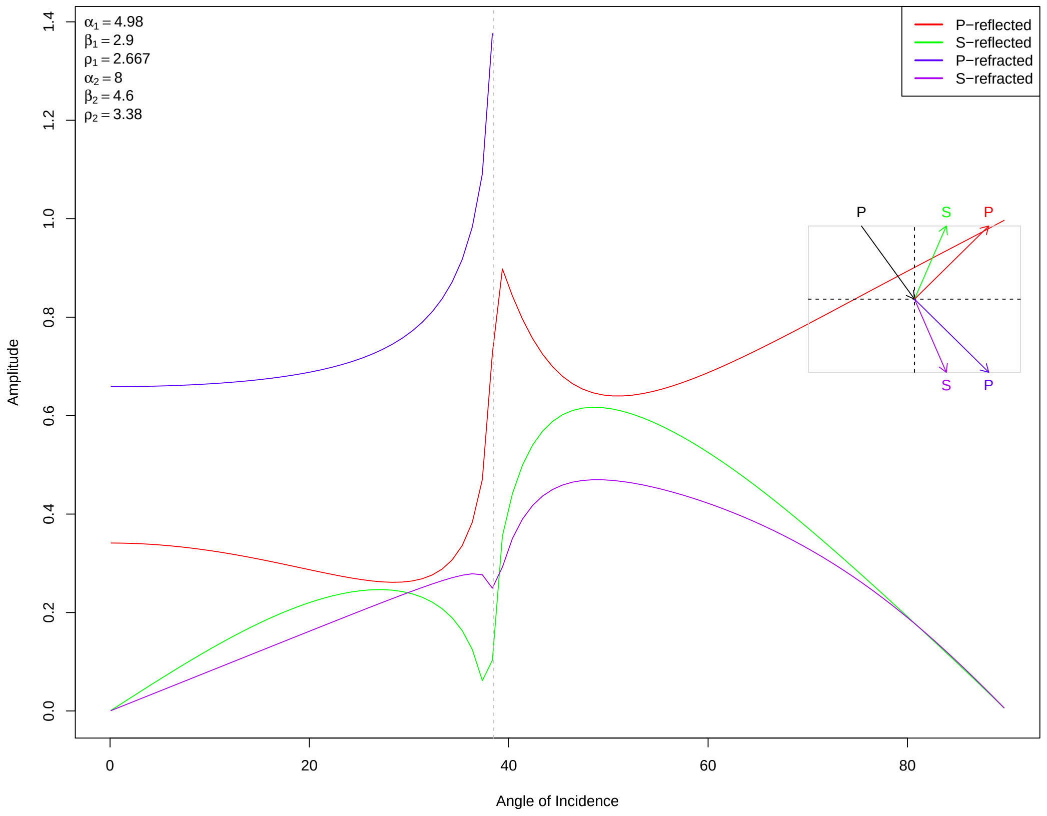

Figure 10.2: P-wave incoming, show all the outgoing amplitudes.

Next we change the outgoing waves to S-waves:

