SeisIO Documentation

Release 0.3.0

Joshua Jones

Aug 02, 2019

INTRODUCTION

SeisIO is a collection of utilities for reading and downloading geophysical timeseries data.

CHAPTER

ONE

INTRO

1.1 Introduction

SeisIO is a framework for working with univariate geophysical data. SeisIO is designed around three basic principles:

- Ease of use: you shouldn't *need* a PhD to study geophysical data.
- Fluidity: working with data shouldn't feel *clumsy*.
- Performance: speed and efficient memory usage *matter*.

The project is home to an expanding set of web clients, file format readers, and analysis utilities.

1.1.1 Overview

SeisIO stores data in minimalist containers that track the bare necessities for analysis. New data are easily added with basic commands like +. Unwanted channels can be removed just as easily. Data can be saved to a native SeisIO format or written to other supported file formats.

1.1.2 Installation

From the Julia prompt: press `]` to enter the Pkg environment, then type

```
add SeisIO; build; precompile
```

Dependencies should install automatically. To verify that everything works correctly, type

```
test SeisIO
```

and allow 10-20 minutes for tests to complete. To get started, exit the Pkg environment by pressing Backspace or Control + C, then type

```
using SeisIO
```

at the Julia prompt. You'll need to repeat that last step whenever you restart Julia, as with any command-line interpreter (CLI) language.

1.1.3 Getting Started

The *tutorial* is designed as a gentle introduction for people less familiar with the Julia language. If you already have some familiarity with Julia, you probably want to start with one of the following topics:

- *Working with Data*: learn how to manage data using SeisIO
- *Reading Data*: learn how to read data from file
- *Requesting Data*: learn how to make web requests

1.1.4 Updating

From the Julia prompt: press `]` to enter the Pkg environment, then type `update`. Once updates finish, restart Julia to use them.

1.1.5 Getting Help

In addition to these documents, a number of help documents can be called at the Julia prompt. These commands are a useful starting point:

```
?chanspec           # how to specify channels in web requests
?get_data           # how to download data
?read_data          # how to read data from file
?timespec           # how to specify times in web requests and data processing
?seed_support       # how much of the SEED data standard is supported?
?seis_www           # list strings for data sources in web requests
?SeisData           # information about SeisIO data types
?SeisIO.KW          # SeisIO shared keywords and their meanings
```

1.2 First Steps

SeisIO is designed around easy, fluid, and fast data access. At the most basic level, SeisIO uses an array-like custom object called a **SeisChannel** for single-channel data; **SeisData** objects store multichannel data and can be created by combining SeisChannels.

1.2.1 Start Here

Create a new, empty **SeisChannel** object with

`Ch = SeisChannel()`

The meanings of the field names are explained *here*; you can also type `?SeisChannel` at the Julia prompt. You can edit field values manually, e.g.,

```
Ch.loc = [-90.0, 0.0, 9300.0, 0.0, 0.0]
Ch.name = "South pole"
```

or you can set them with keywords at creation:

```
Ch = SeisChannel(name="MANOWAR JAJAJA")
```

SeisData structures are collections of channel data. They can be created with the SeisData() command, which can optionally create any number of empty channels at a time, e.g.,

`S = SeisData(1)`

They can be explored similarly:

```
S.name[1] = "South pole"
S.loc[1] = [-90.0, 0.0, 9300.0, 0.0, 0.0]
```

A collection of channels becomes a SeisData structure:

`S = SeisData(SeisChannel(), SeisChannel())`

You can push channels onto existing SeisData structures, like adding one key to a dictionary:

`push!(S, Ch)`

Note that this copies Ch to a new channel in S – S[3] is not a view into C. This is deliberate, as otherwise the workspace quickly becomes a mess of redundant channels. Clean up with `Ch = []` to free memory before moving on.

1.2.2 Operations on SeisData structures

We're now ready for a short tutorial of what we can do with data structures. In the commands below, as in most of this documentation, **Ch** is a SeisChannel object and **S** is a SeisData object.

Adding channels to a SeisData structure

You've already seen one way to add a channel to SeisData: push!(S, SeisChannel()) adds an empty channel. Here are others:

`append!(S, SeisData(n))`

Adds n channels to the end of S by creating a new n-channel SeisData and appending it, similar to adding two dictionaries together.

These methods are aliased to the addition operator:

```
S += SeisChannel()       # equivalent to push!(S, SeisChannel())
S += randseisdata(3)     # adds a random 3-element SeisData structure to S in␣
↪place
S = SeisData(randseisdata(5), SeisChannel(),
      SeisChannel(id="UW.SEP..EHZ", name="Darth Exploded",
      loc=[46.1967, -122.1875, 1440, 0.0, 0.0]))
```

Most web request functions can append to an existing SeisData object by placing an exclamation mark after the function call. You can see how this works by running the *examples*.

Search, Sort, and Prune

The easiest way to find channels of interest in a data structure is to use findid, but you can obtain an array of partial matches with findchan:

```
S = SeisData(randseisdata(5), SeisChannel(),
      SeisChannel(id="UW.SEP..EHZ", name="Darth Exploded",
      loc=[46.1967, -122.1875, 1440, 0.0, 0.0], x=rand(1024)))
findid(S, "UW.SEP..EHZ")     # 7
findchan(S, "EHZ")           # [7], maybe others depending on randseisdata
```

You can sort by channel ID with the *sort* command.

Several functions exist to prune empty and unwanted channels from SeisData structures.

```
delete!(S, 1:2)  # Delete first two channels of S
S -= 3           # Delete third channel of S

# Extract S[1] as a SeisChannel, removing it from S
C = pull(S, 1)

# Delete all channels whose S.x is empty
prune!(S)

# Delete channels containing ".SEP."
delete!(S, ".SEP.", exact=false)
```

In the last example, specifying exact=false means that any channel whose ID partly matches the string ".SEP." gets deleted; by default, passing a string to delete!(S, str) only matches channels where str is the exact ID. This is an efficient way to remove unresponsive subnets and unwanted channel types, but beware of clumsy over-matching.

1.2.3 Next Steps

Because tracking arbitrary operations can be difficult, several functions have been written to keep track of data and operations in a semi-automated way. See the next section, *working with data*, for detailed discussion of managing data from the Julia command prompt.

1.3 Working with Data

This section describes how to track and manage SeisIO data.

1.3.1 Creating Data Containers

Create a new, empty object using any of the following commands:

Object	Purpose
SeisChannel()	A single channel of univariate (usually time-series) data
SeisData()	Multichannel univariate (usually time-series) data
SeisHdr()	Header structure for discrete seismic events
SeisEvent()	Discrete seismic events; includes SeisHdr and SeisData objects

1.3.2 Acquiring Data

- Read files with *read_data*
- Make web requets with *get_data*
- Initiate real-time streaming sessions to SeisData objects with *SeedLink!*

1.3.3 Keeping Track

A number of auxiliary functions exist to keep track of channels:

`findchan`(*id::String*, *S::SeisData*)

`findchan`(*S::SeisData*, *id::String*)

Get all channel indices i in S with id ∈ S.id[i]. Can do partial id matches, e.g. findchan(S, "UW.") returns indices to all channels whose IDs begin with "UW.".

`findid`(*S::SeisData*, *id*)

Return the index of the first channel in **S** where id = **id**. Requires an exact string match; intended as a low-memory equivalent to findfirst for ids.

`findid`(*S::SeisData*, *Ch::SeisChannel*)

Equivalent to findfirst(S.id.==Ch.id).

`namestrip!(S[, convention])`

Remove bad characters from the :name fields of **S**. Specify convention as a string (default is "File"):

Convention	Characters Removed:sup:*(a)*
"File"	`"$*/:<>?@\^|~DEL`
"HTML"	`"&';<>©DEL`
"Julia"	`$\DEL`
"Markdown"	`!#()*+-.[\]_`{}`
"SEED"	`.DEL`
"Strict"	`!"#$%&'()*+,-./:;<=>?@[\]^`{|}~DEL`

[(a)] `DEL` here is \x7f (ASCII/Unicode U+007f).

`timestamp`()

Return current UTC time formatted yyyy-mm-ddTHH:MM:SS.uuu.

track_off!(S)

Turn off tracking in S and return a boolean vector of which channels were added or altered significantly.

track_on!(S)

Begin tracking changes in S. Tracks changes to :id, channel additions, and changes to data vector sizes in S.x.

Does not track data processing operations on any channel i unless length(S.x[i]) changes for channel i (e.g. filtering is not tracked).

Warning: If you have or suspect gapped data in any channel, calling ungap! while tracking is active will flag a channel as changed.

Source Logging

The :src field records the *last* source used to populate each channel (usually a file name and path or a web request URL).

When a data source is added to a channel, including the first time data are added, this is recorded in :notes with the syntax (timestamp) +src: (function) (src).

1.3.4 Channel Maintenance

A few functions exist specifically to simplify data maintenance:

prune!(S::SeisData)

Delete all channels from S that have no data (i.e. S.x is empty or non-existent).

