InfraPy Documentation

Release 1.0

F.Dannemann Dugick, P.Blom, J.Webster

Jun 08, 2020

CONTENTS

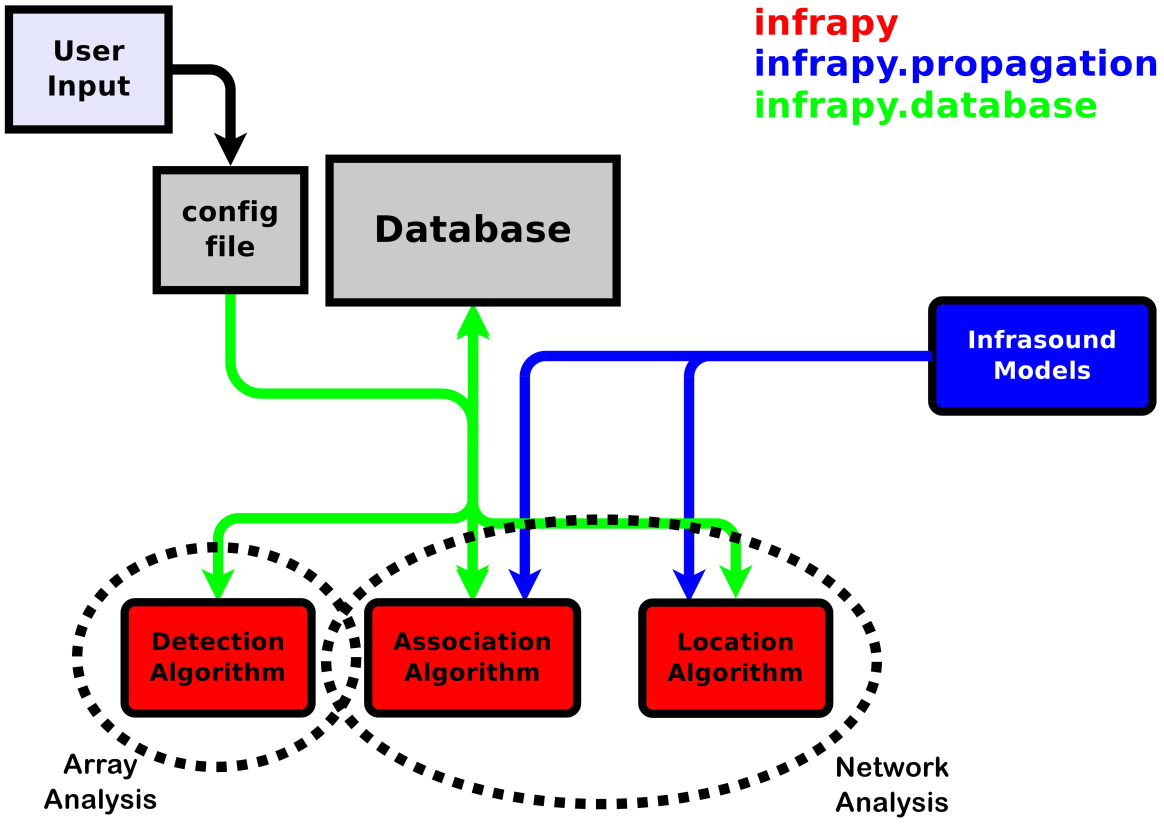
CHAPTER

ONE

CONTENTS

Welcome to Infrapy's documentation. Get started with *Installation* and then get an overview with the *Quickstart*. There is also a more detailed *Tutorial* that demonstrates the various processing capabilities of infrapy. The rest of the docs describe each component in detail, with a full reference in the *API* section.

This document is a work in progress and may be updated as development of Infrapy continues.



This part of the documentation, which is mostly prose, begins with some background information about Infrapy, then focuses on step-by-step instructions for data processing using Infrapy.

1.1 Overview

Infrapy is a tool for processing infrasound and seismic array data. Infrapy implements a database-centric approach for pipeline continuous near real-time analysis. The pipeline includes analysis at station and network levels (using beamforming and clustering techniques, respectively) for the detection, association and location of events. The pipeline relies on the interaction of the algorithms with a relational database structure to organize and store waveform data, the parameters for the analysis, and results of both levels of analysis. Our implementation can interact seamlessly with traditional (e.g.: Oracle) and serverless (e.g.: SQLite) relational databases.

1.2 Authorship

Infrapy was built upon previous similar (InfraMonitor) tools and developed by the LANL SeismoAcoustic Team.

Philip Blom pblom at lanl.gov

Jeremy Webster jwebster at lanl.gov

Fransiska Dannemann Dugick fransiska at lanl.gov

1.3 Installation

1.3.1 Operating Systems

Infrapy can currently be installed on machines running newer versions of Linux or Apple OSX. A Windows-compatible version is in development.

1.3.2 Anaconda

The installation of infrapy currently depends on Anaconda to resolve and download the correct python libraries. So if you don't currently have anaconda installed on your system, please do that first.

Anaconda can be downloaded from https://www.anaconda.com/distribution/. Either 3.x or 2.x will work since the numbers refer to the Python version of the default environment. Infrapy's installation will create a new environment and will install the version of Python that it needs into that environment.

