rf Documentation

Release 0.4.0

Tom Eulenfeld

June 30, 2016

Contents

The receiver function method is a popular technique to investigate crustal and upper mantle velocity discontinuities. Basic concept of the method is that a small part of incident P-waves from a teleseismic event gets converted to S-waves at significant discontinuities under the receiver (for P-receiver functions). These converted Ps phases arrive at the station after the main P phase. The response function of the receiver side (receiver function) is constructed by removing the source and deep mantle propagation effects. Firstly, the S-wave field is separated from the P-wave field by a rotation from the station coordinate system (ZNE - vertical, north, east) to the wave coordinate system (LQT - P-wave polarization, approx. SV-wave polarization, SH-wave polarization). Secondly, the waveform on the L component is deconvolved from the other components, which removes source side and propagation effects. The resulting functions are the Q and T component of the P receiver function. Multiple reflected waves are also visible in the receiver function. The conversion points of the rays are called piercing points.

For a more detailed description of the working flow see e.g. chapter 4.1 of this dissertation.

Left: In a two-layer-model part of the incoming P-wave is converted to a S-wave at the layer boundary. Major multiples are Pppp, Ppps and Ppss.

Right: Synthetic receiver function of Q component in a two-layer-model.

CHAPTER 1

Installation

Dependencies of rf are

- ObsPy and some of its dependencies,
- cartopy, geographiclib, shapely,
- toeplitz (time domain deconvolution), tqdm,
- obspyh5 for hdf5 file support (optional).

After the installation of Obspy rf can be installed with

```
pip install rf
```

The tests can be run with the script

```
rf-runtests
```

To install the development version of obspy download the source code and run

```
python setup.py install
```

CHAPTER 2

Using the underlying Python module

The main functionality is provided by the class `RFStream` which is derived from ObsPy's `Stream` class.

The canonical way to load a waveform file into a RFStream is to use the `read_rf()` function.

```
>>> from rf import read_rf
>>> stream = read_rf('myfile.SAC')
```

If you already have an ObsPy Stream and you want to turn it into a RFStream use the generator of RFStream:

```
>>> from rf import RFStream
>>> stream = RFStream(obspy_stream)
```

The stream is again written to disc as usual by its write method:

```
>>> stream.write('outfile', 'SAC')
```

The RFStream object inherits a lot of useful methods from its ObsPy ancestor (e.g. filter, taper, simulate, ...).

The module automatically maps important (for rf calculation) header information from the stats object attached to every trace to the format specific headers. At the moment only SAC and SH/Q headers are supported. When initializing an RFStream the header information in the format specific headers are written to the stats object and before writing the information stored in the stats object is written back to the format specific headers. In this way the important header information is guaranteed to be saved in the waveform files. The following table reflects the mapping:

stats	SH/Q	SAC
station_latitude	COMMENT	stla
station_longitude	COMMENT	stlo
station_elevation	COMMENT	stel
event_latitude	LAT	evla
event_longitude	LON	evlo
event_depth	DEPTH	evdp
event_magnitude	MAGNITUDE	mag
event_time	ORIGIN	o
onset	P-ONSET	a
type	COMMENT	kuser0
phase	COMMENT	kuser1
moveout	COMMENT	kuser2
distance	DISTANCE	gcarc
back_azimuth	AZIMUTH	baz
inclination	INCI	user0
slowness	SLOWNESS	user1
pp_latitude	COMMENT	user2
pp_longitude	COMMENT	user3
pp_depth	COMMENT	user4
box_pos	COMMENT	user5
box_length	COMMENT	user6

Note: Q-file header COMMENT is used for storing some information, because the Q format has a shortage of predefined headers.

Note: Alternatively the hdf5 file format can be used. It is supported via the obspyh5 package. In this case all supported stats entries are automatically attached to the stored data. The plugin also saves entries not listed here (e.g. processing information).