C = pull(S::SeisData, id::String)

Extract the first channel with id=id from S and return it as a new SeisChannel structure. The corresponding channel in S is deleted.

C = pull(S::SeisData, i::integer)

Extract channel **i** from **S** as a new SeisChannel object **C**, and delete the corresponding channel from **S**.

1.3.5 Taking Notes

Functions that add and process data note these operations in the :notes field of each object affected. One can also add custom notes with the note! command:

note!(S, i, str)

Append **str** with a timestamp to the :notes field of channel number **i** of **S**.

note!(S, id, str)

As above for the first channel in **S** whose id is an exact match to **id**.

note!(S, str)

if **str* mentions a channel name or ID, only the corresponding channel(s) in **S** is annotated; otherwise, all channels are annotated.

Clear all notes from channel `i` of `S`.

```
clear_notes!(S, id)
```

Clear all notes from the first channel in `S` whose id field exactly matches `id`.

```
clear_notes!(S)
```

Clear all notes from every channel in `S`.

CHAPTER

TWO

FILES

2.1 Reading Files

```
read_data!(S, fmt::String, filepat [, KWs])
S = read_data(fmt::String, filepat [, KWs])
```

Read data from a supported file format into memory.

fmt
Lowercase string describin the file format. See below.

filepat
Read files with names matching pattern `filepat`. Supports wildcards.

KWs
Keyword arguments; see also *SeisIO standard KWs* or type `?SeisIO.KW`.
Standard keywords: full, nx_add, nx_new, v
Other keywords: See below.

2.1.1 Supported File Formats

File Format	String
GeoCSV, time-sample pair	geocsv
GeoCSV, sample list	geocsv.slist
Lennartz ASCII	lenartzascii
Mini-SEED, SEED	mseed
PASSCAL SEG Y	passcal
SAC	sac
SEG Y (rev 0 or rev 1)	segy
UW	uw
Win32	win32

Strings are case-sensitive to prevent any performance impact from using matches and/or lowercase().

2.1.2 Supported Keywords

KW	Used By	Type	Default	Meaning
cf	win32	String	""	win32 channel info filestr
full	sac	Bool	false	read full header into **:misc**?
	segy			
	uw			
nx_add	mseed	Int64	360000	minimum size increase of **:x**
nx_new	mseed	Int64	86400000	length of **:x** for new channels
jst	win32	Bool	true	are sample times JST (UTC+9)?
swap	seed	Bool	true	byte swap?
v	mseed	Int64	0	verbosity
	uw			
	win32			

Performance Tip

With mseed or win32 data, adjust *nx_new* and *nx_add* based on the sizes of the data vectors that you expect to read. If the largest has *Nmax* samples, and the smallest has *Nmin*, we recommend *nx_new=Nmin* and *nx_add=Nmax-Nmin*.

Default values can be changed in SeisIO keywords, e.g.,

```
SeisIO.KW.nx_new = 60000
SeisIO.KW.nx_add = 360000
```

The system-wide defaults are *nx_new=86400000* and *nx_add=360000*. Using these values with very small jobs will greatly decrease performance.

Examples

1. **`S = read_data("uw", "99011116541W", full=true)`**
    - Read UW-format data file `99011116541W`
    - Store full header information in `:misc`
2. **`read_data!(S, "sac", "MSH80*.SAC")`**
    - Read SAC-format files matching string pattern *MSH80*.SAC*
    - Read into existing SeisData object `S`
3. **`S = read_data("win32", "20140927*.cnt", cf="20140927*ch", nx_new=360000)`**

- Read win32-format data files with names matching pattern `2014092709*.cnt`
- Use ASCII channel information filenames that match pattern `20140927*ch`
- Assign new channels an initial size of `nx_new` samples

2.1.3 Format Descriptions and Notes

GeoCSV: an extension of "human-readable", tabular file format Comma- Separated Values (CSV).

Lennartz ASCII: ASCII output of Lennartz portable digitizers.

PASSCAL: A single-channel variant SEG Y format developed by PASSCAL/New Mexico Tech and commonly used with PASSCAL field equipment. PASSCAL differs from SEG Y in that PASSCAL format uses neither file headers nor extended textural headers, and the number of samples per trace can exceed 32767.

SEED: SEED stands for Standard for the Exchange of Earthquake Data; used by the International Federation of Digital Seismograph Networks (FDSN) as an omnibus seismic data standard. mini-SEED is a data-only variant that uses only data blockettes and allows longer data records.

SAC: the Seismic Analysis Code data format, originally developed for the eponymous command-line interpreter. Widely used, and supported in virtually every programming language.

SEG Y: Society of Exploration Geophysicists data format. Common in the energy industry, developed and maintained by the SEG. Only SEG Y rev 0 and rev 1 with standard headers are supported. Note that IBM Float support is not yet implemented.

UW: the University of Washington data format has no online reference and is no longer in use. Created by the Pacific Northwest Seismic Network for event archival; used from the 1970s through early 2000s. A UW event is described by a pickfile and corresponding data file, whose names are identical except for the last character; for example, files 99062109485o and 99062109485W describe event 99062109485. Unlike the win32 data format, the data file is self-contained; the pick file is not required to use raw trace data. However, like the win32 data format, instrument locations are stored in an external human-maintained text file. Only UW-2 data files are supported by SeisIO; we have never encountered a UW-1 data file outside of Exabyte tapes and have no reason to suspect that any remain in circulation.

Win32: data format developed by the NIED (National Research Institute for Earth Science and Disaster Prevention), Japan. Data are typically divided into files that contain a minute of continuous data from several channels. Data within each file are stored by channel in one-second segments as variable-precision delta-encoded integers. Channel information is retrieved from an external channel information file. Although accurate channel files are needed to use win32 data, these files are not strictly controlled by any central authority and inconsistencies in channel parameters, particularly gains, are known to exist.

2.1.4 Other File I/O Functions

`rseis` (*fname*)

Read SeisIO native format data into an array of SeisIO structures.

`sachdr` (*fname*)

Print headers from SAC file to stdout.

segyhdr (*fname*[, *PASSCAL=true::Bool*])

Print headers from SEG Y file to stdout. Specify `passcal=true` for PASSCAL SEG Y.

uwdf (*dfname*)

Parse UW event data file `dfname` into a new SeisEvent structure.

writesac (*S*[, *ts=true*])

Write SAC data to SAC files with auto-generated names. Specify ts=true to write time stamps; this will flag the file as generic x-y data in the SAC interpreter.

wseis (*fname*, *S*)

wseis (*fname*, *S*, *T*, *U...*)

Write SeisIO data to fname. Multiple objects can be written at once.

CHAPTER

THREE

WEB

3.1 Web Requests

Data requests use `get_data!` for FDSN or IRIS data services; for (near) real-time streaming, see *SeedLink*.

```
get_data!(S, method, channels; KWs)
S = get_data(method, channels; KWs)
```

Retrieve time-series data from a web archive to SeisData structure **S**.

method
"IRIS": *IRISWS*.
"FDSN": *FDSNWS dataselect*. Change FDSN servers with keyword `src` using the *server list* (also available by typing `?seis_www`).

channels
Channels to retrieve; can be passed as a *string, string array, or parameter file*. Type `?chanspec` at the Julia prompt for more info.

KWs
Keyword arguments; see also *SeisIO standard KWs* or type `?SeisIO.KW`.
Standard keywords: fmt, nd, opts, rad, reg, si, to, v, w, y
Other keywords:
`s`: Start time
`t`: Termination (end) time

3.1.1 Examples

1. `get_data!(S, "FDSN", "UW.SEP..EHZ,UW.SHW..EHZ,UW.HSR..EHZ", "IRIS", t=(-600))`: using FDSNWS, get the last 10 minutes of data from three short-period vertical-component channels at Mt. St. Helens, USA.

2. `get_data!(S, "IRIS", "CC.PALM..EHN", "IRIS", t=(-120), f="sacbl")`: using IRISWS, fetch the last two minutes of data from component EHN, station PALM (Palmer Lift (Mt. Hood), OR, USA,), network CC (USGS Cascade Volcano Observatory, Vancouver, WA, USA), in bigendian SAC format, and merge into SeisData structure *S*.
3. `get_data!(S, "FDSN", "CC.TIMB..EHZ", "IRIS", t=(-600), w=true)`: using FDSNWS, get the last 10 minutes of data from channel EHZ, station TIMB (Timberline Lodge, OR, USA), save the data directly to disk, and add it to SeisData structure *S*.
4. `S = get_data("FDSN", "HV.MOKD..HHZ", "IRIS", s="2012-01-01T00:00:00", t=(-3600))`: using FDSNWS, fill a new SeisData structure *S* with an hour of data ending at 2012-01-01, 00:00:00 UTC, from HV.MOKD..HHZ (USGS Hawai'i Volcano Observatory).

3.1.2 FDSN Queries

The International Federation of Digital Seismograph Networks (FDSN) is a global organization that supports seismology research. The FDSN web protocol offers near-real-time access to data from thousands of instruments across the world.

FDSN queries in SeisIO are highly customizable; see *data keywords list* and *channel id syntax*.

