SeisIO Documentation

Release 0.1.2 rc

Joshua Jones

Mar 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 geophysical time series data. 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 data types that track record times and other necessary quantities for further processing. New data are easily merged into existing structures with basic commands like `+`. Unwanted channels can be removed just as easily. Data can be saved to a native SeisIO format or written to SAC.

1.1.2 Installation

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

```
add https://github.com/jpjones76/SeisIO.jl; build; precompile
```

Dependencies should be installed automatically. To run tests that verify functionality works correctly, type

```
test SeisIO
```

in the Pkg environment. Allow 10-20 minutes for all tests to complete.

To get started, exit the Pkg environment by pressing Control + C, then type

```
using SeisIO
```

1.1.3 Updating

From the Julia prompt: press `]` to enter the Pkg environment, then type `update`. You may need to restart the Julia REPL afterward to use the updated version.

1.2 Working with Data

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

1.2.1 First Steps

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

`Ch = SeisChannel()`

The meanings of the field names are explained *here<dkw>*. 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<webex>*.

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.

Merge

SeisData structures can be merged using the function **merge!**, but this is much more complicated than addition.

`merge!(S)`

- Does nothing to channels with unique IDs.
- For sets of channels in S that share an ID… + Adjusts all matching channels to the :gain, :fs, :loc, and :resp fields of the channel the latest data + Time-sorts data from all matching channels by *S.t* + Averages data points that occur simultaneously in multiple members of the set
- throws an error if joining data that have the same ID and different units.

1.2.3 Keeping Track

Because tracking arbitrary operations can be difficult, several functions have been written to keep track of data and operations in a semi-automated way.

Taking Notes

Most functions that add or process data note this in the appropriate channel's :notes field. However, you can also make your own 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`.

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**.

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` is \x7f (ASCII/Unicode U+007f).

timestamp ()

Return current UTC time formatted yyyy-mm-ddTHH:MM:SS.$\mu\mu\mu$.

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

SeisIO functions record the *last* source used to populate each channel in the :src field. Typically this is a string.

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 Data Types

- SeisChannel: single-channel univariate data
- SeisData: multi-channel univariate data
- SeisHdr: seismic event header
- SeisEvent: composite type for events with header and trace data

Data types in SeisIO can be manipulated using standard Julia commands.

1.3.1 Initialization

SeisChannel

SeisChannel()

Initialize an empty SeisChannel structure.

SeisChannel(*; [KWs]*)

Set fields at creation by specifying fieldnames as keywords, e.g. **SeisChannel(fs=100.0)** creates a new SeisChannel structure with fs = 100.0 Hz.

SeisData

SeisData()

Initialize an empty SeisData structure. Fields cannot be set at creation.

SeisData(*n*)

Initialize an empty SeisData structure with S.n channel containers.

SeisData(*S::SeisData*, *Ev::SeisEvent*, *C1::SeisChannel*, *C2::SeisChannel*)

Create a SeisData structure by copying S and appending Ev.data, C1, and C2. This syntax can be used to form a new SeisData structure from arbitrary combinations of SeisData and SeisChannel objects.

SeisHdr, SeisEvent

SeisHdr()

Create an empty SeisHdr structure.

SeisHdr(*; KWs*)

Set fields at creation by specifying fieldnames as keywords.

SeisEvent()

Initialize an empty SeisEvent structure with an empty SeisHdr in .hdr and an empty SeisData in .data.

Example

Create a new SeisData structure with three channels

```
C1 = SeisChannel(name="BRASIL", id="IU.SAML.00.BHZ")
C2 = SeisChannel(name="UKRAINE", id="IU.KIEV.00.BHE")
S = SeisData(C1, C2, SeisChannel(name="CHICAGO"))
```

SeisData Indexing

Individual channels in a SeisData structure can be accessed by channel index. Indexing a single channel, e.g. **C=S[3]**, outputs a SeisChannel; indexing several outputs a new SeisData structure.

The same syntax can be used to ovewrwrite data by channel (or channel range). For example, **S[2] = T**, where T is a SeisChannel instance, replaces the second channel of S with T.