1.3.3 Infrapy Installation

Once Anaconda is installed, you can install infrapy by navigating to the base directory of the infrapy package (there will be a file there named infrapy_env.yml), and run:

```
>> conda env create -f infrapy_env.yml
```

If this command executes correctly and finishes without errors, it should print out instructions on how to activate and deactivate the new environment:

To activate the environment, use:

```
>> conda activate infrapy_env
```

To deactivate an active environment, use

```
>> conda deactivate
```

1.3.4 Testing

Once the installation is complete, you can test some things by first activating the environment with:

```
>> conda activate infrapy_env
```

Then navigate to the /example directory located in the infrapy base directory, and run the test scripts via something like:

```
>> python test_beamforming.py
```

If infrapy was successfully installed, all of the test scripts should run and finish without any errors.

1.3.5 Running the InfraView GUI Application

Once installation is complete, and the new environment is activated, you can run the GUI with the command:

```
>> infraview
```

1.4 Quickstart

A series of scripts illustrating how to use infrapy subroutines as stand-alone modules are found in the /examples folder. The jupyter notebook documenting these steps is found in /tutorials/Quick Start.ipynb. The notebook can be run by installing jupyter notebook via conda.

```
>> conda install jupyter notebook
```

Beamforming:

1. Run Bartlett, Capon or Generalized Least Squares beamforming processes on an hour-long dataset from the BRP array in Utah

```
>> python test_beamforming.py
```

2. Visualize beamforming results in the sx/sy space

```
>> python test_slowness-grid.py
```

Detection:

1. Run detection on the series of beamforming results produced in the above step

```
>> python test_detection.py
```

Association

1. Associate a number of detections contained in a .dat file (/data/detection_set1.dat or /data/detection_set2.dat)

```
>> python test_assoc.py
```

Location

1. Test the Bayesian Infrasonic Source Localization (BISL) methodology using a set of provided detections (/data/detection_set1.dat or /data/detection_set2.dat). Location will be run twice, once assuming uniform atmospheric propagation and a second time applying provided atmospheric propagation priors for the Western US (see Blom et al., 2015 for further explanation)

```
>> python test_bisl.py
```

1.5 Interfacing with Pisces

Infrapy leverages pisces to connect with and process data in either local sqlite databases or oracle databases. More information about pisces can be found at https://jkmacc-lanl.github.io/pisces/.

1.5.1 Converting Data into Sqlite Databases

Data in miniseed or sac formats can be loaded into a sqlite database for pipeline processing using commands from pisces.

1. mseed to database (ms2db.py)

```
>> ms2db.py sqlite:///example.sqlite mslist.txt
```

2. sac to database (sac2db)

```
>> pisces sac2db sqlite:///example.sqlite *.sac
```

As infrapy is an array processing tool, after your sqlite database is created, you will need to update the REFSTA for each array using update_refsta.py

```
>> update_refsta.py sqlite:///example.sqlite <array name>
```

You can update the calibration for each array using update_calib.py

```
>> update_calib.py sqlite:///example.sqlite <array name> <calibration>
```

1.5.2 Connecting to a SQL Database

Infrapy employs two main methods for connecting to either Oracle or sqlite databases. Example files to facilitate these connections are found in tutorials/.

1.5.2.1 Defining Schema Specific Tables

Pipeline processing in infrapy utilizes information from CSS3.0 Site and Wfdisc tables. If your database schema differs from the CSS3.0 schema in any way, you can define the differences using a _global.py file. An example _global.py file is found in tutorial/ .

1.5.2.2 Connection within pipeline processing configuration file

The first three lines of your configuration file define the database you will connect to:

Example Configuration File for Sqlite Processing (Sqlite_Config.txt)

```
[database] # required
# url to database where you have the pointers to data and metadata
url = sqlite:///example.sqlite
# schema specific tables for your site and wfdisc files.  If you are processing in a␣
↪sqlite database, these variables will refer to schema specified in pisces. If you␣
↪are processing in an oracle database, these variables will refer to schema␣
↪specified in your global_.py file
site = pisces.tables.css3:Site
wfdisc = pisces.tables.css3:Wfdisc
```

Example Configuration File for Oracle DB Processing (Oracle_Config.txt)

```
[database] # required
url = oracle://<database name>:<port>
site = global_:Site
wfdisc = global_:Wfdisc_raw
```

1.5.2.3 Connection with a db.cfg file

Some modules in infrapy (db2sac) require a .cfg file to establish connection with a database. Examples are found in tutorial/ . More information can be found in the pisces documentation.

Example Configuration File for Oracle DB Processing (oracle_connection.cfg)

```
[database] # required
url = oracle://<db name>:<db port>
site = global_:Site
wfdisc = global_:Wfdisc_raw
origin = global_:Origin
```

Example Configuration File for Sqlite Processing (sqlite_connection.cfg)

```
[database] # required
url = sqlite:///example.sqlite
site = pisces.tables.css3:Site
wfdisc = pisces.tables.css3:Wfdisc
origin = pisces.tables.css3:Origin
```

1.6 Algorithms

- Analyst methods are modular so that results from other processing tools can be used in later analysis steps.
- Algorithms are written to be data agnostic so analysis can be performed regardless of data source once IO method is established

1.6.1 Station Level Processing

1.6.1.1 Array Processing

- Beamforming estimates parameters of coherent signals
- Capabilities include methods to characterize transients as well as persistent signals
- Transient signals are identified using standard Bartlett beaming
- Persistent signals can be investigated using Minimum Variance Distortionless Response (MVDR) or MUltiple Signal Classification (MUSIC) algorithms