Data Query

`get_data!(S, "FDSN", channels; KWs)`

`S = get_data("FDSN", channels; KWs)`

FDSN data query with get_data! wrapper.

Shared keywords: fmt, nd, opts, rad, reg, s, si, t, to, v, w, y
Other keywords:
`s`: Start time
`t`: Termination (end) time
`xf`: Name of XML file to save station metadata

Station Query

`FDSNsta!(S, chans, KW)`

`S = FDSNsta(chans, KW)`

Fill channels *chans* of SeisData structure *S* with information retrieved from remote station XML files by web query.

Shared keywords: src, to, v
Other keywords:

`s`: Start time
`t`: Termination (end) time

3.1.3 IRIS Queries

Incorporated Research Institutions for Seismology (IRIS) is a consortium of universities dedicated to the operation of science facilities for the acquisition, management, and distribution of seismological data.

Data Query Features

- Stage zero gains are removed from trace data; all IRIS data will appear to have a gain of 1.0.
- IRISWS disallows wildcards in channel IDs.
- Channel spec *must* include the net, sta, cha fields; thus, CHA = "CC.VALT..BHZ" is OK; CHA = "CC.VALT" is not.

3.2 SeedLink

SeedLink is a TCP/IP-based data transmission protocol that allows near-real-time access to data from thousands of geophysical monitoring instruments. See *data keywords list* and *channel id syntax* for options.

```
SeedLink!(S, chans, KWs)
SeedLink!(S, chans, patts, KWs)
S = SeedLink(chans, KWs)
```

chans
Channel specification can use any of the following options:

1. A comma-separated String where each pattern follows the syntax NET.STA.LOC.CHA.DFLAG, e.g. UW.TDH..EHZ.D. Use "?" to match any single character.
2. An Array{String,1} with one pattern per entry, following the above syntax.
3. The name of a configuration text file, with one channel pattern per line; see *Channel Configuration File syntax*.

patts
Data selection patterns. See official SeedLink documentation; syntax is identical.

KWs
Keyword arguments; see also *SeisIO standard KWs* or type `?SeisIO.KW`.
Standard keywords: fmt, opts, q, si, to, v, w, y
SL keywords: gap, kai, mode, port, refresh, safety, x_on_err

Other keywords:
u specifies the URL without "http://"

Initiate a SeedLink session in DATA mode to feed data from channels chans with selection patterns patts to SeisData structure S. A handle to a TCP connection is appended to S.c.Data are periodically parsed until the connection is closed. One SeisData object can support multiple connections, provided that each connection's streams feed unique channels.

3.2.1 Special Rules

1. **SeedLink follows unusual rules for wild cards in sta and patts:**
    a. * is not a valid SeedLink wild card.
    b. The LOC and CHA fields can be left blank in sta to select all locations and channels.
2. **DO NOT feed one data channel with multiple SeedLink streams. This can have severe consequences:**
    a. A channel fed by multiple live streams will have many small time sequences out of order. merge! is not guaranteed to fix it.
    b. SeedLink will almost certainly crash.
    c. Your data may be corrupted.
    d. The Julia interpreter can freeze, requiring kill -9 on the process.
    e. This is not an "issue". There will never be a workaround. It's what happens when one intentionally causes TCP congestion on one's own machine while writing to open data streams in memory. Hint: don't do this.

3.2.2 Special Methods

- close(S.c[i]) ends SeedLink connection i.
- !deleteat(S.c, i) removes a handle to closed SeedLink connection i.

SeedLink Utilities

SL_info(*v*, *url*)

Retrieve SeedLink information at verbosity level **v** from **url**. Returns XML as a string. Valid strings for **L** are ID, CAPABILITIES, STATIONS, STREAMS, GAPS, CONNECTIONS, ALL.

has_sta(*sta*[, *u*=*url*, *port*=*n*])

SL keywords: gap, port
Other keywords: u specifies the URL without "http://"

Check that streams exist at *url* for stations *sta*, formatted NET.STA. Use "?" to match any single character. Returns true for stations that exist. *sta* can also be the name of a valid config file or a 1d string array.

Returns a BitArray with one value per entry in *sta.*

has_stream(*cha::Union{String*, *Array{String*, *1}}*, *u::String*)

SL keywords: gap, port
Other keywords: u specifies the URL without "http://"

Check that streams with recent data exist at url *u* for channel spec *cha*, formatted NET.STA.LOC.CHA.DFLAG, e.g. "UW.TDH..EHZ.D, CC.HOOD..BH?.E". Use "?" to match any single character. Returns *true* for streams with recent data.

cha can also be the name of a valid config file.

has_stream(*sta::Array{String*, *1}*, *sel::Array{String*, *1}*, *u::String*, *port=N::Int*, *gap=G::Real*)

SL keywords: gap, port
Other keywords: u specifies the URL without "http://"

If two arrays are passed to has_stream, the first should be formatted as SeedLink STATION patterns (formated "SSSSS NN", e.g. ["TDH UW", "VALT CC"]); the second be an array of SeedLink selector patterns (formatted LLCCC.D, e.g. ["??EHZ.D", "??BH?.?"]).

CHAPTER

FOUR

PROCESSING

4.1 Data Processing Functions

Supported processing operations are described below. Functions are organized categorically.

In most cases, a "safe" version of each function can be invoked to create a new SeisData object with the processed output.

Any function that can logically operate on a SeisChannel object will do so. Any function that operates on a SeisData object will also operate on a SeisEvent object by applying itself to the SeisData object in the `:data` field.

4.1.1 Signal Processing

Remove the mean from all channels i with S.fs[i] > 0.0. Specify irr=true to also remove the mean from irregularly sampled channels. Ignores NaNs.

Remove the polynomial trend of degree n from every regularly-sampled channel i in S using a least-squares polynomial fit. Ignores NaNs.

`filtfilt!(S::SeisData[; KWs])`

Apply a zero-phase filter to data in **S.x**.

`filtfilt!(Ev::SeisEvent[; KWs])`

Apply zero-phase filter to **Ev.data.x**.

`filtfilt!(C::SeisChannel[; KWs])`

Apply zero-phase filter to **C.x**

Filtering is applied to each contiguous data segment of each channel separately.

Keywords Keywords control filtering behavior; specify e.g. filtfilt!(S, fl=0.1, np=2, rt="Lowpass"). Default values can be changed by adjustin the *shared keywords*, e.g., SeisIO.KW.Filt.np = 2 changes the default number of poles to 2.

KW	Default	Type	Description
fl	1.0	Float64	lower corner frequency [Hz] [(a)]
fh	15.0	Float64	upper corner frequency [Hz] [(a)]
np	4	Int64	number of poles
rp	10	Int64	pass-band ripple (dB)
rs	30	Int64	stop-band ripple (dB)
rt	"Bandpass"	String	response type (type of filter)
dm	"Butterworth"	String	design mode (name of filter)

[(a)] By convention, the lower corner frequency (fl) is used in a Highpass filter, and fh is used in a Lowpass filter.

`taper!(S)`

Cosine taper each channel in S around time gaps.

`unscale!(S[, irr=false])`

Divide the gains from all channels **i** with **S.fs[i] > 0.0**. Specify **irr=true** to also remove gains of irregularly-sampled channels.

4.1.2 Merge

`merge!(S::SeisData, U::SeisData)`

Merge two SeisData structures. For timeseries data, a single-pass merge-and-prune operation is applied to value pairs whose sample times are separated by less than half the sampling interval.

`merge!(S::SeisData)`

"Flatten" a SeisData structure by merging data from identical channels.

Merge Behavior

Which channels merge?

- Channels merge if they have identical values for `:id`, `:fs`, `:loc`, `:resp`, and `:units`.
- An unset `:loc`, `:resp`, or `:units` field matches any set value in the corresponding field of another channel.

What happens to merged fields?

- The essential properties above are preserved.
- Other fields are combined.
- Merged channels with different *:name* values use the name of the channel with the latest data before the merge; other names are logged to *:notes*.

What does `merge!` resolve?

Issue	Resolution
Empty channels	Delete
Duplicated channels	Delete duplicate channels
Duplicated windows in channel(s)	Delete duplicate windows
Multiple channels, same properties[(a)]	Merge to a single channel
Channel with out-of-order time windows	Sort in chronological order
Overlapping windows, identical data, time-aligned	Windows merged
Overlapping windows, identical data, small time offset[(a)]	Time offset corrected, windows merged
Overlapping windows, non-identical data	Samples averaged, windows merged

[(a)] "Properties" here are `:id`, `:fs`, `:loc`, `:resp`, and `:units`. [(b)] Data offset >4 sample intervals are treated as overlapping and non-identical.

When SeisIO Won't Merge

SeisIO does **not** combine data channels if **any** of the five fields above are non-empty and different. For example, if a SeisData object S contains two channels, each with id "XX.FOO..BHZ", but one has fs=100 Hz and the other fs=50 Hz, **merge!** does nothing.

It's best to merge only unprocessed data. Data segments that were processed independently (e.g. detrended) will be averaged pointwise when merged, which can easily leave data in an unusuable state.