Multiple channels in a SeisData structure S can be overwritten with another SeisData structure T using **setindex!(S, T, I)**; the last input is the range of indices in S to overwrite (which must satisfy **length(I) == T.n**).

Julia is a "pass by reference" language. The precaution here is best illustrated by example: if we assign **T = S[2]**, subsequent changes to **T** modify **S[2]** in place.

1.3.2 Commands by Category

SeisIO extends a number of built-in Julia methods to work with its custom data types. In addition, many custom functions exist to simplify processing.

Append, Merge

append!(S::SeisData, U::SeisData)

Append all channels in **U** to **S**. No checks against redundancy are performed; can result in duplicate channels (fix with **merge!(S)**).

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

S += U

Merge **U** into **S**. Also works if **U** is a SeisChannel structure. Merges are based on matching channel IDs; channels in **U** without IDs in **S** are simply assigned to new channels. **merge!** and **+=** work identically for SeisData and SeisChannel instances.

Data can be merged directly from the output of any SeisIO command that outputs a compatible structure; for example, **S += readsac(sacfile.sac)** merges data from **sacfile.sac** into S.

For two channels i, j with identical ids, pairs of non-NaN data x_i, x_j with overlapping time stamps (i.e. $| t_i - t_j | < 0.5/fs$) are *averaged*.

merge!(S::SeisData)

Applying **merge!** to a single SeisData structure merges pairs of channels with identical IDs.

Delete, Extract

delete!(S::SeisData, j)

deleteat!(S::SeisData, j)

S-=j

Delete channel number(s) **j** from **S**. **j** can be an Int, UnitRange, Array{Int,1}, a String, or a Regex. In the last two cases, any channel with an id that matches **j** will be deleted; for example, **S-="CC.VALT"** deletes all channels whose IDs match **"CC.VALT"**.

T = pull(S, i)

If **i** is a string, extract the first channel from **S** with **id=i** and return it as a new SeisData structure **T**. The corresponding channel in **S** is deleted. If **i** is an integer, **pull** operates on the corresponding channel number.

purge!(S)

Remove all empty channels from **S**. Empty channels are defined as the set of all channel indices **i** s.t. **isempty(S.x[i]) = true**.

Read, Write

A = rseis(fname::String)

Read SeisIO data from **fname** into an array of SeisIO structures.

wsac(*S*)

Write SAC data from SeisData structure **S** to SAC files with auto-generated names. SAC data can only be saved to single precision.

Specify **ts=true** to write time stamps. Time stamped SAC files created by SeisIO are treated by the SAC program itself as unevenly spaced, generic **x-y** data (**LEVEN=0, IFTYPE=4**). Third-party readers might interpret timestamped files less predictably: depending on the reader, timestamped data might be loaded as the real part of a complex time series, with time stamps as the imaginary part … or the other way around … or they might not load at all.

wseis (*fname::String*, *S*)

Write SeisIO data from S to **fname**. Supports splat expansion for writing multiple objects, e.g. **wseis(fname, S, T, U)** writes **S**, **T**, and **U** to **fname**.

To write arrays of SeisIO objects to file, use "splat" notation: for example, for an array **A** of type **Array{SeisEvent,1}**, use syntax **wseis(fname, A…)**.

Search, Sort

sort!(S::SeisData, rev=false)

In-place sort by **S.id**. Specify **rev=true** to reverse the sort order.

i = findid(S, C)

Return the index of the first channel in S with id matching **C**. If **C** is a string, **findid** is equivalent to **findfirst(S.id.==C)**; if **C** is a SeisChannel, **findid** is equivalent to **findfirst(S.id.==C.id)**.

CHAPTER

TWO

FILES

2.1 File Formats

Current format support: (e = endianness; B = big, l = little, * = either)

Format	e	Command	Creates/modifies
miniSEED	B	readmseed!	existing SeisData
	B	readmseed	new SeisData
SAC	*	readsac	new SeisData
	*	sachdr	dumps header to stdout
	l	writesac	sac files on disk
SEG Y	B	readsegy (a)	new SeisData
	B	segyhdr	dumps header to stdout
UW	B	readuw	new SeisEvent
	B	uwpf!	existing SeisEvent
	B	uwpf	new SeisHdr
	B	uwdf	new SeisData
win32	B	readwin32	new SeisData

(a) Use keyword PASSCAL=true for PASSCAL SEG Y.