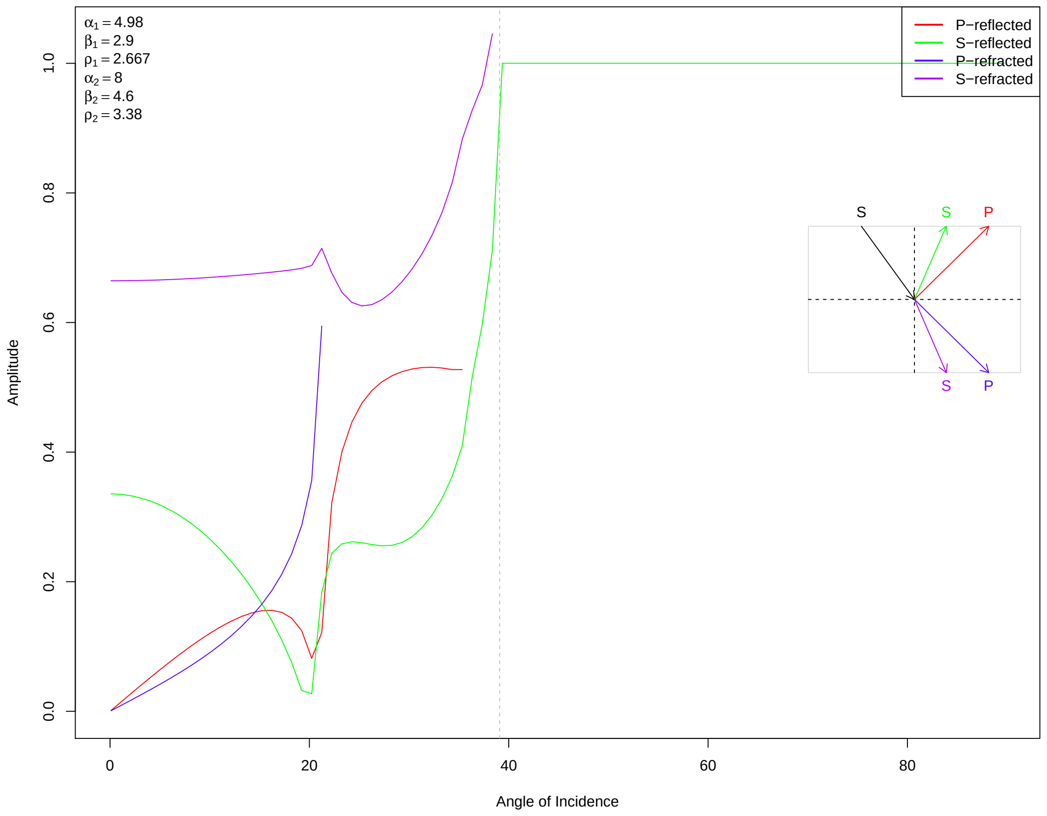

Figure 10.3: S-wave incoming, show all the outgoing amplitudes.