1.6.1.2 The Adaptive F-Detector

- Adaptive Fisher statistics determine when to declare a detection

1.6.2 Network Level Processing

1.6.2.1 Association

- Events are identified using a pair-based Bayesian algorithm that defines the association between detection pairs from their joint-likelihood and identifies events via hierarchical clustering analysis
- Current implementation utilizes only spatial and temporal coincidence, but additional detection information can further improve event identification
- Evaluation using a synthetic data set shows some mixing of spatially similar events poorly resolved by network geometry and occasional inclusion of "noise" detections in event clustering

1.6.2.2 Bayesian Infrasonic Source Localization

- Event analysis using the Bayesian Infrasonic Source Localization (BISL) methodology to estimate both the spatial location of the event as well as the origin time with quantified uncertainty
- Preliminary analysis of the back projections can be used to define the spatial region of interest or it can be specified by the analysis
- Analysis identifies the maximum posteriori solution
- The marginalized spatial distribution is approximated as 2d-normal to define 95 and 99% confidence ellipse bounds
- The marginalized temporal distribution is analyzed to identify the exact 95 and 99% confidence bounds
- Likelihood definitions relating detection parameters to spatial and temporal source characteristics are shared between association and localization analysis for consistency

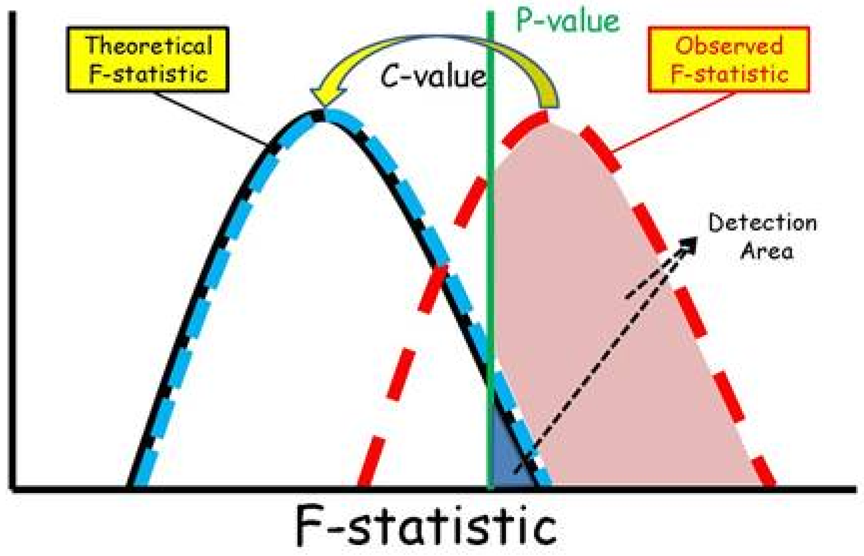
1.6.2.2.1 Array Processing

The use of infrasonic arrays, specifically for CTBT applications, is preferable due to the inherent reduction in signal-to-noise ratios (SNR) originating from the summation of four or more recordings at each array. The nature of a decision-rule based detector requires data that has been pre-processed using beamforming methods. Beamforming, a form of array processing, is the first step in the analysis of data from infrasonic arrays. Conventional beamforming methods (Bartlett, Capon) separate coherent and incoherent parts of a signal through the assumption of planar waves arriving at the array. A signal backazimuth and slowness can be estimated as signals are shifted to account for travel time differentials across array elements, bringing the signal into phase across as the noise deconstructively cancels out. In the classical, or Bartlett methodology , data records on each array element are time-shifted versions of the other with local noise,

See the following for more references on beamforming: Rost and Thomas 2002 Olson and Szuberla 2010 Costley 2013

1.6.2.2.2 The Adaptive F-Detector

The standard F-detector is based on decision rules for the F-statistic, which provides an estimate of the signal's beam power and is calculated as the power in the beam divided by the average over all channels of the power difference between the beam and the individual channels cite{Blandford:1982}. The AFD accounts for both correlated and uncorrelated noise through an increase in the value of the F-statistic required to declare a detection based upon background F-values being elevated from coherent or persistent noise sources. The figure below illustrates how the AFD remaps the F-statistic through the application of a C-value, which effective reduces the detection threshold (p-value) and decreases the number of noise-related detections.



See the following for more references on the Adaptive F-Detector:

Arrowsmith et al., 2008 Arrowsmith et al., 2009

1.6.2.2.3 Association

- Events are identified using a pair-based Bayesian algorithm that defines the association between detection pairs from their joint-likelihood and identifies events via hierarchical clustering analysis
- Current implementation utilizes only spatial and temporal coincidence, but additional detection information can further improve event identification
- Evaluation using a synthetic data set shows some mixing of spatially similar events poorly resolved by network geometry and occasional inclusion of "noise" detections in event clustering