2.1.1 Format Descriptions

miniSEED: SEED stands for Standard for the Exchange of Earthquake Data; the data format is used by FDSN as a universal omnibus-type standard for seismic data. miniSEED is a data-only format with a limited number of blockette types.[1]

SAC: widely-used data format developed for the Seismic Analysis Code interpreter, supported in virtually every programming language.[2][3][4]

[1] FDSN SEED manual: https://www.fdsn.org/seed_manual/SEEDManual_V2.4.pdf
[2] SAC data format intro: https://ds.iris.edu/ds/nodes/dmc/kb/questions/2/sac-file-format/
[3] SAC file format: https://ds.iris.edu/files/sac-manual/manual/file_format.html
[4] SAC software homepage: https://seiscode.iris.washington.edu/projects/sac

SEG Y: standard energy industry seismic data format, developed and maintained by the Society for Exploration Geophysicists[(a)][5] A single-channel SEG Y variant format, referred to here as "PASSCAL SEG Y" was developed by PASSCAL/New Mexico Tech and is used with PASSCAL field equipment.[6]

UW: the University of Washington data format was designed for event archival. A UW event is described by a pickfile and a corresponding data file, whose names are identical except for the last character, e.g. 99062109485o, 99062109485W.[(b)]

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 channels on a single network or subnet; data within each file are stored by channel as variable-precision integers in 1s segments. Channel information for each stream is retrieved from a channel information file.[(c)][7][8]

Usage Warnings

[(a)] **SEG Y** v $\leq$ rev 1 is supported. Trace header field definitions in SEG Y are not ridigly controlled by any central authority, so some industry SEG Y files may be unreadable. Please address questions about unreadable SEG Y files to their creators.

[(b)] **UW** data format has no online documentation. Please contact the SeisIO creators or the Pacific Northwest Seismic Network (University of Washington, United States) if additional help is needed to read these files.

[(c)] **Win32** channel information files are not ridigly controlled by any central authority; thus, inconsistencies in channel parameters (e.g. gains) are known to exist. Please remember that redistribution of Win32 files is strictly prohibited by the NIED.

External References

2.1.2 File I/O Functions

`readmseed` (*fname*)

`readmseed!(S, fname)`

Read miniSEED data file `fname` into a new or existing SeisData structure.

`readsac` (*fname*[, *full=false::Bool*])

`rsac` (*fname*[, *full=false::Bool*])

Read SAC data file `fname` into a new SeisData structure. Specify keyword `full=true` to save all SAC header values in field `:misc`.

`readsegy` (*fname*[, *passcal=true::Bool*])

Read SEG Y data file `fname` into a new SeisData structure. Use keyword `passcal=true` for PASSCAL-modified SEG Y.

[5] SEG Y Wikipedia page: http://wiki.seg.org/wiki/SEG_Y
[6] PASSCAL SEG Y trace files: https://www.passcal.nmt.edu/content/seg-y-what-it-is
[7] How to use Hi-net data: http://www.hinet.bosai.go.jp/about_data/?LANG=en
[8] WIN data format (in Japanese): http://eoc.eri.u-tokyo.ac.jp/WIN/Eindex.html

readuw (*fname*)

Read UW data file into new SeisData structure. `fname` can be a pick file (ending in [a-z]), a data file (ending in W), or a file root (numeric UW event ID).

readwin32 (*fstr*, *cf*)

Read win32 data from files matching pattern `fstr` into a new SeisData structure using channel information file `cf`. `fstr` can be a path with wild card filenames, but cannot use wild card directories.

..function:: rlennasc(fname)

Read Lennartz-formatted ASCII file into a new SeisData structure.

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.

uwpf!(evt, pfname)

Parse UW event pick file into SeisEvent structure.

uwpf (*pfname*)

Parse UW event pick file `pfname` into a new SeisEvent structure.