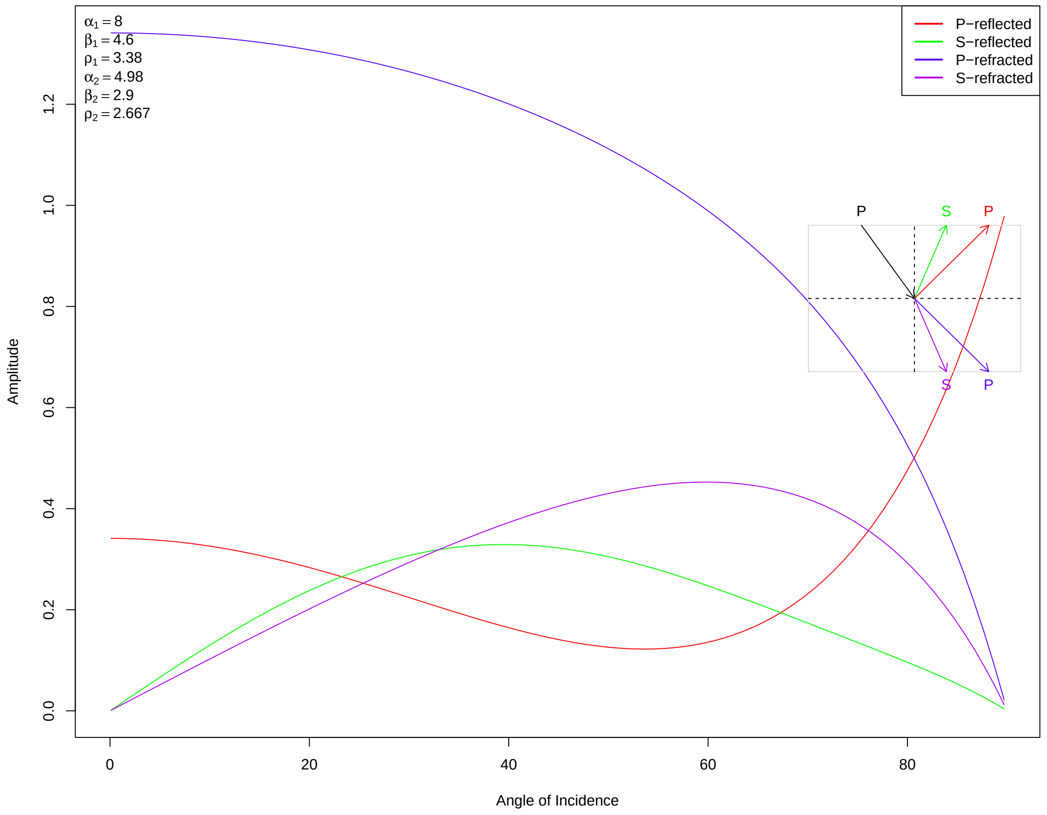

Figure 10.4: High velocity above low velocity. P-wave incoming, show all the outgoing amplitudes.

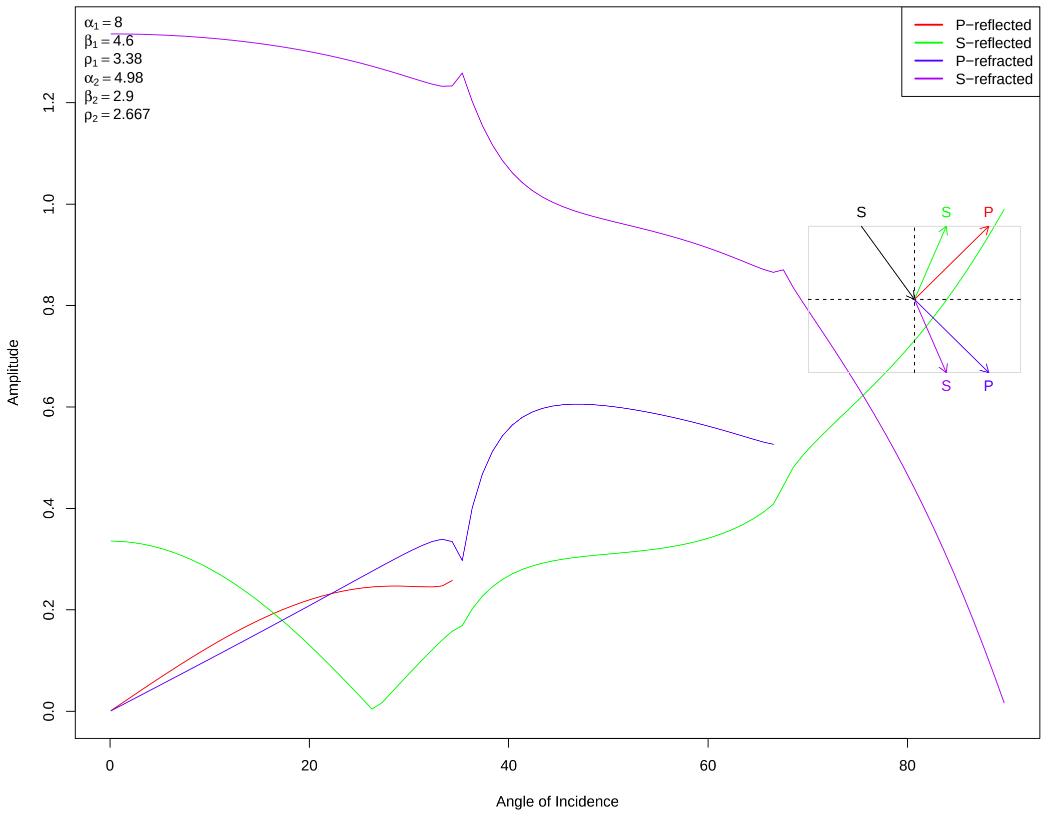

Figure 10.5: High velocity above low velocity. S-wave incoming, show all the outgoing amplitudes.