See the following for more references on Association: Blom et al., 2020

1.6.2.2.4 Bayesian Infrasonic Source Localization

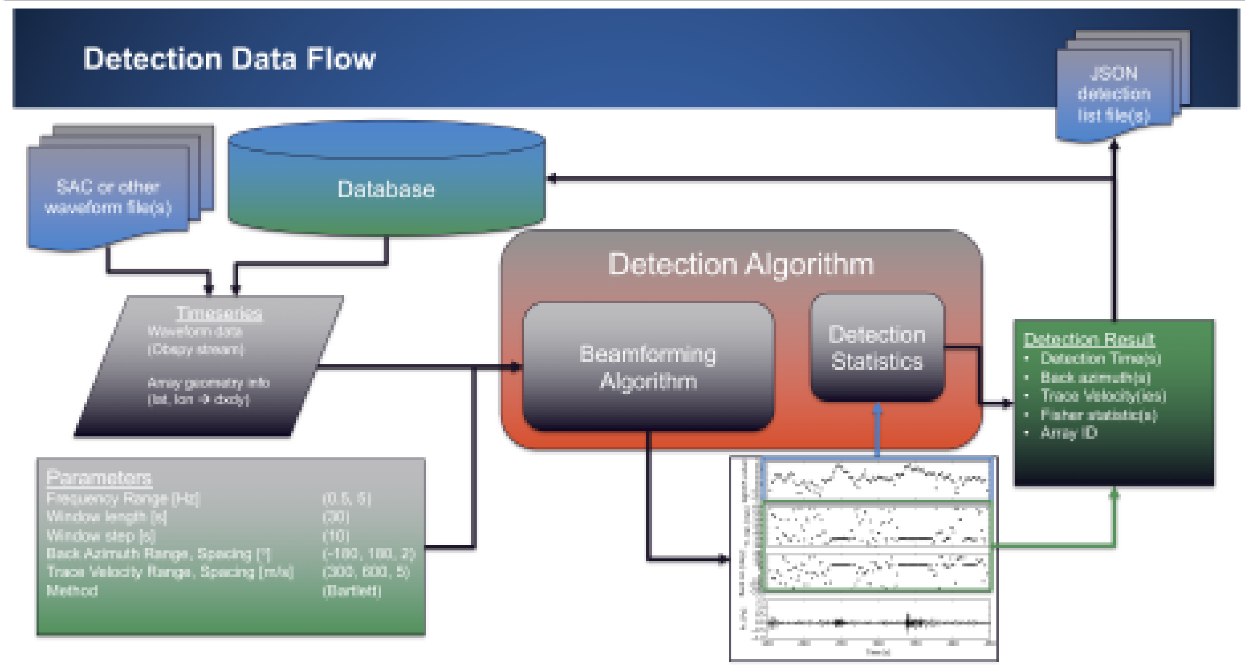
- Event analysis uses the Bayesian Infrasonic Source Localization (BISL) methodology to estimate both the spatial location of the event as well as the origin time with quantified uncertainty
- Preliminary analysis of the back projections can be used to define the spatial region of interest or it can be specified by the analysis
- Analysis identifies the maximum posteriori solution
- The marginalized spatial distribution is approximated as 2d-normal to define 95 and 99% confidence ellipse bounds
- The marginalized temporal distribution is analyzed to identify the exact 95 and 99% confidence bounds
- Likelihood definitions relating detection parameters to spatial and temporal source characteristics are shared between association and localization analysis for consistency

See the following for more references on BISL: Modrak et al., 2010 Marcillo et al., 2013 Blom et al., 2015

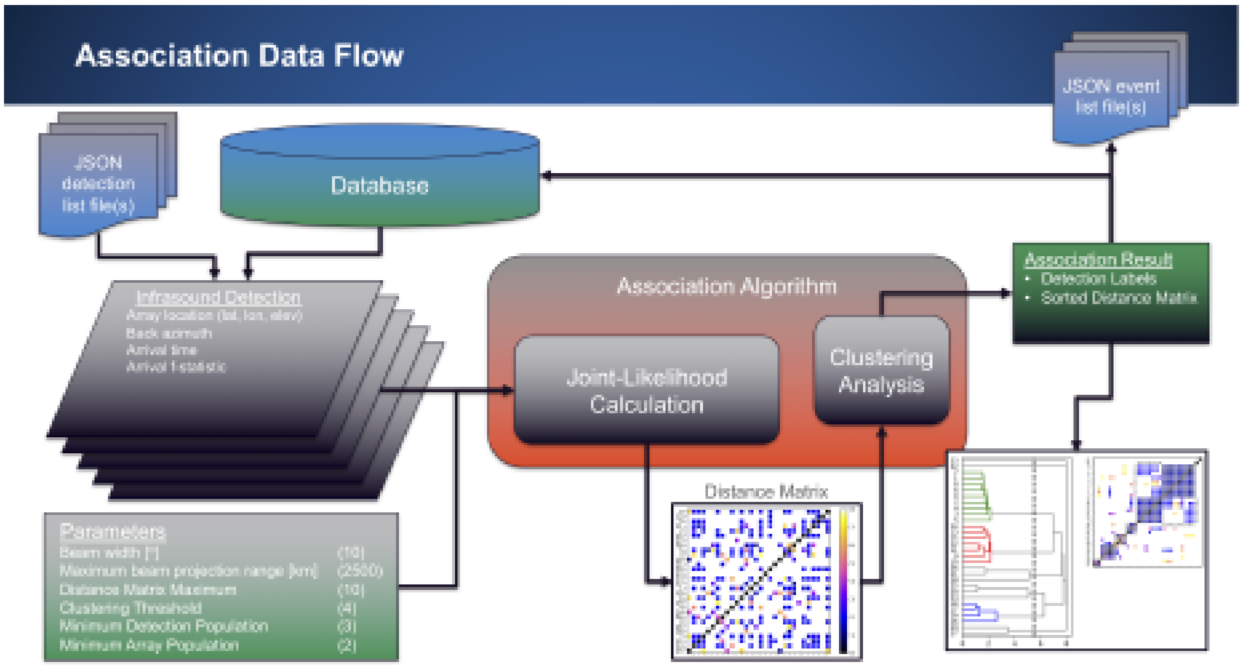
1.7 Data Processing Flow

Data can be processed using InfraPy in a variety of ways. See the images below for examples of data processing workflows.