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

wsac (*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!` as a wrapper to either FDSN or IRIS data services; for live 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, opts, q, 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, opts, q, s, si, t, to, v, w, y
Other keywords:
`--s`: Start time
`--t`: Termination (end) time

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

Event Header Query

H = FDSNevq(ot)

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

Multi-server query for the event(s) with origin time(s) closest to *ot*. Returns a SeisHdr.

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

Ev = 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, opts, pha, q, src, to, v, w
Other keywords:
`--len`: desired record length *in minutes*.

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.

Phase Onset Query

Command-line interface to IRIS online travel time calculator, which calls TauP [1-2]. Returns a matrix of strings.

Specify Δ in decimal degrees, 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

3.2 SeedLink

3.2.1 SeedLink Client

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)`

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.

Argument Syntax

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 SeedLink documentation; syntax is identical.

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.

Special Methods

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

3.2.2 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

Basic data processing operations are described below.

`autotap!(S)`

Cosine taper each channel in S around time gaps, then fill time gaps with the mean of non-NaN data points.

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.

“Safe” demean with results output to a new structure.

Remove the polynomial trend of degree n from every regularly-sampled channel i in S using a least-squares polynomial fit. Ignores NaNs. Channels of irregularly-sampled data are not (and cannot be) detrended.

Warning: detrend! does *not* check for data gaps; if this is problematic, call ungap!(S, m=true) first!

“Safe” detrend with results output to a new structure.

`equalize_resp!(S, resp_new::Array[, hc_new=HC, C=CH])`

Translate all data in SeisData structure `S` to instrument response `resp_new`. Expected structure of `resp_new` is a complex Float64 2d array with zeros in `resp[:,1]`, poles in `resp[:,2]`. If channel `i` has key `S.misc[i]["hc"]`, the corresponding value is used as the critical damping constant; otherwise a value of 1.0 is assumed.

`lcfs` (*fs::Array{Float64, 1}*)

Find *L*owest *C*ommon *fs*, the lowest sampling frequency at which data can be upsampled by repeating an integer number of copies of each sample value.

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

Merge multiple SeisData structures into S.

`prune!(S::SeisData)`

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

`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.

pull (*S::SeisData*, *i::integer*)

Extract channel i from S as a new SeisChannel struct, deleting it from S.

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

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="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*.

Related functions: time, Dates.DateTime, parsetimewin

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

Cosine taper all subsequences of regularly-sampled data and fill gaps with the mean of non-NaN data points. **m=false** leaves time gaps set to NaNs; **w=false** prevents cosine tapering.

T = ungap(S)

"Safe" ungap of SeisData object S to a new SeisData object T.

unscale!(S[, all=false])

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

CHAPTER FIVE

APPENDICES

5.1 Utility Functions

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

`distaz!(Ev::SeisEvent)`

Fill Ev with great-circle distance, azimuth, and back-azimuth for each channel. Writes to evt.data.misc.

`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]`.

`(dist, az, baz) = gcdist([lat_src, lon_src], rec)`

Compute great circle distance, azimuth, and backazimuth from source coordinates `[lat_src, lon_src]` to receiver coordinates in `rec` using the Haversine formula. `rec` must be a two-column matix arranged [lat lon]. Returns a tuple of arrays.

`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`(*path*)

Wrapper to *sh /bin/ls -1*; returns output as a string array. In Windows, provides similar functionality to Unix ls. `ls()` with no arguments lists contents of cwd.

`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.

webhdr()

Generate a Dict{String,String} to set UserAgent in web requests.

"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.

5.1.1 RandSeis

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

- Channels have SEED-compliant IDs, sampling frequencies, and data types.
- Channel data are randomly generated.
- Some time gaps are automatically inserted into regularly-sampled data.
- Instrument location parameters are randomly set.

C = 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.

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.

H = randSeisHdr()

Generate a SeisHdr structure filled with random values.