4.1.3 Synchronize

`sync!(S::SeisData)`

Synchronize the start times of all data in S to begin at or after the last start time in S.

`sync!(S::SeisData[, s=ST, t=EN, v=VV])`

Synchronize all data in S to start at *ST* and terminate at *EN* with verbosity level VV.

For regularly-sampled channels, gaps between the specified and true times are filled with the mean; this isn't possible with irregularly-sampled data.

Specifying start time (s)

- s="last": (Default) sync to the last start time of any channel in *S*.
- s="first": sync to the first start time of any channel in *S*.
- A numeric value is treated as an epoch time (*?time* for details).
- A DateTime is treated as a DateTime. (see Dates.DateTime for details.)

- Any string other than "last" or "first" is parsed as a DateTime.

Specifying end time (t)

- t="none": (Default) end times are not synchronized.
- t="last": synchronize all channels to end at the last end time in *S*.
- t="first" synchronize to the first end time in *S*.
- numeric, datetime, and non-reserved strings are treated as for *-s*.

`mseis!(S::SeisData, U::SeisData, ...)`

Merge multiple SeisData structures into S.

4.1.4 Seismic Instrument Responses

`translate_resp!(S, resp_new[, C=chans, wl=g])`

`translate_resp!(Ch, resp_new[, wl=g])`

Translate the instrument response of seismic data channels to **resp_new**. Replaces field **:resp** with **resp_new** for all affected channels.

Channels to translate can be specified with keyword **C=chans**; however, only seismic data channels (with units "m", "m/s", or "m/s2") will be translated.

SeisChannel objects whose units are not "m", "m/s", or "m/s2" are returned with no response translation done.

`remove_resp!(S, C=cha, wl=g])`

`remove_resp!(Ch, wl=g])`

Remove (flatten to DC) the instrument response of seismic data channels **cha** in **S** or **Ch**. Replaces **:resp** with the appropriate (all-pass) response.

Response Keywords

- **C=cha** restricts response translation for SeisData object **S** to channel(s) **cha**. Accepts an Integer, UnitRange, or Array{Int64,1} argument; does *not* accept string IDs. By default, all seismic data channels in **S** have their responses translated to **resp_new**.
- **wl=g** sets the waterlevel to g (default: g = eps(Float32) ~ 1.1f-7). The waterlevel is the minimum magnitude (absolute value) of the normalized old frequency response; in other words, if the old frequency response has a maximum magnitude of 1.0, then no response coefficient can be lower than g. This is useful to prevent "divide by zero" errors, but setting it too high will cause errors.

Precision and Memory Optimization

To optimize speed and memory use, instrument response translation maps data to Complex{Float32} before translation; thus, with Float64 data, there can be minor rounding errors.

Instrument responses are also memory-intensive. The minimum memory consumption to translate the response of a gapless Float32 SeisChannel object is ~7x the size of the object itself.

More precisely, for an object **S** (of Type <: GphysData or GphysChannel), translation requires memory ~ 2 kB + the greater of (7x the size of the longest Float32 segment, or 3.5x the size of the longest Float64 segment). Translation uses four vectors – three complex and one real – that are updated and dynamically resized as the algorithm loops over each segment:

- Old response container: Array{Complex{Float32,1}}(undef, Nx)
- New response container: Array{Complex{Float32,1}}(undef, Nx)
- Complex data container: Array{Complex{Float32,1}}(undef, Nx)
- Real frequencies for FFT: Array{Float32,1}(undef, Nx)

. . . where **Nx** is the number of samples in the longest segment in **S**.

4.1.5 Other Processing Functions

`nanfill!(S)`

For each channel **i** in **S**, replace all NaNs in **S.x[i]** with the mean of non-NaN values.

`ungap!(S[, m=true])`

For each channel **i** in **S**, fill time gaps in **S.t[i]** with the mean of non-NAN data in **S.x[i]**. If **m=false**, gaps are filled with NANs.

CHAPTER

FIVE

SUBMODULES

5.1 Quake

The Quake submodule (accessed with "using SeisIO.Quake") was introduced in SeisIO v0.3.0 to isolate handling of discrete earthquake events from handling of continuous geophysical data. While the channel data are similar, fully describing an earthquake event requires many additional Types (objects) and more information (fields) in channel descriptors.

5.1.1 Types

See Type help text for field descriptions and SeisIO behavior.

EQMag()

Earthquake magnitude object.

EQLoc()

Computed earthquake location.

EventChannel()

A single channel of trace data (digital seismograms) associated with a discrete event (earthquake).

EventTraceData()

A custom structure designed to describe trace data (digital seismograms) associated with a discrete event (earthquake).

PhaseCat()

A seismic phase catalog is a dictionary with phase names for keys (e.g. "pP", "PKP") and SeisPha objects for values.

SeisEvent()

A compound Type comprising a SeisHdr (event header), SeisSrc (source process), and EventTraceData (digital seismograms.)

SeisHdr()

Earthquake event header object.

SeisPha()

A description of a seismic phase measured on a data channel.

SeisSrc()

Seismic source process description.

SourceTime()

QuakeML-compliant seismic source-time parameterization.

5.1.2 Web Queries

Event Header Query

FDSNevq(*ot*)

Shared keywords: evw, rad, reg, mag, nev, src, to, v, w

Multi-server query for the event(s) with origin time(s) closest to *ot*. Returns a tuple consisting of an Array{SeisHdr,1} and an Array{SeisSrc,1}, so that the *i'th entry of each array describes the header and source process of event 'i*.

Notes:

1. Specify *ot* as a string formatted YYYY-MM-DDThh:mm:ss in UTC (e.g. "2001-02-08T18:54:32"). Returns a SeisHdr array.
2. Incomplete string queries are read to the nearest fully-specified time constraint; thus, *FDSNevq("2001-02-08")* returns the nearest event to 2001-02-08T00:00:00.
3. If no event is found in the specified search window, FDSNevq exits with an error.

Shared keywords: evw, reg, mag, nev, src, to, w

Event Header and Data Query

FDSNevt(*ot::String*, *chans::String*)

Get trace data for the event closest to origin time *ot* on channels *chans*. Returns a SeisEvent.

Shared keywords: fmt, mag, nd, opts, pha, rad, reg, src, to, v, w
Other keywords:
`len`: desired record length *in minutes*.

Phase Onset Query

get_pha (*:math:'Delta'::Float64*, *z::Float64*)

Command-line interface to IRIS online implementation of the TauP travel time calculator [1-2]. Returns a matrix of strings. Specify Δ in decimal degrees and z in km with + = down.

Shared keywords keywords: pha, to, v
Other keywords:
`-model`: velocity model (defaults to "iasp91")

References

- Crotwell, H. P., Owens, T. J., & Ritsema, J. (1999). The TauP Toolkit: Flexible seismic travel-time and ray-path utilities, SRL 70(2), 154-160.
- TauP manual: http://www.seis.sc.edu/downloads/TauP/taup.pdf

5.1.3 QuakeML Reader

read_qml (*fpat::String*)

Read QuakeML files matching string pattern *fpat*. Returns a tuple containing an array of *SeisHdr* objects and an array of *SeisSrc* objects, such that the *i'th entry of each array is the preferred location (origin) and event source (focal mechanism or moment tensor) of event 'i.*

If multiple focal mechanisms, locations, or magnitudes are present in a single Event element of the XML file(s), the following rules are used to select one of each per event:

FocalMechanism

1. **preferredFocalMechanismID** if present
2. Solution with best-fitting moment tensor
3. First **FocalMechanism** element

Magnitude

1. **preferredMagnitudeID** if present
2. Magnitude whose ID matches **MomentTensor/derivedOriginID**
3. Last moment magnitude (lowercase scale name begins with "mw")
4. First **Magnitude** element

Origin

1. **preferredOriginID** if present
2. **derivedOriginID** from the chosen **MomentTensor** element
3. First **Origin** element

Non-essential QuakeML data are saved to *misc* in each SeisHdr or SeisSrc object as appropriate.

5.1.4 File Readers

uwpf (*pf*[, *v*])

Read UW-format seismic pick file *pf*. Returns a tuple of (SeisHdr, SeisSrc).

uwpf!(W, pf[, v::Int64=KW.v])

Read UW-format seismic pick info from pickfile *f* into SeisEvent object *W*. Overwrites W.source and W.hdr with pickfile information. Keyword *v* controls verbosity.

readuwevt (*fpat*)

Read University of Washington-format event data with file pattern stub fpat into a SeisEvent object. `fpat` can be a datafile name, a pickfile name, or a stub.

5.1.5 Utility Functions

distaz!(Ev::SeisEvent)

Compute distnace, azimuth, and backazimuth by the Haversine formula. Overwrites Ev.data.dist, Ev.data.az, and Ev.data.baz.

gcdist ([*lat_src*, *lon_src*], *rec*)

Compute great circle distance, azimuth, and backazimuth from a single source with coordinates *[s_lat, s_lon]* to receivers *rec* with coordinates *[r_lat r_lon]* in each row.

show_phases (*P::PhaseCat*)

Formatted display of seismic phases in dictionary P.