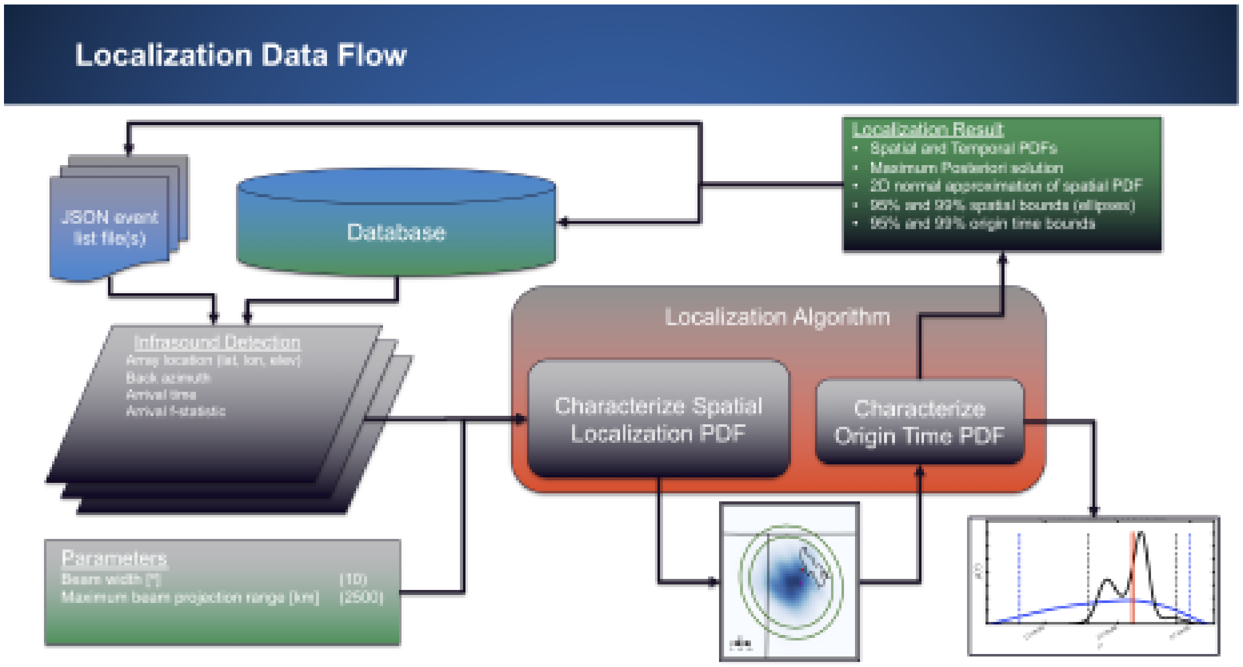
1.7.1 The Adaptive F-Detector



1.7.2 Association



1.7.3 Bayesian Infrasonic Source Localization


Localization Data Flow
JSON event list file(s)
Database
Localization Result
Spatial and Temporal PDFs
Maximum Posteriori solution
2D normal approximation of spatial PDF
95% and 99% spatial bounds (ellipses)
95% and 99% origin time bounds
Infrasound Detection
Array location (lat, lon, elev)
Back azimuth
Arrival time
Arrival f-statistic
Localization Algorithm
Characterize Spatial Localization PDF
Characterize Origin Time PDF
Parameters
Beam width [°] (10)
Maximum beam projection range [km] (2500)

1.7.3.1 Stand-Alone Processing

Modules within Infrapy can be run 'stand-alone' as package imports following the scripts found in the /examples folder

1.7.3.2 Running Pipeline Processing in Infrapy

The folder tutorials/cli contains all necessary data and configuration files to begin utilizing the pipeline processing methodologies in Infrapy.

Once installed, the steps to run pipeline processing in Infrapy are:

1. Either load local waveform data into a sqlite database or connect to a Oracle database following instructions in *Interfacing with Pisces*.
2. Create a configuration file. See *Configuration Files* for a detailed description of the parameters that need to be included. Two example configuration files, one for connecting to the provided sqlite file and one for connecting to an oracle database are provided.
3. Run the FK analysis for a specific array:

```
>> infrapy run_fk --config_file BRPConfig.txt
```

4. Run the FD analysis for a specific array that has already FK results, remember to locate the parameter id from your FK analysis (you can use the script read_pfk.py to find the correct id).

```
  >> infrapy run_fd --config_file BRPConfig.txt

4. Once you have run detection on 2+ array, run association processing:

```

(continues on next page)

(continued from previous page)

```
.. code-block:: python

    >> infrapy run_assoc --config_file BRPConfig.txt
```

1.7.3.2.1 Configuration Files

We run the different steps of the processing by running a script along with a configuration file. This applies to both station and network analysis levels. Configuration files set the parameters that are required to perform a specific analysis. The configuration file for the array processing (station level analysis) has the following structure:

An example configuration file is provided in tutorials/. Parameters for each field within the configuration file are outlined below.

```
# configuration file to run array (FK) processing and (FD) detection

[database] # required
url = sqlite:///example.sqlite
# database for processing
site = pisces.tables.css3:Site
wfdisc = pisces.tables.css3:Wfdisc
affiliation = pisces.tables.css3:Affiliation
# schemas for tables utilized in processing
# if processing in sqlite database, tables remain the same
# if processing in Oracle database, tables should point to your custom global_.py file

[GeneralParams]
year=2012
# year for processing
dayofyearini=100
# Julian Day to begin processing
dayofyearend=102
# Julian Day to stop processing
station=BRP
# REFSTA of station for processing
channel=EDF
# channel
name=example
# name of processing parameters
cpucnt=30
# number of cpu cores to use for processing
domain=time
# domain (time or frequency) to run FK and FD processing


[FKParams]
name=mid band fk test
# name for fk processing
freqmin=0.5
# minimum frequency
freqmax=5.0
# maximum frequency
beamwinlen=60
# beam window length
beamwinstep=30
# beam window step
```

(continues on next page)

(continued from previous page)

```
backazmin=-180.0
# minimum bz for processing
backazmax=180.0
# maximum bz for processing
backazstep=1.5
# bz step
trvelmin=300.0
# minimum trace velocity
trvelmax=600.0
# maximum trace velocity
trvelstep=2.5
# trace velocity step
beammethod=bartlett
# beam method
fkresults=fk_res_brp
# where fk processing results are stored
numsources = 1
func_fk = None

[FDetectParams]
back_az_lim=10
# limit of bz deviation between consecutive fk results
detwinlen=300.0
# window length for adaptive f detection
detthresh=0.99
# detection threshold
dsegmin=5
detmethod=fstat
tb_prod=4000
adaptivewlen=1200
#length of window for AFD
pthreshold=.01
#p-value for time domain detection
pfkid=0
# pkfid for FK results
corrthreshold=0.5
# threshold for correlation values
mineventlength
# minimum event length in seconds
fkresults=fk_res_brp
# fk results to run detection on
fdresults=fd_res_example_brp
# where detection results are saved



[AssocLocParams]
network=YJ
# network for association
pfdetectid=0
# detection ID from detection processing (all arrays for assoc must have same detect␣
↪ID)
pfkid=2
# beamforming ID (all arrays must have same FK ID)
beamwidth=10.0
rangemax=1000.0
clusterthresh=4.0
```

(continues on next page)

(continued from previous page)

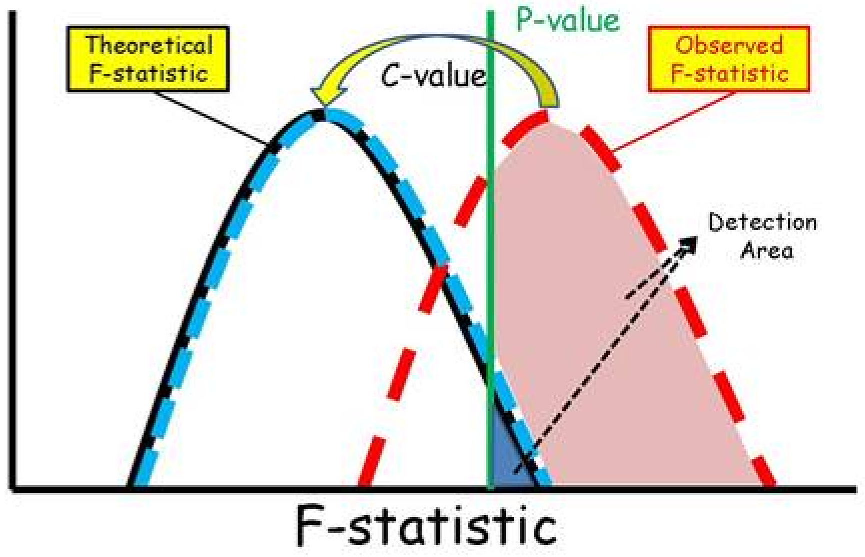
```
trimthresh=None
eventdetmin=3
# minimum # of detections to form event
eventarrmin=2
# minimum number of arrays for event
duration = 60
# duration (minutes) for association windows

fdtable_1=fd_res_example_brp
#fdtable_2=fd_res_fsu
#fdtable_3=fd_res_wmu
# tables where detection results are stored
resultstable = test_assoc
# table where association results will be stored
```

1.7.3.2.2 Infrapy FK Processing

```
>> infrapy run_fk --config_file BRPConfig.txt
```

1.7.3.2.3 Infrapy Detection (FD) Processing

Infrapy detects signals using an Adaptive F-Detector (AFD). The adaptive F-detector was developed by Arrowsmith et al., (2009) to account for both correlated and uncorrelated noise through modification of the conventional F-statistic. The detector accounts for temporal changes in noise by applying an adaptive window to update the detection distribution, which allows for the distinction between signal and correlated noise.