5.2 Structure and Field Descriptions

5.2.1 SeisChannel Fields

Name	Type	Meaning
id	String	unique channel ID formatted *net.sta.loc.cha*
name	String	freeform channel name string
src	String	description of data source
units	String	units of dependent variable[1]
fs	Float64	sampling frequency in Hz
gain	Float64	scalar to convert x to SI units in flat part of power spectrum[2]
loc	Array{Float64,1}	sensor location: [lat, lon, ele, az, inc][3]
resp	Array{Complex {Float64},2}	complex instrument response[4]
misc	Dict{String,Any}	miscellaneous information[5]
notes	Array{String,1}	timestamped notes
t	Array{Int64,2}	time gaps *(see below)*
x	Array{Float64,1}	univariate data

Table Footnotes

5.2.2 SeisData Fields

As SeisChannel, plus

Name	Type	Meaning
n	Int64	number of channels
c	Array{TCPSocket,1}	array of TCP connections

Time Convention

The units of `t` are *integer microseconds*, measured from Unix epoch time (1970-01-01T00:00:00.000).

For *regularly sampled* data (`fs > 0.0`), each `t` is a sparse delta-compressed representation of *time gaps* in the corresponding `x`. The first column stores indices of gaps; the second, gap lengths.

[1] Use UCUM-compliant abbreviations wherever possible.
[2] Gain has an identical meaning to the "Stage 0 gain" of FDSN XML.
[3] Azimuth is measured clockwise from North; incidence of 0° = vertical; both use degrees.
[4] Zeros in `:resp[i][:,1]`, poles in `:resp[i][:,2]`.
[5] Arrays in `:misc` should each contain a single Type (e.g. Array{Float64,1}, never Array{Any,1}). See the *SeisIO file format description* for a full list of allowed value types in :misc.

Within each time field, `t[1,2]` stores the time of the first sample of the corresponding `x`. The last row of each `t` should always take the form ‘ *[length(x) 0]*‘. Other rows take the form `[(starting index of gap) (length of gap)]`.

For *irregularly sampled data* (`fs = 0`), `t[:,2]` is a dense representation of *time stamps for each sample*.

5.2.3 SeisHdr Fields

Name	Type	Meaning
id	Int64	numeric event ID
ot	DateTime	origin time
loc	Array{Float64, 1}	hypocenter
mag	Tuple{Float32, String}	magnitude, scale
int	Tuple{UInt8, String}	intensity, scale
mt	Array{Float64, 1}	moment tensor: (1-6) tensor, (7) scalar moment, (8) %dc
np	Array{Tuple{Float64, Float64, Float64},1}	nodal planes
pax	Array{Tuple{Float64, Float64, Float64},1}	principal axes, ordered P, T, N
src	String	data source (e.g. url/filename)

5.2.4 SeisEvent Fields

Name	Type	Meaning
hdr	SeisHdr	event header
data	SeisData	event data

5.3 SeisIO File Format

Files are written in little-endian byte order.

Table 1: Abbreviations used in this section

Var	Meaning	Julia	C <stdint.h>
c	unsigned 8-bit character	Char	unsigned char
f32	32-bit float	Float32	float
f64	64-bit float	Float64	double
i64	signed 64-bit integer	Int64	int64_t
u8	unsigned 8-bit int	UInt8	uint8_t
u32	unsigned 32-bit int	UInt32	int32_t
u64	unsigned 64-bit int	UInt64	uint64_t
u(8)	unsigned 8-bit integer	UInt8	uint8_t
i(8)	signed 8-bit integer	Int8	int8_t
f(8)	8-bit float	Float8	float or double

5.3.1 File header

Table 2: File header (14 bytes + TOC)

Var	Meaning	T	N
	"SEISIO"	c	6
`V`	SeisIO version	f32	1
`jv`	Julia version	f32	1
`J`	# of SeisIO objects in file	u32	1
`C`	Character codes for each object	c	J
`B`	Byte indices for each object	u64	J

The Julia version stores VERSION.major.VERSION.minor as a Float32, e.g. v0.5 is stored as 0.5f0; SeisIO version is stored similarly.

Table 3: Object codes

Char	Meaning
'D'	SeisData
'H'	SeisHdr
'E'	SeisEvent

5.3.2 SeisHdr

Structural overview:

```
Int64_vals
:mag[1]        # Float32
Float64_vals
UInt8_vals
:misc
```

Table 4: Int64 values

Var	Meaning
id	event id
ot	origin time in integer μs from Unix epoch
L_int	length of intensity scale string
L_src	length of src string
L_notes	length of notes string

Magnitude is stored as a Float32 after the Int64 values.