The first task when calculating receiver functions is calculating some ray specific values like azimuth and epicentral distance. An appropriate stats dictionary can be calculated with *rfstats()*:

```
>>> from rf import rfstats
>>> stats = rfstats(station=station, event=event, phase='P', dist_range=(30,90))
>>> for tr in stream:
>>>     tr.stats.update(stats)
```

or if the station and event information is already stored in the stats object:

```
>>> rfstats(stream)
```

A typical workflow for P receiver function calculation and writing looks like

```
>>> stream.filter('bandpass', freqmin=0.05, freqmax=1.)
>>> stream.rf()
>>> stream.write('rf', 'Q')
```

rf can also calculate S receiver functions (not much tested):

```
>>> stream.rf(method='S')
```

When calling stream.rf the following operations are performed depending on the given kwargs:

- filtering

- trimming data to window relative to onset
- downsampling
- rotation
- deconvolution

Please see *RFStream.rf()* for a more detailed description. RFStream provides the possibility to perform moveout correction, piercing point calculation and profile stacking.

Command line tool for batch processing

The rf package provides a command line utility ‘rf’ which runs all the necessary steps to perform receiver function calculation. Use `rf -h` and `rf {your_command} -h` to discover the interface. All you need is an inventory file (StationXML) and a file with events (QuakeML) you want to analyze.

The command

```
rf create
```

creates a *template configuration file* in the current directory. This file is in JSON format and well documented. After adapting the file to your needs you can use the various commands of rf to perform different tasks (e.g. receiver function calculation, plotting).

To create the tutorial with a small included dataset and working configuration you can use

```
rf create --tutorial
```

Now start using rf ..., e.g.

```
rf data calc myrf
rf moveout myrf myrfmout
rf plot myrfmout myrfplot
rf --moveout-phase Psss moveout myrf myrfPsssmout
```

CHAPTER 4

Miscellaneous

Please feel free to request features, report bugs or contribute code on GitHub. The code is continuously tested by travis-ci. The test status of this version is .

CHAPTER 5

API Documentation

5.1 `rfstream` Module

Classes and functions for receiver function calculation.

class `rf.rfstream.`**`RFStream`**(*traces=None*, *warn=True*)

Bases: `obspy.core.stream.Stream`

Class providing the necessary functions for receiver function calculation.

Parameters

- **`traces`** – list of traces, single trace or stream object
- **`warn`** – display warnings when reading and mapping rf headers

To initialize a RFStream from a Stream object use

```
>>> rfstream = RFStream(stream)
```

To initialize a RFStream from a file use

```
>>> rfstream = read_rf('test.SAC')
```

Format specific headers are loaded into the stats object of all traces.

`deconvolve`(**args*, ***kwargs*)

Deconvolve source component of stream.

All args and kwargs are passed to the function `deconvolve()`.

`method`

Property for used rf method, 'P' or 'S'

`moveout`(*phase=None*, *ref=6.4*, *model='iasp91'*)

In-place moveout correction to a reference slowness.

Needs stats attributes slowness and onset.

Parameters

- **`phase`** – 'Ps', 'Sp', 'Ppss' or other multiples, if None is set to 'ps' or 'Sp' depending on method
- **`ref`** – reference ray parameter in s/deg
- **`model`** – Path to model file (see `SimpleModel`, default: iasp91)

plot_profile(**args*, ***kwargs*)
Create receiver function profile plot.

See `imaging.plot_profile()` for help on arguments.

plot_rf(**args*, ***kwargs*)
Create receiver function plot.

See `imaging.plot_rf()` for help on arguments.

ppoints(*pp_depth*, *pp_phase=None*, *model='iasp91'*)
Return coordinates of piercing point calculated by 1D ray tracing.

Piercing point coordinates are stored in the stats attributes plat and plon. Needs stats attributes station_latitude, station_longitude, slowness and back_azimuth.

Parameters

- **pp_depth** – depth of interface in km
- **pp_phase** – 'P' for piercing points of P wave, 'S' for piercing points of S wave or multiples, if None will be set to 'P' or 'S' depending on method
- **model** – path to model file (see `SimpleModel`, default: iasp91)

Returns NumPy array with coordinates of piercing points

Note: `phase='S'` is usually wanted for P receiver functions and `'P'` for S receiver functions.

profile(**args*, ***kwargs*)
Return profile of receiver functions in the stream.

See `profile.profile()` for help on arguments.

rf(*method=None*, *filter=None*, *trim=None*, *downsample=None*, *rotate='ZNE->LQT'*, *deconvolve='time'*, *source_components=None*, ***kwargs*)
Calculate receiver functions in-place.