5.2 RandSeis

This submodule is used to quickly generate SeisIO objects with quasi-random field contents. Access it with "using SeisIO.RandSeis"

The following are true of all random data objects generated by the RandSeis module:

- Channels have SEED-compliant IDs, sampling frequencies, and data types.
- Random junk fills :notes and :misc.
- Sampling frequency (:fs) is chosen from a set of common values.
- Channel data are randomly generated.
- Time gaps are automatically inserted into regularly-sampled data.

randPhaseCat ()

Generate a random seismic phase catalog suitable for testing EventChannel, EventTraceData, and SeisEvent objects.

randSeisChannel([*c=false*, *s=false*])

Generate a SeisChannel of random data. Specify c=true for campaign-style (irregularly-sampled) data (fs = 0.0); specify s=true to guarantee seismic data. s=true overrides c=true.

randSeisData([*c=0.2*, *s=0.6*])

Generate 8 to 24 channels of random seismic data as a SeisData object.

- 100*c% of channels *after the first* will have irregularly-sampled data (fs = 0.0)
- 100*s% of channels *after the first* are guaranteed to have seismic data.

randSeisData(*N*[, *c=0.2*, *s=0.6*])

Generate N channels of random seismic data as a SeisData object.

randSeisEvent([*c=0.2*, *s=0.6*])

Generate a SeisEvent structure filled with random values.

- 100*c% of channels *after the first* will have irregularly-sampled data (fs = 0.0)
- 100*s% of channels *after the first* are guaranteed to have seismic data.

randSeisEvent(*N*[, *c=0.2*, *s=0.6*])

Generate a random SeisEvent object with N channels of data.

randSeisHdr()

Generate a SeisHdr structure filled with random values.

randSeisSrc()

Generate a SeisSrc structure filled with random values.

CHAPTER

SIX

APPENDICES

6.1 Utility Functions

This appendix covers utility functions that belong in no other category.

d2u(*DT::DateTime*)

Aliased to `Dates.datetime2unix`.

Keyword `hc_new` specifies the new critical damping constant. Keyword `C` specifies an array of channel numbers on which to operate; by default, every channel with fs > 0.0 is affected.

fctopz(*fc*)

Convert critical frequency `fc` to a matrix of complex poles and zeros; zeros in `resp[:,1]`, poles in `resp[:,2]`.

find_regex(*path::String*, *r::Regex*)

OS-agnostic equivalent to Linux *find*. First argument is a path string, second is a Regex. File strings are postprocessed using Julia's native PCRE Regex engine. By design, *find_regex* only returns file names.

getbandcode(*fs*, *fc=FC*)

Get SEED-compliant one-character band code corresponding to instrument sample rate `fs` and corner frequency `FC`. If unset, `FC` is assumed to be 1 Hz.

ls(*s::String*)

Similar functionality to Bash ls with OS-agnostic output. Accepts wildcards in paths and file names. * Always returns the full path and file name. * Partial file name wildcards (e.g. "*ls(data/2006*.sac)*) invoke *glob*. * Path wildcards (e.g. *ls(/data/*/*.sac)*) invoke *find_regex* to circumvent glob limitations. * Passing ony "*" as a filename (e.g. "*ls(/home/*)*) invokes *find_regex* to recursively search subdirectories, as in the Bash shell.

ls()

Return full path and file name of files in current working directory.

j2md(*y*, *j*)

Convert Julian day **j** of year **y** to month, day.

md2j(*y*, *m*, *d*)

Convert month **m**, day **d** of year **y** to Julian day **j**.

Remove unwanted characters from S.

parsetimewin (s, t)

Convert times **s** and **t** to strings α, ω sorted $\alpha < \omega$. **s** and **t** can be real numbers, DateTime objects, or ASCII strings. Expected string format is "yyyy-mm-ddTHH:MM:SS.nnn", e.g. 2016-03-23T11:17:00.333.

"Safe" synchronize of start and end times of all trace data in SeisData structure `S` to a new structure `U`.

u2d (x)

Alias to `Dates.unix2datetime`.

function:: w_time(W::Array{Int64,2}, fs::Float64)

Convert matrix W from time windows (w[:,1]:w[:,2]) in integer μs from the Unix epoch (1970-01-01T00:00:00) to sparse delta-encoded time representation. Specify fs in Hz.

6.2 SeisIO Write Formats

Files are written in little-endian byte order. Abbreviations used:

Type	Meaning	C	Fortran 77
Char	Unicode character	wchar	CHARACTER*4
Float32	32-bit float	float	REAL
Float64	64-bit float	double	REAL*8
Int8	signed 8-bit int	short	INTEGER
Int16	signed 16-bit int	int	INTEGER*2
Int32	signed 32-bit int	long	INTEGER*4
Int64	signed 64-bit integer	long long	INTEGER*8
UInt8	unsigned 8-bit int	unsigned short	CHARACTER
UInt16	unsigned 16-bit int	unsigned	
UInt32	unsigned 32-bit int	unsigned long	
UInt64	unsigned 64-bit int	unsigned long long	

Special instructions:

Parentheses, "()", denote a custom object Type.
"{ (condition)" denotes the start of a loop; (condition) is the control flow.
"}" denotes the end of a loop.

Note that String in Julia has no exact C equivalence. SeisIO writes each String in two parts: an Int64 (String length in bytes) followed by the String contents (as bytes, equivalent to UInt8). Unlike C/Fortran, there are no issues with strings that contain the null character (0x00 or `'\x0'`).

6.2.1 SeisIO File

Var	Meaning	T	N
	"SEISIO"	UInt8	6
V	SeisIO file format version	Float32	1
J	# of SeisIO objects in file	UInt32	1
C	*SeisIO object codes* for each object	UInt32	J
B	Byte indices for each object	UInt64	J
{			for i = 1:J
	(Objects)	variable	J
}			
ID	ID hashes	UInt64	variable
TS	Start times	Int64	variable
TE	End times	Int64	variable
P	Parent object index in C and B	variable	
bID	Byte offset of ID array	Int64	1
bTS	Byte offset of TS array	Int64	1
bTE	Byte offset of TE array	Int64	1
bP	Byte offset of P array	Int64	1

ID, TS, and TE are the ID, data start time, and data end time of each channel in each object. P is the index of the parent object in C and B. TS and TE are measured from Unix epoch time (1970-01-01T00:00:00Z) in integer microseconds.

Intent: when seeking data from channel `i` between times `s` and `t`, if `hash(i)` matches `ID[j]` and the time windows overlap, retrieve index `k = P[j]` from NP, seek to byte offset `B[k]`, and read an object of type `C[k]` from file.

If an archive contains no data objects, ID, TS, TE, and P are empty; equivalently, `bID == bTS`.

6.3 Simple Object Types

Fields of these objects are written in one of three ways: as "plain data" types, such as UInt8 or Float64; as arrays; or as strings.

In a simple object, each array is stored as follows: 1. Int64 number of dimensions (e.g. 2) 2. Int64 array of dimensions themselves (e.g. 2, 2) 3. Array values (e.g. 0.08250153, 0.023121119, 0.6299772, 0.79595184)

6.3.1 EQLoc

Var	Type	N	Meaning
lat	Float64	1	latitude
lon	Float64	1	longitude
dep	Float64	1	depth
dx	Float64	1	x-error
dy	Float64	1	y-error
dz	Float64	1	z-error
dt	Float64	1	t-error (error in origin time)
se	Float64	1	standard error
rms	Float64	1	rms pick error
gap	Float64	1	azimuthal gap
dmin	Float64	1	minimum source-receiver distance in location
dmax	Float64	1	maximum source-receiver distance in location
nst	Int64	1	number of stations used to locate earthquake
flags	UInt8	1	one-bit flags for special location properties
Ld	Int64	1	length of "datum" string in bytes
datum	UInt8	Ld	Datum string
Lt	Int64	1	length of "typ" (event type) string in bytes
typ	UInt8	Lt	earthquake type string
Li	Int64	1	length of "sig" (error significance) string in bytes
sig	UInt8	Li	earthquake location error significance string
Lr	Int64	1	length of "src" (data source) string in bytes
src	UInt8	Lr	data source string

flag meanings: (0x01 = true, 0x00 = false)

1. x fixed?
2. y fixed?
3. z fixed?
4. t fixed?

In Julia, get the value of flag[n] with `>>(<<(flags, n-1), 7)`.

6.3.2 EQMag

Var	Type	N	Meaning
val	Float32	1	magnitude value
gap	Float64	1	largest azimuthal gap between stations in magnitude
nst	Int64	1	number of stations used in magnitude computation
Lsc	Int64	1	length of magnitude scale string
msc	UInt8	Lsc	magnitude scale string
Lr	Int64	1	length of data source string
src	UInt8	Lr	data source string

6.3.3 SeisPha

Var	Type	N	Meaning
F	Float64	8	amplitude, distance, incidence angle, residual,
			ray parameter, takeoff angle, travel time, uncertainty
C	Char	2	polarity, quality

6.3.4 SourceTime

Var	Type	N	Meaning
Ld	Int64	1	size of descriptive string in bytes
desc	UInt8	1	descriptive string
F	Float64	3	duration, rise time, decay time

6.3.5 StringVec

A vector of variable-length strings; its exact Type in Julia is Array{String,1}.