Table 5: Float64 values

Var	N	Meaning
loc	3	lat, lon, dep
mt	8	tensor, scalar moment, %dc
np	6	np (nodal planes: 1st, 2nd)
pax	9	pax (principal axes: P, T, N)

Table 6: UInt8 values

Var	N	Meaning
msc	2	magnitude scale characters
c	1	separator for notes
i	1	intensity value
i_sc	L_int	intensity scale string
src	L_src	`:src` as a string
notes	L_notes	`:notes` joined a string with delimiter `c`

Entries in Misc are stored after UInt8 values. See below for details.

5.3.3 SeisData

Structural overview:

```
S.n          # UInt32
# Repeated for each channel
Int64_vals
Float64_vals
UInt8_vals    # including compressed S.x
:misc
```

S.x is compressed with BloscLZ before writing to disk.

Channel data

Table 7: Int64 values

Var	N	Meaning
L_t		length(S.t)
r		length(S.resp)
L_units		length(S.units)
L_src		length(S.src)
L_name		length(S.name)
L_notes		length of notes string
lxc		length of BloscLZ-compressed S.x
L_x		length(S.x)
t	L_t	S.t

Table 8: Float64 values

Var	N	Meaning
fs	1	S.fs
gain	1	S.gain
loc	5	S.loc (lat, lon, dep, az, inc)
resp	2*r	real(S.resp[:]) followed by imag(S.resp[:])

Convert resp with `resp = rr[1:r] + im*rr[r+1:2*r]` and reshape to a two-column array with `r` rows. The first column of the new, complex-valued `resp` field holds zeros, the second holds poles.

Table 9: UInt8 values

Var	N	Meaning
c	1	separator for notes
ex	1	type code for S.x
id	15	S.id
units	L_units	S.units
src	L_src	S.src
name	L_name	S.name
notes	L_notes	S.notes joined as a string with delimiter `c`
xc	lxc	Blosc-compressed S.x

S.misc is written last, after the compressed S.x

Storing misc

`:misc` is a Dict{String,Any} for both SeisData and SeisHdr, with limited support for key value types. Structural overview:

```
L_keys
char_separator  # for keys
keys            # joined as a string
# for each key k
type_code       # UInt8 code for misc[k]
value           # value of misc[k]
```

Table 10: `:misc` keys

Var	Meaning	T	N
`L`	length of keys string	i64	1
`p`	character separator	u8	1
`K`	string of keys	u8	p

Table 11: Supported `:misc` value Types

code	value Type	code	value Type
0	Char	128	Array{Char,1}
1	String	129	Array{String,1}
16	UInt8	144	Array{UInt8,1}
17	UInt16	145	Array{UInt16,1}
18	UInt32	146	Array{UInt32,1}
19	UInt64	147	Array{UInt64,1}
20	UInt128	148	Array{UInt128,1}
32	Int8	160	Array{Int8,1}
33	Int16	161	Array{Int16,1}
34	Int32	162	Array{Int32,1}
35	Int64	163	Array{Int64,1}
36	Int128	164	Array{Int128,1}
48	Float16	176	Array{Float16,1}
49	Float32	177	Array{Float32,1}
50	Float64	178	Array{Float64,1}
80	Complex{UInt8}	208	Array{Complex{UInt8},1}
81	Complex{UInt16}	209	Array{Complex{UInt16},1}
82	Complex{UInt32}	210	Array{Complex{UInt32},1}
83	Complex{UInt64}	211	Array{Complex{UInt64},1}
84	Complex{UInt128}	212	Array{Complex{UInt128},1}
96	Complex{Int8}	224	Array{Complex{Int8},1}
97	Complex{Int16}	225	Array{Complex{Int16},1}
98	Complex{Int32}	226	Array{Complex{Int32},1}
99	Complex{Int64}	227	Array{Complex{Int64},1}
100	Complex{Int128}	228	Array{Complex{Int128},1}
112	Complex{Float16}	240	Array{Complex{Float16},1}
113	Complex{Float32}	241	Array{Complex{Float32},1}
114	Complex{Float64}	242	Array{Complex{Float64},1}