Parameters

- **method** – 'P' for P receiver functions, 'S' for S receiver functions, if None method will be determined from the phase
- **filter** (*dict*) – filter stream with its `filter` method and given kwargs
- **trim** (*tuple (start, end)*) – trim stream relative to P- or S-onset with `trim2()` (seconds)
- **downsample** (*float*) – downsample stream with its `decimate()` method to the given frequency
- **rotate** (*'ZNE->LQT' or 'NE->RT'*) – rotate stream with its `rotate()` method with the angles given by the back_azimuth and inclination attributes of the traces stats objects. You can set these to your needs or let them be computed by `rfstats()`.
- **deconvolve** – 'time' or 'freq' for time or frequency domain deconvolution by the streams `deconvolve()` method. See `deconvolve()`, `deconvt()` and `deconvf()` for further documentation.
- **source_components** – parameter is passed to deconvolve. If None, source components will be chosen depending on method.
- ****kwargs** – all other kwargs not mentioned here are passed to deconvolve

After performing the deconvolution the Q/R and T components are multiplied by -1 to get a positive phase for a Moho-like positive velocity contrast. Furthermore for method='S' all components are mirrored at t=0 for a better comparison with P receiver functions. See source code of this function for the default deconvolution windows.

`slice2`(*starttime=None*, *endtime=None*, *reftime=None*, *keep_empty_traces=False*, ***kwargs*)
Alternative slice method accepting relative times.

See `slice()` and `trim2()`.

`stack`()
Return stack of traces with same seed ids.

Traces with same id need to have the same number of datapoints. Each trace in the returned stream will correspond to one unique seed id.

`trim2`(*starttime=None*, *endtime=None*, *reftime=None*, ***kwargs*)
Alternative trim method accepting relative times.

See `trim()`.

Parameters

- **`endtime`** (`starttime,`) – accept UTCDateTime or seconds relative to reftime
- **`reftime`** – reference time, can be an UTCDateTime object or a string. The string will be looked up in the stats dictionary (e.g. 'starttime', 'endtime', 'onset').

`type`
Property for the type of stream, 'rf', 'profile' or None

`write`(*filename*, *format*, ***kwargs*)
Save stream to file including format specific headers.

See `Stream.write()` in ObsPy.

class `rf.rfstream.`**`RFTrace`**(*data=array([], dtype=float64)*, *header=None*, *trace=None*, *warn=True*)
Bases: `obspy.core.trace.Trace`

Class providing the Trace object for receiver function calculation.

`write`(*filename*, *format*, ***kwargs*)
Save current trace into a file including format specific headers.

See `Trace.write()` in ObsPy.

`rf.rfstream.`**`obj2stats`**(*event=None*, *station=None*)
Map event and station object to stats with attributes.

Parameters

- **`event`** – ObsPy `Event` object
- **`station`** – station object with attributes latitude, longitude and elevation

Returns `stats` object with station and event attributes

`rf.rfstream.`**`read_rf`**(*pathname_or_url=None*, *format=None*, ***kwargs*)
Read waveform files into RFStream object.

See `read()` in ObsPy.

`rf.rfstream.`**`rfstats`**(*obj=None*, *event=None*, *station=None*, *phase='P'*, *dist_range='default'*, *tt_model='iasp91'*, *pp_depth=None*, *pp_phase=None*, *model='iasp91'*)
Calculate ray specific values like slowness for given event and station.

Parameters

- **obj** – Stats object with event and/or station attributes. Can be None if both event and station are given. It is possible to specify a stream object, too. Then, rfstats will be called for each Trace.stats object and traces outside dist_range will be discarded.
- **event** – ObsPy Event object
- **station** – station object with attributes latitude, longitude and elevation
- **phase** – string with phase. Usually this will be 'P' or 'S' for P and S receiver functions, respectively.
- **dist_range** (*tuple of length 2*) – if epicentral of event is not in this intervall, None is returned by this function,

  if phase == 'P' defaults to (30, 90),

  if phase == 'S' defaults to (50, 85)
- **tt_model** – model for travel time calculation. (see the obspy.taup module, default: iasp91)
- **pp_depth** – Depth for piercing point calculation (in km, default: None -> No calculation)
- **pp_phase** – Phase for pp calculation (default: 'S' for P-receiver function and 'P' for S-receiver function)
- **model** – Path to model file for pp calculation (see *SimpleModel*, default: iasp91)

Returns Stats object with event and station attributes, distance, back_azimuth, inclination, onset and slowness or None if epicentral distance is not in the given interval. Stream instance if stream was specified instead of stats.