Configuration file

Detection is run using the same configuration file as FK processing. Detection requires input from FK processing results

```
>> infrapy run_fd --config_file BRPConfig.txt
```

1.7.3.2.4 Infrapy Assoc Processing

```
>> infrapy run_assoc --config_file BRPConfig.txt
```

1.7.3.2.5 Scripts

Scripts to manipulate Infrapy FK results

Use "any_command".py -h to get specific information to run the script

1. read_pfk.py to see the different set of configuration parameters used previously for FK analysis

-h –help show this help message and exit
-d SQ name of the database connection, e.g.: -d sqlite:///UT_tutorial.sqlite

```
>> read_pfk.py [-h] -d SQ
```

2. print_rfk.py to see fk results for a specific array and FK parameter ID

-h –help show this help message and exit
-d SQ name of the database connection, e.g.: -d sqlite:///UT_tutorial.sqlite
-a ARRAY array name, e.g.: -HWU4
-t FKRESULTS specific table with results, e.g.: -t fk_HWU
-s TS starttime plot, e.g.: -s /'2014-03-02T00:00:00/'
-e TE endtime plot, e.g.: -s /'2014-03-03T00:00:00/'
-F FVAL limit Fvalue, e.g.: -F 0
-o OUTTEXT print fk data, e.g.: -o res_FILE

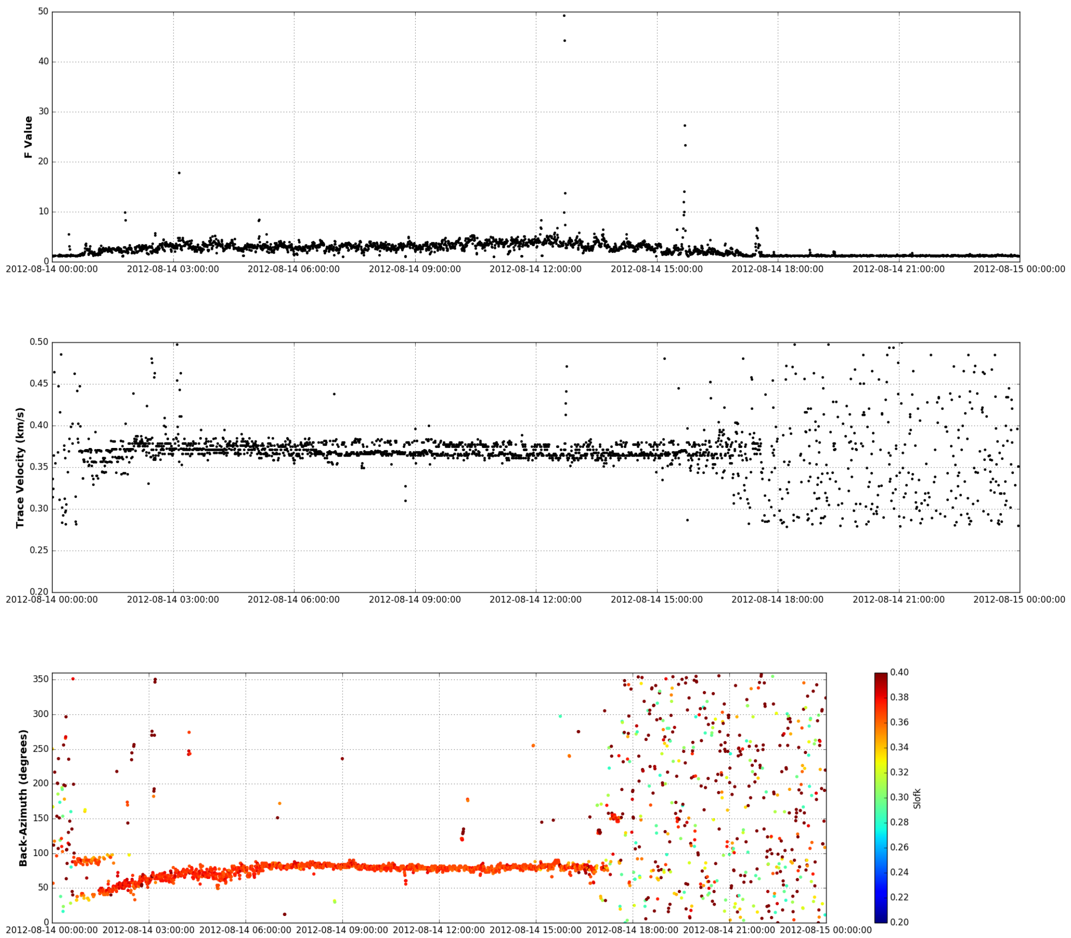
```
>> print_rfk.py [-h] -d SQ -a ARRAY -f PFK_ID -t FKRESULTS [-s TS] [-s TE] [-F FVAL] -
↪o res_FILE
```

3. plot1_rfk.py to plot FK results

-h –help show this help message and exit
-d SQ name of the database connection, e.g.: -d sqlite:///UT_tutorial.sqlite
-a ARRAY array name, e.g.: -a HWU4
-f PFK_ID FK parameter ID to be plot, e.g.: -f 3

-t FKRESULTS specific table with results, e.g.: -t fk_res_HWU
-w WAVEPLOT plot waveforms, e.g.: -w 0
-s TS starttime plot, e.g.: -s /'2014-03-02T00:00:00/'
-e TE endtime plot, e.g.: -s /'2014-03-03T00:00:00/'
-F FVAL limit Fvalue, e.g.: -F 0
-slo SLOFK limit slofk, e.g.: -slo 0
-bzmin BZMIN limit min bz, e.g.: -bzmin 0
-bzmax BZMAX limit max bz, e.g.: -bzmax 360

```
>>plot1_rfk.py [-h] -d SQ -a ARRAY -f PFK_ID [-t FKRESULTS] [-w WAVEPLOT] [-s TS] [-e
↪TE] [-F FVAL] [-slo SLOFK] [-bzmin BZMIN] [-bzmax BZMAX]
```