Table 1: StringVec

Var	Type	N	Meaning
ee	UInt8	1	is this string vector empty?[1]
L	Int64	1	number of strings to read
{			i = 1:L
nb	Int64	1	length of string in bytes
str	UInt8	nb	string
}			

[1] If `ee == 0x00`, then no values are stored for L, nb, or str.

6.4 Location Types

6.4.1 GenLoc

Var	Type	N	Meaning
Ld	Int64	1	length of datum string in bytes
datum	UInt8	Ld	datum string
Ll	Int64	1	length of location vector in bytes
loc	Float64	Ll	location vector

6.4.2 GeoLoc

Var	Type	N	Meaning
Ld	Int64	1	length of datum string in bytes
datum	UInt8	Ld	datum string
F	Float64	6	latitude, longitude, elevation,
			depth, azimuth, incidence

6.4.3 UTMLoc

Var	Type	N	Meaning
Ld	Int64	1	length of datum string in bytes
datum	UInt8	N	datum string
zone	Int8	1	UTM zone number
hemi	Char	1	hemisphere
E	UInt64	1	Easting
N	UInt64	1	Northing
F	Float64	4	elevation, depth, azimuth, incidence

6.4.4 XYLoc

Var	Type	N	Meaning
Ld	Int64	1	Length of datum string in bytes
datum	UInt8	Ld	datum string
F	Float64	8	x, y, z, azimuth, incidence, origin x, origin y, origin z

6.5 Response Types

6.5.1 GenResp

Var	Type	N	Meaning
Ld	Int64	1	length of descriptive string in bytes
desc	UInt8	Ld	descriptive string
nr	Int64	1	Number of rows in complex response matrix
nc	Int64	1	Number of columns in complex response matrix
resp	Complex{Float64,2}	nr*nc	complex response matrix

6.5.2 PZResp

Var	Type	N	Meaning
c	Float32	1	damping constant
np	Int64	1	number of complex poles
p	Complex{Float32,1}	np	complex poles vector
nz	Int64	1	number of complex zeros
z	Complex{Float32,1}	nz	complex zeros vector

PZResp64 is identical to PZResp with Float64 values for c, p, z, rather than Float32.

6.6 The Misc Dictionary

Most compound objects below contain a dictionary (Dict{String,Any}) for non-essential information in a field named `:misc`. The tables below describe how this field is written to disk.

6.6.1 Misc

Var	Type	N	Meaning
N	Int64	1	number of items in dictionary[2]
K	(StringVec)	1	dictionary keys
{			for i = 1:N
c	UInt8	1	*Type code* of object i
o	variable	1	object i
}			

[2] If `N == 0`, then N is the only value present.

Dictionary Contents

These subtables describe how to read the possible data types in a Misc dictionary.

6.6.2 String Array (c == 0x81)

Var	Type	N	Meaning
A	(StringVec)	1	string vector

6.6.3 Other Array (c == 0x80 or c > 0x81)

Var	Type	N	Meaning
nd	Int64	1	number of dimensions in array
dims	Int64	nd	array dimensions
arr	varies	prod(nd)	array

6.6.4 String (c == 0x01)

Var	Type	N	Meaning
L	Int64	1	size of string in bytes
str	UInt8	1	string

6.6.5 Bits Type (c == 0x00 or 0x01 < c < 0x7f)

Read a single value whose Type corresponds to the UInt8 *Type code*.

6.7 Compound Object Types

Each of these objects contains at least one of the above simple object types.

6.7.1 PhaseCat

Var	Type	N	Meaning
N	Int64	1	number of SeisPha objects to read[3]
K	(StringVec)	1	dictionary keys
pha	(SeisPha)	N	seismic phases

[3] If `N == 0`, then N is the only value present.

6.7.2 EventChannel

A single channel of data related to a seismic event

Var	Type	N	Meaning
Ni	Int64	1	size of id string in bytes
id	UInt8	Ni	id string
Nn	Int64	1	size of name string in bytes
name	UInt8	Nn	name string
Lt	UInt8	1	*location Type code*
loc	(Loc Type)	1	instrument position
fs	Float64	1	sampling frequency in Hz
gain	Float64	1	scalar gain
Rt	UInt8	1	*response Type code*
resp	(Resp Type)	1	instrument response
Nu	Int64	1	size of units string in bytes
units	UInt8	Nu	units string
az	Float64	1	azimuth
baz	Float64	1	backazimuth
dist	Float64	1	source-receiver distance
pha	(PhaseCat)	1	phase catalog
Nr	Int64	1	size of data source string in bytes
src	UInt8	Nr	data source string
misc	(Misc)	1	dictionary for non-essential information
notes	(StringVec)	1	notes and automated logging
Nt	Int64	1	length of time gaps matrix
T	Int64	2Nt	time gaps matrix
Xc	UInt8	1	*Type code* of data vector
Nx	Int64	1	number of samples in data vector
X	variable	NX	data vector

6.7.3 SeisChannel

A single channel of univariate geophysical data

Var	Type	N	Meaning
Ni	Int64	1	size of id string in bytes
id	UInt8	Ni	id string
Nn	Int64	1	size of name string in bytes
name	UInt8	Nn	name string
Lt	UInt8	1	*location Type code*
loc	(Loc Type)	1	instrument position
fs	Float64	1	sampling frequency in Hz
gain	Float64	1	scalar gain
Rt	UInt8	1	*response Type code*
resp	(Resp Type)	1	instrument response
Nu	Int64	1	size of units string in bytes
units	UInt8	Nu	units string
Nr	Int64	1	size of data source string in bytes
src	UInt8	Nr	data source string
misc	(Misc)	1	dictionary for non-essential information
notes	(StringVec)	1	notes and automated logging
Nt	Int64	1	length of time gaps matrix
T	Int64	2Nt	time gaps matrix
Xc	UInt8	1	*Type code* of data vector
Nx	Int64	1	number of samples in data vector
X	variable	NX	data vector

6.7.4 EventTraceData

A multichannel record of time-series data related to a seismic event.

Var	Type	N	Meaning
N	Int64	1	number of data channels
Lc	UInt8	N	*location Type codes* for each data channel
Rc	UInt8	N	*response Type codes* for each data channel
Xc	UInt8	N	data *Type codes* for each data channel
cmp	UInt8	1	are data compressed? (0x01 = yes)
Nt	Int64	N	number of rows in time gaps matrix for each channel
Nx	Int64	N	length of data vector for each channel[4]
id	(StringVec)	1	channel ids
name	(StringVec)	1	channel names
loc	(Loc Type)	N	instrument positions
fs	Float64	N	sampling frequencies of each channel in Hz
gain	Float64	N	scalar gains of each channel
resp	(Resp Type)	N	instrument responses
units	(StringVec)	1	units of each channel's data
az	Float64	N	event azimuth
baz	Float64	N	backazimuths to event
dist	Float64	N	source-receiver distances
pha	(PhaseCat)	N	phase catalogs for each channel
src	(StringVec)	1	data source strings for each channel
misc	(Misc)	N	dictionaries of non-essential information for each channel
notes	(StringVec)	N	notes and automated logging for each channel
{			for i = 1:N
T	Int64	2Nt[i]	Matrix of time gaps for channel i
}			
{			for i = 1:N
X	Xc[i]	Nx[i]	Data vector i[5]
}			

6.7.5 SeisData

A record containing multiple channels of univariate geophysical data.

[4] If cmp == 0x01, each value in Nx is the number of bytes of compressed data to read; otherwise, this is the number of samples in each channel.

[5] If cmp == 0x01, read Nx[i] samples of type UInt8 and pass through lz4 decompression to generate data vector i; else read Nx[i] samples of the type corresponding to code Xc[i].

Var	Type	N	Meaning
N	Int64	1	number of data channels
Lc	UInt8	N	*location Type codes* for each data channel
Rc	UInt8	N	*response Type codes* for each data channel
Xc	UInt8	N	data *Type codes* for each data channel
cmp	UInt8	1	are data compressed? (0x01 = yes)
Nt	Int64	N	number of rows in time gaps matrix for each channel
Nx	Int64	N	length of data vector for each channel[6]
id	(StringVec)	1	channel ids
name	(StringVec)	1	channel names
loc	(Loc Type)	N	instrument positions
fs	Float64	N	sampling frequencies of each channel in Hz
gain	Float64	N	scalar gains of each channel
resp	(Resp Type)	N	instrument responses
units	(StringVec)	1	units of each channel's data
src	(StringVec)	1	data source strings for each channel
misc	(Misc)	N	dictionaries of non-essential information for each channel
notes	(StringVec)	N	notes and automated logging for each channel
{			for i = 1:N
T	Int64	2Nt[i]	Matrix of time gaps for channel i
}			
{			for i = 1:N
X	Xc[i]	Nx[i]	Data vector i[7]
}			

[6] If cmp == 0x01, each value in Nx is the number of bytes of compressed data to read; otherwise, this is the number of samples in each channel.