5.2 deconvolve Module

Frequency and time domain deconvolution.

rf.deconvolve.**deconvf**(*rsp_list*, *src*, *sampling_rate*, *waterlevel=0.05*, *gauss=2.0*, *tshift=10.0*, *pad=0*, *length=None*, *normalize=0*, *return_info=False*)

Frequency-domain deconvolution using waterlevel method.

Deconvolve src from arrays in rsp_list.

Parameters

- **rsp_list** – either a list of arrays containing the response functions or a single array
- **src** – array with source function
- **sampling_rate** – sampling rate of the data
- **waterlevel** – waterlevel to stabilize the deconvolution
- **gauss** – Gauss parameter of Low-pass filter
- **tshift** – delay time 0s will be at time tshift afterwards
- **pad** – multiply number of samples used for fft by 2**pad
- **length** – number of data points in results, optional
- **normalize** – normalize all results so that the maximum of the trace with supplied index is 1. Set normalize to 'src' to normalize for the maximum of the prepared source. Set normalize to None for no normalization.

- **return_info** – return additionally a lot of different parameters in a dict for debugging purposes

Returns (list of) array(s) with deconvolution(s)

rf.deconvolve.**deconvolve**(*stream*, *method='time'*, *source_components='LZ'*, *response_components=None*, *winsrc='P'*, ***kwargs*)

Deconvolve one component of a stream from other components.

The deconvolutions are written to the data arrays of the stream. To keep the original data use the copy method of Stream. The stats dictionaries of the traces inside stream must have an 'onset' entry with a `UTCDateTime` object. This will be used for determining the data windows.

Parameters

- **stream** – stream including responses and source
- **method** – 'time' -> use time domain deconvolution in `deconvt()`,

  'freq' -> use frequency domain deconvolution in `deconvf()`
- **source_components** – names of components identifying the source traces, e.g. 'LZ' for P receiver functions and 'QR' for S receiver functions
- **response_components** – names of components identifying the response traces (default None: all traces are used as response)
- **winsrc** (*tuple (start, end, taper)*) – data window for source function, in seconds relative to onset. The function will be cosine tapered at both ends (seconds).

  winsrc can also be a string ('P' or 'S'). In this case the function defines a source time window appropriate for this type of receiver function and deconvolution method (see source code for details).
- ****kwargs** – other kwargs are passed to the underlying deconvolution functions `deconvt()` and `deconvf()`

Note: If parameter normalize is not present in kwargs and source component is not excluded from the results by response_components, results will be normalized such that the maximum of the deconvolution of the trimmed and tapered source from the untouched source is 1. If the source is excluded from the results, the normalization will performed against the first trace in results.

rf.deconvolve.**deconvt**(*rsp_list*, *src*, *shift*, *spiking=1.0*, *length=None*, *normalize=0*)

Time domain deconvolution.

Deconvolve src from arrays in rsp_list. Calculate Toeplitz auto-correlation matrix of source, invert it, add noise and multiply it with cross-correlation vector of response and source.

In one formula:

```
RF = (STS + spiking*I)^-1 * STR

N... length
     ( S0   S-1  S-2 ... S-N+1 )
     ( S1   S0   S-1 ... S-N+2 )
S =  ( S2   ...                )
     ( ...                     )
     ( SN-1 ...           S0   )
R = (R0 R1 ... RN-1)^T
RF = (RF0 RF1 ... RFN-1)^T
S... source matrix (shape N*N)
```

```
    R... response vector (length N)
    RF... receiver function (deconvolution) vector (length N)
    STS = S^T*S = symmetric Toeplitz autocorrelation matrix
    STR = S^T*R = cross-correlation vector
    I... Identity
```

Parameters

- **`rsp_list`** – either a list of arrays containing the response functions or a single array
- **`src`** – array of source function
- **`shift`** – shift the source by that amount of samples to the left side to get onset in RF at the desired time (negative -> shift source to the right side)

  shift = (middle of rsp window - middle of src window) + (0 - middle rf window)
- **`spiking`** – random noise added to autocorrelation (eg. 1.0, 0.1)
- **`length`** – number of data points in results
- **`normalize`** – normalize all results so that the maximum of the trace with supplied index is 1. Set normalize to None for no normalization.