Julia code for converting between data types and UInt8 type codes is given below.

```
findtype(c::UInt8, T::Array{Type,1}) = T[findfirst([sizeof(i)==2^c for i in T])]
function code2typ(c::UInt8)
  t = Any::Type
  if c >= 0x80
    t = Array{code2typ(c-0x80)}
  elseif c >= 0x40
    t = Complex{code2typ(c-0x40)}
  elseif c >= 0x30
    t = findtype(c-0x2f, Array{Type,1}(subtypes(AbstractFloat)))
  elseif c >= 0x20
    t = findtype(c-0x20, Array{Type,1}(subtypes(Signed)))
  elseif c >= 0x10
    t = findtype(c-0x10, Array{Type,1}(subtypes(Unsigned)))
  elseif c == 0x01
    t = String
  elseif c == 0x00
    t = Char
  else
    t = Any
  end
  return t
end

tos(t::Type) = round(Int64, log2(sizeof(t)))
function typ2code(t::Type)
  n = 0xff
  if t == Char
    n = 0x00
  elseif t == String
    n = 0x01
  elseif t <: Unsigned
    n = 0x10 + tos(t)
  elseif t <: Signed
    n = 0x20 + tos(t)
  elseif t <: AbstractFloat
    n = 0x30 + tos(t)-1
  elseif t <: Complex
    n = 0x40 + typ2code(real(t))
  elseif t <: Array
    n = 0x80 + typ2code(eltype(t))
  end
  return UInt8(n)
end
```

Type "Any" is provided as a default; it is not supported.

Standard Types in `:misc`

Most values in `:misc` are saved as a *UInt8 code* followed by the value itself.

Unusual Types in `:misc`

The tables below describe how to read non-bitstype data into `:misc`.

Table 12: Array{String}

Var	Meaning	T	N
nd	array dimensionality	u8	1
d	array dimensions	i64	nd
	if d!=[0]:		
sep	string separator	c	1
L_S	length of char array	i64	1
S	string array as chars	u8	L_S

If d=[0], indicating an empty String array, set S to an empty String array and do not read sep, L_S, or S.

Table 13: Array{Complex}

Var	Meaning	T	N
nd	array dimensionality	u8	1
d	array dimensions	i64	nd
rr	real part of array	τ	d
ii	imaginary part of array	τ	d

Here, τ denotes the type of the real part of one element of v.

Table 14: Array{Real}

Var	Meaning	T	N
nd	array dimensionality	u8	1
d	array dimensions	i64	nd
v	array values	τ	d

Here, τ denotes the type of one element of v.

Table 15: String

Var	Meaning	T	N
L_S	length of string	i64	1
S	string	u8	L_S

5.3.4 SeisEvent

A SeisEvent structure is stored as a SeisHdr object followed by a SeisData object. However, the combination of SeisHdr and SeisData objects that comprises a SeisEvent object counts as one object, not two, in the file TOC.

5.4 Data Requests Syntax

5.4.1 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 ID string 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
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.

5.4.2 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)

5.5 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 current cannot be modified, but this may change as the Julia language matures.

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
nev	1	I	number of events returned per query
opts	""	S	user-specified options[2]
q	'B'	C	data quality[3]
pha	"P"	S	seismic phase arrival times to retrieve
reg	[-90.0, 90.0,	A{F,1}	geographic search region[4]
	-180.0, 180.0,		
	-30.0, 660.0]		
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]

Table Footnotes

5.5.1 SeedLink Keywords

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]
safety	0x00	U8	safety check level[9]
x_on_err	true	Bool	exit on error?

[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] Queries to some FDSN servers will fail with **-q='R'**.

[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.

[5] Not used with IRISWS.

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

[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 even if data processing fails.

Table Footnotes

5.6 Examples

5.6.1 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)
```

5.6.2 FDSN station query

A sample FDSN station query

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

5.6.3 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)
```

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

[9] Before initiating a SeedLink connection, **safety=0x01** calls **has_sta**, **safety=0x02** calls **has_stream**

5.6.4 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")
```

5.6.5 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