[7] If cmp == 0x01, read Nx[i] samples of type UInt8 and pass through lz4 decompression to generate data vector i; else read Nx[i] samples of the type corresponding to code Xc[i].

6.7.6 SeisHdr

Var	Type	N	Meaning
Li	Int64	1	length of event ID string
id	UInt8	Li	event ID string
iv	UInt8	1	intensity value
Ls	Int64	1	length of intensity scale string
isc	UInt8	Ls	intensity scale string
loc	(EQLoc)	1	earthquake location
mag	(EQMag)	1	earthquake magnitude
misc	(Misc)	1	dictionary containing non-essential information
notes	(StringVec)	1	notes and automated logging
ot	Int64	1	origin time[8]
Lr	Int64	1	length of data source string
src	UInt8	Lr	data source string
Lt	Int64	1	length of event type string
typ	UInt8	Lt	event type string

6.7.7 SeisSrc

Var	Type	N	Meaning
Li	Int64	1	length of source id string
id	UInt8	Li	id string
Le	Int64	1	length of event id string
eid	UInt8	Le	event id string
m0	Float64	1	scalar moment
Lm	Int64	1	length of moment tensor vector
mt	Float64	Lm	moment tensor vector
Ld	Int64	1	length of moment tensor misfit vector
dm	Float64	Ld	moment tensor misfit vector
np	Int64	1	number of polarities
gap	Float64	1	max. azimuthal gap
pad	Int64	2	dimensions of principal axes matrix
pax	Float64	pad[1]*pad[2]	principal axes matrix
pld	Int64	2	dimensions of nodal planes matrix
planes	Float64	pld[1]*pld[2]	nodal planes matrix
Lr	Int64	1	length of data source string
src	UInt8	1	data source string
st	(SourceTime)	1	source-time description
misc	(Misc)	1	Dictionary containing non-essential information
notes	(StringVec)	1	Notes and automated logging

[8] Measured from Unix epoch time (1970-01-01T00:00:00Z) in integer microseconds

6.7.8 SeisEvent

Var	Type	N	Meaning
hdr	(SeisHdr)	1	event header
source	(SeisSrc)	1	event source process
data	(EventTraceData)	1	event trace data

6.8 Data Type Codes

Each Type code is written to disk as a UInt8, with the important exception of SeisIO custom object Type codes (which use UInt32).

6.8.1 Loc Type Codes

UInt8	Type
0x00	GenLoc
0x01	GeoLoc
0x02	UTMLoc
0x03	XYLoc

6.8.2 Resp Type Codes

UInt8	Type
0x00	GenResp
0x01	PZResp
0x02	PZResp64

6.8.3 Other Type Codes

Only the Types below are faithfully preserved in write/read of a :misc field dictionary; other Types are not written to file and can cause `wseis` to throw errors.

Type	UInt8	Type	UInt8
Char	0x00	Array{Char,N}	0x80
String	0x01	Array{String,N}	0x81
UInt8	0x10	Array{UInt8,N}	0x90
UInt16	0x11	Array{UInt16,N}	0x91
UInt32	0x12	Array{UInt32,N}	0x92
UInt64	0x13	Array{UInt64,N}	0x93
UInt128	0x14	Array{UInt128,N}	0x94
Int8	0x20	Array{Int8,N}	0xa0
Int16	0x21	Array{Int16,N}	0xa1
Int32	0x22	Array{Int32,N}	0xa2
Int64	0x23	Array{Int64,N}	0xa3
Int128	0x24	Array{Int128,N}	0xa4
Float16	0x30	Array{Float16,N}	0xb0
Float32	0x31	Array{Float32,N}	0xb1
Float64	0x32	Array{Float64,N}	0xb2
Complex{UInt8}	0x50	Array{Complex{UInt8},N}	0xd0
Complex{UInt16}	0x51	Array{Complex{UInt16},N}	0xd1
Complex{UInt32}	0x52	Array{Complex{UInt32},N}	0xd2
Complex{UInt64}	0x53	Array{Complex{UInt64},N}	0xd3
Complex{UInt128}	0x54	Array{Complex{UInt128},N}	0xd4
Complex{Int8}	0x60	Array{Complex{Int8},N}	0xe0
Complex{Int16}	0x61	Array{Complex{Int16},N}	0xe1
Complex{Int32}	0x62	Array{Complex{Int32},N}	0xe2
Complex{Int64}	0x63	Array{Complex{Int64},N}	0xe3
Complex{Int128}	0x64	Array{Complex{Int128},N}	0xe4
Complex{Float16}	0x70	Array{Complex{Float16},N}	0xf0
Complex{Float32}	0x71	Array{Complex{Float32},N}	0xf1
Complex{Float64}	0x72	Array{Complex{Float64},N}	0xf2

SeisIO Object Type codes

UInt32 Code	Object Type
0x20474330	EventChannel
0x20474331	SeisChannel
0x20474430	EventTraceData
0x20474431	SeisData
0x20495030	GenLoc
0x20495031	GeoLoc
0x20495032	UTMLoc
0x20495033	XYLoc
0x20495230	GenResp
0x20495231	PZResp64
0x20495232	PZResp
0x20504330	PhaseCat
0x20534530	SeisEvent
0x20534830	SeisHdr
0x20535030	SeisPha
0x20535330	SeisSrc
0x20535430	SourceTime
0x45514c30	EQLoc
0x45514d30	EQMag

6.9 Data Requests Syntax

6.9.1 Channel ID Syntax

`NN.SSSSS.LL.CC` (net.sta.loc.cha, separated by periods) is the expected syntax for all web functions. The maximum field width in characters corresponds to the length of each field (e.g. 2 for network). Fields can't contain whitespace.

`NN.SSSSS.LL.CC.T` (net.sta.loc.cha.tflag) is allowed in SeedLink. `T` is a single-character data type flag and must be one of `DECOTL`: Data, Event, Calibration, blOckette, Timing, or Logs. Calibration, timing, and logs are not in the scope of SeisIO and may crash SeedLink sessions.

The table below specifies valid types and expected syntax for channel lists.

Type	Description	Example
String	Comma-delineated list of IDs	"PB.B004.01.BS1, PB.B002.01.BS1"
Array{String,1}	String array, one ID string per entry	["PB.B004.01.BS1", "PB.B002.01.BS1"]
Array{String,2}	String array, one set of IDs per row	["PB" "B004" "01" "BS1"; "PB" "B002" "01" "BS1"]

The expected component order is always network, station, location, channel; thus, "UW.TDH..EHZ" is OK,

but "UW.TDH.EHZ" fails.

chanspec()

Type ?chanspec in Julia to print the above info. to stdout.

Wildcards and Blanks

Allowed wildcards are client-specific.

- The LOC field can be left blank in any client: "UW.ELK..EHZ" and ["UW" "ELK" "" "EHZ"] are all valid. Blank LOC fields are set to -- in IRIS, * in FDSN, and ?? in SeedLink.
- ? acts as a single-character wildcard in FDSN & SeedLink. Thus, CC.VALT..??? is valid.
- * acts as a multi-character wildcard in FDSN. Thus, CC.VALT..* and CC.VALT..??? behave identically in FDSN.
- Partial specifiers are OK, but a network and station are always required: "UW.EL?" is OK, ".ELK.." fails.

Channel Configuration Files

One entry per line, ASCII text, format NN.SSSSS.LL.CCC.D. Due to client-specific wildcard rules, the most versatile configuration files are those that specify each channel most completely:

```
# This only works with SeedLink
GE.ISP..BH?.D
NL.HGN
MN.AQU..BH?
MN.AQU..HH?
UW.KMO
CC.VALT..BH?.D

# This works with FDSN and SeedLink, but not IRIS
GE.ISP..BH?
NL.HGN
MN.AQU..BH?
MN.AQU..HH?
UW.KMO
CC.VALT..BH?

# This works with all three:
GE.ISP..BHZ
GE.ISP..BHN
GE.ISP..BHE
MN.AQU..BHZ
MN.AQU..BHN
MN.AQU..BHE
MN.AQU..HHZ
MN.AQU..HHN
MN.AQU..HHE
```

(continues on next page)

(continued from previous page)

```
UW.KMO..EHZ
CC.VALT..BHZ
CC.VALT..BHN
CC.VALT..BHE
```

Server List

String	Source
BGR	http://eida.bgr.de
EMSC	http://www.seismicportal.eu
ETH	http://eida.ethz.ch
GEONET	http://service.geonet.org.nz
GFZ	http://geofon.gfz-potsdam.de
ICGC	http://ws.icgc.cat
INGV	http://webservices.ingv.it
IPGP	http://eida.ipgp.fr
IRIS	http://service.iris.edu
ISC	http://isc-mirror.iris.washington.edu
KOERI	http://eida.koeri.boun.edu.tr
LMU	http://erde.geophysik.uni-muenchen.de
NCEDC	http://service.ncedc.org
NIEP	http://eida-sc3.infp.ro
NOA	http://eida.gein.noa.gr
ORFEUS	http://www.orfeus-eu.org
RESIF	http://ws.resif.fr
SCEDC	http://service.scedc.caltech.edu
TEXNET	http://rtserve.beg.utexas.edu
USGS	http://earthquake.usgs.gov
USP	http://sismo.iag.usp.br

seis_www()

Type `?seis_www` in Julia to print the above info. to stdout.