Returns (list of) array(s) with deconvolution(s)

5.3 `imaging` Module

Functions for receiver function plotting.

`rf.imaging.`**`plot_ppoints`**(*ppoints*, *inventory=None*, *label_stations=True*, *ax=None*, *crs=None*, ***kwargs*)

Plot piercing points with stations.

Parameters

- **`ppoints`** – list of (lat, lon) tuples of piercing points
- **`label_stations`** (*`inventory,`*) – plot stations, see *`plot_stations`*
- **`ax`** – geoaxes (default None: new ax will be created)
- **`crs`** – coordinate reference system for new geoaxis, (default: None, then AzimuthalEquidistant projection with appropriate center is used.)
- **`**kwargs`** – other kwargs are passed to ax.scatter() call

`rf.imaging.`**`plot_profile`**(*profile*, *fname=None*, *scale=1*, *fillcolors=('r', 'b')*, *trim=None*, *top=None*, *moveout_model='iasp91'*)

Plot receiver function profile.

Parameters

- **`profile`** – stream holding the profile
- **`fname`** – filename to save plot to. Can be None. In this case the figure is left open.
- **`scale`** – scale for individual traces
- **`fillcolors`** – fill colors for positive and negative wiggles
- **`trim`** – trim stream relative to onset before plotting using *`slice2()`*

- **top** – show second axes on top of profile with additional information. Valid values: 'hist' - Plot histogram showing the number of receiver functions stacked in the corresponding bin
- **moveout_model** – string with model filename. Will be loaded into a *SimpleModel* object to calculate depths for tick labels.

`rf.imaging.`**`plot_profile_map`**(*boxes*, *inventory=None*, *label_stations=True*, *ppoints=None*, *ax=None*, *crs=None*, ***kwargs*)

Plot profile map with stations and piercing points.

Parameters

- **boxes** – boxes created with *get_profile_boxes()*
- **label_stations** (*inventory,*) – plot stations, see *plot_stations*
- **ppoints** – list of (lat, lon) tuples of piercing points, see *plot_ppoints*
- **ax** – geoaxes (default None: new ax will be created)
- **crs** – coordinate reference system for new geoaxis, (default: None, then AzimuthalEquidistant projection with appropriate center is used.)
- ****kwargs** – other kwargs are passed to ax.add_geometries() call

`rf.imaging.`**`plot_rf`**(*stream*, *fname=None*, *fig_width=7.0*, *trace_height=0.5*, *stack_height=0.5*, *scale=1*, *fillcolors=(None*, *None)*, *trim=None*, *info=(('back_azimuth'*, *u'baz (\xb0)'*, *'b')*, *('distance'*, *u'dist (\xb0)'*, *'r'))*)

Plot receiver functions.

Parameters

- **stream** – stream to plot
- **fname** – filename to save plot to. Can be None. In this case the figure is left open.
- **fig_width** – width of figure in inches
- **trace_height** – height of one trace in inches
- **stack_height** – height of stack axes in inches
- **scale** – scale for individual traces
- **fillcolors** – fill colors for positive and negative wiggles
- **trim** – trim stream relative to onset before plotting using *slice2()*
- **info** – Plot one additional axes showing maximal two entries of the stats object. Each entry in this list is a list consisting of three entries: key, label and color. info can be None. In this case no additional axes is plotted.

`rf.imaging.`**`plot_stations`**(*inventory*, *label_stations=True*, *ax=None*, *crs=None*, ***kwargs*)

Plot stations.

Parameters

- **inventory** – station inventory
- **label_stations** – weather to label stations
- **ax** – geoaxes (default None: new ax will be created)
- **crs** – coordinate reference system for new geoaxis, (default: None, then AzimuthalEquidistant projection with appropriate center is used.)
- ****kwargs** – other kwargs are passed to ax.scatter() call

5.4 `profile` Module

Functions for receiver function profile calculation.