6.9.2 Time Syntax

Specify time inputs for web queries as a DateTime, Real, or String. The latter must take the form YYYY-MM-DDThh:mm:ss.nnn, where `T` is the uppercase character *T* and `nnn` denotes milliseconds; incomplete time strings treat missing fields as 0.

type(s)	type(t)	behavior
DT	DT	Sort only
R	DT	Add `s` seconds to `t`
DT	R	Add `t` seconds to `s`
S	R	Convert `s` to DateTime, add `t`
R	S	Convert `t` to DateTime, add `s`
R	R	Add `s, t` seconds to `now()`

(above, R = Real, DT = DateTime, S = String, I = Integer)

6.10 SeisIO Standard Keywords

SeisIO.KW is a memory-resident structure of default values for common keywords used by package functions. KW has one substructure, SL, with keywords specific to SeedLink. These defaults can be modified, e.g., SeisIO.KW.nev=2 changes the default for nev to 2.

KW	Default	T[1]	Meaning
evw	[600.0, 600.0]	A{F,1}	time search window [o-evw[1], o+evw[2]]
fmt	"miniseed"	S	request data format
mag	[6.0, 9.9]	A{F,1}	magnitude range for queries
nd	1	I	number of days per subrequest
nev	1	I	number of events returned per query
nx_add	360000	I	length increase of undersized data array
nx_new	8640000	I	number of samples for a new channel
opts	""	S	user-specified options[2]
pha	"P"	S	seismic phase arrival times to retrieve
rad	[]	A{F,1}	radial search region[3]
reg	[]	A{F,1}	rectangular search region[4]
si	true	B	autofill station info on data req?[5]
to	30	I	read timeout for web requests (s)
v	0	I	verbosity
w	false	B	write requests to disc?[6]
y	false	B	sync data after web request?[7]

[1] Types: A = Array, B = Boolean, C = Char, DT = DateTime, F = Float, I = Integer, R = Real, S = String, U8 = Unsigned 8-bit integer

[2] String is passed as-is, e.g. "szsrecs=true&repo=realtime" for FDSN. String should not begin with an ampersand.

[3] Specify region **[center_lat, center_lon, min_radius, max_radius, dep_min, dep_max]**, with lat, lon, and radius in decimal degrees (°) and depth in km with + = down. Depths are only used for earthquake searches.

[4] Specify region **[lat_min, lat_max, lon_min, lon_max, dep_min, dep_max]**, with lat, lon in decimal degrees (°) and depth in km with + = down. Depths are only used for earthquake searches.

[5] Not used with IRISWS.

[6] **-v=0** = quiet; 1 = verbose, 2 = debug; 3 = verbose debug

[7] If **-w=true**, a file name is automatically generated from the request parameters, in addition to parsing data to a SeisData structure. Files are created from the raw download even if data processing fails, in contrast to get_data(... wsac=true).

Table Footnotes

6.10.1 SeedLink Keywords

Change these with SeisIO.KW.SL.[key] = value, e.g., SeisIO.KW.SL.refresh = 30.

kw	def	type	meaning
gap	3600	R	a stream with no data in >gap seconds is considered offline
kai	600	R	keepalive interval (s)
mode	"DATA"	I	"TIME", "DATA", or "FETCH"
port	18000	I	port number
refresh	20	R	base refresh interval (s)[8]
x_on_err	true	Bool	exit on error?

Table Footnotes

6.11 Examples

6.11.1 FDSN data query

1. Download 10 minutes of data from four stations at Mt. St. Helens (WA, USA), delete the low-gain channels, and save as SAC files in the current directory.

```
S = get_data("FDSN", "CC.VALT, UW.SEP, UW.SHW, UW.HSR", src="IRIS", t=-600)
S -= "SHW.ELZ..UW"
S -= "HSR.ELZ..UW"
writesac(S)
```

2. Get 5 stations, 2 networks, all channels, last 600 seconds of data at IRIS

```
CHA = "CC.PALM, UW.HOOD, UW.TIMB, CC.HIYU, UW.TDH"
TS = u2d(time())
TT = -600
S = get_data("FDSN", CHA, src="IRIS", s=TS, t=TT)
```

3. A request to FDSN Potsdam, time-synchronized, with some verbosity

```
ts = "2011-03-11T06:00:00"
te = "2011-03-11T06:05:00"
R = get_data("FDSN", "GE.BKB..BH?", src="GFZ", s=ts, t=te, v=1, y=true)
```

6.11.2 FDSN station query

A sample FDSN station query

[8] This value is modified slightly by each SeedLink session to minimize the risk of congestion

```
S = FDSNsta("CC.VALT..,PB.B001..BS?,PB.B001..E??")
```

6.11.3 FDSN event header/data query

Get seismic and strainmeter records for the P-wave of the Tohoku-Oki great earthquake on two borehole stations and write to native SeisData format:

```
S = FDSNevt("201103110547", "PB.B004..EH?,PB.B004..BS?,PB.B001..BS?,PB.B001..
↪EH?")
wseis("201103110547_evt.seis", S)
```

6.11.4 IRISWS data query

Note that the "src" keyword is not used in IRIS queries.

1. Get trace data from IRISws from `TS` to `TT` at channels `CHA`

```
S = SeisData()
CHA = "UW.TDH..EHZ, UW.VLL..EHZ, CC.VALT..BHZ"
TS = u2d(time()-86400)
TT = 600
get_data!(S, "IRIS", CHA, s=TS, t=TT)
```

2. Get synchronized trace data from IRISws with a 55-second timeout on HTTP requests, written directly to disk.

```
CHA = "UW.TDH..EHZ, UW.VLL..EHZ, CC.VALT..BHZ"
TS = u2d(time())
TT = -600
S = get_data("IRIS", CHA, s=TS, t=TT, y=true, to=55, w=true)
```

3. Request 10 minutes of continuous vertical-component data from a small May 2016 earthquake swarm at Mt. Hood, OR, USA:

```
STA = "UW.HOOD.--.BHZ,CC.TIMB.--.EHZ"
TS = "2016-05-16T14:50:00"; TE = 600
S = get_data("IRIS", STA, "", s=TS, t=TE)
```

4. Grab data from a predetermined time window in two different formats

```
ts = "2016-03-23T23:10:00"
te = "2016-03-23T23:17:00"
S = get_data("IRIS", "CC.JRO..BHZ", s=ts, t=te, fmt="sacbl")
T = get_data("IRIS", "CC.JRO..BHZ", s=ts, t=te, fmt="miniseed")
```

6.11.5 SeedLink sessions

1. An attended SeedLink session in DATA mode. Initiate a SeedLink session in DATA mode using config file SL.conf and write all packets received directly to file (in addition to parsing to S itself). Set nominal refresh interval for checking for new data to 10 s. A mini-seed file will be generated automatically.

```
S = SeisData()
SeedLink!(S, "SL.conf", mode="DATA", r=10, w=true)
```

2. An unattended SeedLink download in TIME mode. Get the next two minutes of data from stations GPW, MBW, SHUK in the UW network. Put the Julia REPL to sleep while the request fills. If the connection is still open, close it (SeedLink's time bounds arent precise in TIME mode, so this may or may not be necessary). Pause briefly so that the last data packets are written. Synchronize results and write data in native SeisIO file format.

```
sta = "UW.GPW,UW.MBW,UW.SHUK"
s0 = now()
S = SeedLink(sta, mode="TIME", s=s0, t=120, r=10)
sleep(180)
isopen(S.c[1]) && close(S.c[1])
sleep(20)
sync!(S)
fname = string("GPW_MBW_SHUK", s0, ".seis")
wseis(fname, S)
```

3. A SeedLink session in TIME mode

```
sta = "UW.GPW, UW.MBW, UW.SHUK"
S1 = SeedLink(sta, mode="TIME", s=0, t=120)
```

4. A SeedLink session in DATA mode with multiple servers, including a config file. Data are parsed roughly every 10 seconds. A total of 5 minutes of data are requested.

```
sta = ["CC.SEP", "UW.HDW"]
# To ensure precise timing, we'll pass d0 and d1 as strings
st = 0.0
en = 300.0
dt = en-st
(d0,d1) = parsetimewin(st,en)

S = SeisData()
SeedLink!(S, sta, mode="TIME", r=10.0, s=d0, t=d1)
println(stdout, "...first link initialized...")

# Seedlink with a config file
config_file = "seedlink.conf"
SeedLink!(S, config_file, r=10.0, mode="TIME", s=d0, t=d1)
println(stdout, "...second link initialized...")

# Seedlink with a config string
SeedLink!(S, "CC.VALT..???, UW.ELK..EHZ", mode="TIME", r=10.0, s=d0, t=d1)
println(stdout, "...third link initialized...")
```

INDEX