`rf.profile.`**`get_profile_boxes`**(*latlon0*, *azimuth*, *bins*, *width=2000*)

Create 2D boxes for usage in *`profile()`* function.

Parameters

- **`latlon0`** (*`tuple`*) – coordinates of starting point of profile
- **`azimuth`** – azimuth of profile direction
- **`bins`** (*`tuple`*) – Edges of the distance bins in km (e.g. (0, 10, 20, 30))
- **`width`** – width of the boxes in km (default: large)

Returns List of box dicts. Each box has the entries 'poly' (shapely polygon with lonlat corners), 'length' (length in km), 'pos' (midpoint of box in km from starting coordinates), 'latlon' (midpoint of box as coordinates)

`rf.profile.`**`profile`**(*stream*, *boxes*, *crs=None*)

Stack traces in stream by piercing point coordinates in defined boxes.

Parameters

- **`stream`** – stream with pre-calculated piercing point coordinates
- **`boxes`** – boxes created with *`get_profile_boxes()`*
- **`crs`** – cartopy projection (default: AzimuthalEquidistant)

Returns profile stream

5.5 `simple_model` Module

Simple move out and piercing point calculation.

class `rf.simple_model.`**`SimpleModel`**(*z*, *vp*, *vs*, *n=None*)

Bases: `object`

Simple 1D velocity model for move out and piercing point calculation.

Parameters

- **`z`** – depths in km
- **`vp`** – P wave velocities at provided depths in km/s
- **`vs`** – S wave velocities at provided depths in km/s
- **`n`** – number of support points between provided depths

All arguments can be of type numpy.ndarray or list.

`calculate_delay_times`(*slowness*, *phase='PS'*)

Calculate delay times between direct wave and converted phase.

Parameters

- **`slowness`** – ray parameter in s/deg
- **`phase`** – Converted phase or multiple (e.g. Ps, Pppp)

Returns delay times at different depths

calculate_vertical_slowness (*slowness*, *phase='PS'*)
Calculate vertical slowness of P and S wave.

Parameters

- **slowness** – slowness in s/deg
- **phase** – Weather to calculate only P, only S or both vertical slownesses

Returns vertical slowness of P wave, vertical slowness of S wave at different depths (z attribute of model instance)

moveout (*stream*, *phase='Ps'*, *ref=6.4*)
In-place moveout correction to reference slowness.

Parameters

- **stream** – stream with stats attributes onset and slowness.
- **phase** – 'Ps', 'Sp', 'Ppss' or other multiples
- **ref** – reference slowness (ray parameter) in s/deg

ppoint (*stats*, *depth*, *phase='S'*)
Calculate latitude and longitude of piercing point.

Piercing point coordinates and depth are saved in the pp_latitude, pp_longitude and pp_depth entries of the stats object or dictionary.

Parameters

- **stats** – Stats object or dictionary with entries slowness, back_azimuth, station_latitude and station_longitude
- **depth** – depth of interface in km
- **phase** – 'P' for piercing point of P wave, 'S' for piercing point of S wave. Multiples are possible, too.

Returns latitude and longitude of piercing point

ppoint_distance (*depth*, *slowness*, *phase='S'*)
Calculate horizontal distance between piercing point and station.

Parameters

- **depth** – depth of interface in km
- **slowness** – ray parameter in s/deg
- **phase** – 'P' or 'S' for P wave or S wave. Multiples are possible.

Returns horizontal distance in km

stretch_delay_times (*slowness*, *phase='Ps'*, *ref=6.4*)
Stretch delay times of provided slowness to reference slowness.

First, calculate delay times (time between the direct wave and the converted phase or multiples at different depths) for the provided slowness and reference slowness. Secondly, stretch the the delay times of provided slowness to reference slowness.

Parameters

- **slowness** – ray parameter in s/deg
- **phase** – 'Ps', 'Sp' or multiples
- **ref** – reference ray parameter in s/deg

Returns original delay times, delay times stretched to reference slowness

`rf.simple_model.`**`load_model`**(*fname='iasp91'*)

Load model from file.

Parameters **`fname`** – path to model file or 'iasp91'

Returns *`SimpleModel`* instance

The model file should have 4 columns with depth, vp, vs, n. The model file for iasp91 starts like this:

```
#IASP91 velocity model
#depth  vp    vs   n
 0.00  5.800 3.360 0
 0.00  5.800 3.360 0
10.00  5.800 3.360 4
```

5.6 `util` Module

Utility functions and classes for receiver function calculation.

`rf.util.`**`DEG2KM`** **= 111.2**

Conversion factor from degrees epicentral distance to km

class `rf.util.`**`IterMultipleComponents`**(*stream*, *key=None*, *number_components=None*)

Bases: `object`

Return iterable to iterate over associated components of a stream.

Parameters

- **`stream`** – Stream with different, possibly many traces. It is split into substreams with the same seed id (only last character i.e. component may vary)
- **`key`** (*`str or None`*) – Additionally, the stream is grouped by the values of the given stats entry to differentiate between e.g. different events (for example key='starttime', key='onset')
- **`number_components`** (*`int, tuple of ints or None`*) – Only iterate through substreams with matching number of components.

`rf.util.`**`direct_geodetic`**(*latlon*, *azi*, *dist*)

Solve direct geodetic problem with geographiclib.

Parameters

- **`latlon`** (*`tuple`*) – coordinates of first point
- **`azi`** – azimuth of direction
- **`dist`** – distance in km

Returns coordinates (lat, lon) of second point on a WGS84 globe

`rf.util.`**`iter_event_data`**(*events*, *inventory*, *get_waveforms*, *phase='P'*, *request_window=None*, *pad=10*, *pbar=None*, ***kwargs*)

Return iterator yielding three component streams per station and event.

Parameters

- **`events`** – list of events or `Catalog` instance
- **`inventory`** – `Inventory` instance with station and channel information

- **get_waveforms** – Function returning the data. It has to take the arguments network, station, location, channel, starttime, endtime.
- **phase** – Considered phase, e.g. 'P', 'S', 'PP'
- **request_window** (*tuple (start, end)*) – requested time window around the onset of the phase
- **pad** (*float*) – add specified time in seconds to request window and trim afterwards again
- **pbar** – tqdm instance for displaying a progressbar

Returns three component streams with raw data

Example usage with progressbar:

```
from tqdm import tqdm
from rf.util import iter_event_data
with tqdm() as t:
    for stream3c in iter_event_data(*args, pbar=t):
        do_something(stream3c)
```

rf.util.**iter_event_metadata**(*events*, *inventory*, *pbar=None*)
Return iterator yielding metadata per station and event.

Parameters

- **events** – list of events or Catalog instance
- **inventory** – Inventory instance with station and channel information
- **pbar** – tqdm instance for displaying a progressbar

rf.util.**minimal_example_Srf**()
Return S receiver functions calculated from example data.

rf.util.**minimal_example_rf**()
Return receiver functions calculated from the data returned by read_rf().

5.7 batch Module

rf: receiver function calculation - batch command line utility

class rf.batch.**ConfigJSONDecoder**(*encoding=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)
Bases: json.decoder.JSONDecoder

Strip lines from comments.

decode(*s*)

exception rf.batch.**ParseError**
Bases: exceptions.Exception

rf.batch.**init_data**(*data*, *client_options=None*, *plugin=None*, *cache_waveforms=False*)
Return appropriate get_waveforms function.

See example configuration file for a description of the options.

rf.batch.**iter_event_processed_data**(*events*, *inventory*, *pin*, *format*, *yield_traces=False*, *pbar=None*)
Iterator yielding streams or traces which are read from disc.

`rf.batch.`**`load_func`**`(`*modulename*, *funcname*`)`
Load and return function from Python module.

`rf.batch.`**`run`**`(`*command*, *conf=None*, *tutorial=False*, ***kw*`)`
Create example configuration file and tutorial or load config.

After that call `run_commands`.

`rf.batch.`**`run_cli`**`(`*args=None*`)`
Command line interface of rf.

After parsing call `run`.

`rf.batch.`**`run_commands`**`(`*command*, *commands=()*, *events=None*, *inventory=None*, *objects=None*, *get_waveforms=None*, *data=None*, *plugin=None*, *cache_waveforms=None*, *phase=None*, *moveout_phase=None*, *path_in=None*, *path_out=None*, *format='Q'*, *newformat=None*, ***kw*`)`
Load files, apply commands and write result files.

`rf.batch.`**`tqdm`**`()`

`rf.batch.`**`write`**`(`*stream*, *root*, *format*, *type=None*`)`
Write stream to one or more files depending on format.

CHAPTER 6

Template Configuration File

```
### Configuration file for rf package in json format
# Comments are indicated with "#" and ignored while parsing

{

### Options for input and output ###

# Filename of events file (QuakeML format)
"events": "example_events.xml",

# Filename of inventory of stations (StationXML format)
"inventory": "example_inventory.xml",

# Data can be
#   1. a glob expression of files. All files are read at once. If you have
#      a huge dataset think about using one of the other two available
#      options.
#   2. one of the client modules supported by ObsPy (e.g "arclink", "fdsn")
#      for getting the data from a webservice.
#      Option "client_options" is available.
#   3. "plugin" for a heterogeneus dataset. You have the freedom to get your
#       data from different locations. In this case the option "plugin" is
#       availlable
"data": "example_data.mseed",

# Options for the webservices which are passed to Client.__init__.
# See the documentation of the clients in ObsPy for availlable options.
"client_options": {"user": "name@insitution.com"},

# Where to find the plugin in the form "module : func"
# 'module' has to be importable (located in current path or PYTHON_PATH
# environment variable).
# 'func' has to be the function which delivers the data, e.g. for a
# FSDN client:
#     # in module.py
#     from obspy.fsdn import Client
#     client = Client()
#     def func(**kwargs):
#         kwargs.pop('event')  # event kwarg is not needed by client
#         return client.get_waveforms(**kwargs)
# Kwargs passed to func are: network, station, location, channel,
# starttime, endtime and event
"plugin": "module : func",

# Directory for optional waveform caching from webservice or plugin.
# This works only if the waveforms are requested with the exact same
# parameters (start and end time) i.e. if options
# events and request_window are left untouched.
# This option needs the module joblib.
"cache_waveforms": null,

# File format for output of script (one of "Q", "SAC" or "H5")
#"format": "Q",



### Options for rf ###

"options": {  # See documentation of util.iter_event_data and rfstream.rfstats
    # Phase to use (e.g. "P", "PP", "S"), last letter will determine
    # if P- or S-receiver functions are calculated.
#     "phase": "P",
    # Data request window around onset in seconds
#     "request_window":  [-50, 150],
    # Events outside this distance range (epicentral degree) will be discarded
#     "dist_range": [30, 90],
    # Depth of piercing points in km
    "pp_depth": 50
},

"rf": {  # See RFStream.rf for options
    "filter":   {"type": "bandpass", "freqmin": 0.01, "freqmax": 2.0},
    # Trim window around P-onset
    "trim": [-30, 100]
    # Downsample stream to this frequency in Hz.
#     "downsample": 10,
    # Roate stream with this method.
    # rotate should be one of 'ZNE->LQT', 'NE->RT'
#     "rotate": "ZNE->LQT",
    # Deconvolve stream with this method.
    # deconvolve is one of 'time' or 'freq' for time or
    # frequency domain deconvolution.
#     "deconvolve": "time",
    # time domain deconvolution options
#     "spiking": 1.0,  # spiking factor for noise suppression
    # frequency domain deconvolution options
#     "water": 0.05,  # water level for noise suppression
#     "gauss": 2.0,  # low pass Gauss filter with corner frequency in Hz
    # window for source function relative to onset, (start, end, tapering)
#     "winsrc": [-10, 30, 5]
},


### Options for other routines ###

#"moveout": {},  # See RFStream.moveout

"plot": {"fillcolors": ["black", "gray"], "trim": [-5, 22]},  # See RFStream.plot_rf

# [start of profile in km, end of profile in km, number of bins],
# if specified, these values will be passed to np.linspace to generate a
# bin list for the boxes dictionary
"boxbins": [0, 10, 10],
"boxes": {"latlon0": [-21.0, -69.6], "azimuth": 90}  # See profile.get_profile_boxes
#"profile": {}  # See profile.profile
#"plot_profile": {}  # See RFStream.plot_profile
}
```

Python Module Index