Scripts to manipulate Infrapy FD results

1. read_pfd.py to see the different set of configuration parameters used previously for detection analysis

-h, –help show this help message and exit
-d SQ name of the database connection, e.g.: -d sqlite:///mydb.sqlite

```
>> read_pfd.py [-h] -d SQ
```

2. read_rfd.py to see the available detection results

-h, –help show this help message and exit
-d SQ name of the database connection, e.g.: -db sqlite:///UT_tutorial.sqlite
-a ARRAY array name, e.g.: -a HWU4
-f PFK_ID, –pfkid PFK_ID FK parameter ID to be plot, e.g.: -f 0
-j PFDID, –pfdid PFDID fd parameter id, e.g.: -j 0
-t FDRESULTS specific table with results, e.g.: -t fd_res_HWU

```
>> read_rfd.py [-h] -d SQ -a ARRAY -f PFK_ID -j PFDID [-t FDRESULTS]
fdid: 1  pfdid: 0 pfkid: 0   timeini: 12-08-14 00:24:30   timeend: 12-08-14 00:26:00 ␣
↪ maxf0: 5.54282460217 0.60800443287
fdid: 2  pfdid: 0 pfkid: 0   timeini: 12-08-14 00:47:00   timeend: 12-08-14 00:53:30 ␣
↪ maxf0: 3.67253815208 0.46716764663
fdid: 3  pfdid: 0 pfkid: 0   timeini: 12-08-14 00:54:30   timeend: 12-08-14 00:56:30 ␣
↪ maxf0: 2.61098286001 0.372113714177
fdid: 4  pfdid: 0 pfkid: 0   timeini: 12-08-14 01:47:30   timeend: 12-08-14 01:49:00 ␣
↪ maxf0: 9.91099311406 0.749628285504
```

3. read_rfd_fast.py to write the available detection results in text file

-h, –help show this help message and exit
-d SQ name of the database connection, e.g.: -d sqlite:///UT_tutorial.sqlite
-a ARRAY array name, e.g.: -a I37NO
-f PFKID, –pfkid PFKID fk parameter id, e.g.: -f 0
-j PFDID, –pfdid PFDID fd parameter id, e.g.: -j 0
-t FDRESULTS specific table with results, e.g.: -t fd_I37
-o OUTTEXT fd parameter id, e.g.: -o res_FILE

```
>> read_rfd_fast.py [-h] -d SQ -a ARRAY -f PFKID -j PFDID [-t FDRESULTS] [-o OUTTEXT]
```

1.8 Tutorial

A series of jupyter notebooks illustrating how to use infrapy subroutines are found in the /tutorial folder. You can run these by navigating to that folder and running:

```
>> jupyter notebook
```

A browser will launch with a webpage showing a directory listing, click on the InfraPyTutorial.ipynb link to open the main window, which will provide links to the various tutorials.

1.9 Schema

The purpose of this document is to define the schema used for the operation of the infrasound analysis tool, infrapy. The tables described by this document extend the CSS3.0 or KB core schema to include information required for the operation of infrapy. This document is divided into three sections, the first being this introduction. Section two defines eight new, infrasonic data processing-specific database tables. Both internal (ORACLE) and external formats for the attributes are defined, along with a short description of each attribute. Section three of the document shows the relationships between the different tables by using entity-relationship diagrams.

This schema is a work in progress and may be updated as development of infrapy continues.

1.9.1 Table Descriptions

This section describes the logical structure of each table used in the Infrapy software package. The name of the table is first, followed by a description of the purpose and use of the table. Below the description is a listing of the columns, in the order which they are defined in the tables. The storage column gives the actual ORACLE datatype for the column in question. The external format and character positions columns are provided for the convenience of database users who wish to transfer data between the ORACLE database tables and flat files.

1.9.1.1 Conventions

The following conventions are used, following Carr et al (2002):

Element	Appearance	Example
Database table	Bold	**arrival**
Database columns	Italic	*sta*
Database table and column when written in the dot notation	Bold.italic	**arrival**.*sta*
Value of a key or component of a key	Courier font	arid

1.9.1.2 Table Definitions: Infrapy-Specific Tables

Table descriptions can be found in the API section of the documentation.

1.10 API

1.10.1 Association

1.10.1.1 HJL

`infrapy.association.hjl.`**`build_distance_matrix`**(*det_list*, *bm_width=10.0*, *rng_max=10005.97260168349*, *rad_min=100.0*, *rad_max=1000.0*, *resol=180*, *pool=None*, *progress=False*)

Computes the joint-likelihood for all pairs of detections to define the distance matrix

Computes the joint-likelihood value for each unique pair of detections in a provided list and uses a negative-log-joint-likelihood to convert to non-Euclidean distance for clustering analysis

det_list [`list` of `InfrasoundDetection`] List of detections (see infrapy.propagation.likelihoods)

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

resol [int] Number of radial and azimuthal points used in the polar projection of the likelihood PDFs

pool [pathos.multiprocessing.ProcessingPool] Multiprocessing pool for accelerating calculations

Returns: dist_matrix : 2darray

Distance matrix describing joint-likelihood separations for all pairs

`infrapy.association.hjl.`**`cluster`**(*distance_matrix*, *threshold*, *linkage_method='weighted'*, *show_result=False*, *file_id=None*, *den_label_size=9*, *mat_label_size=7*, *trim_indices=[]*)

Computes the clustering solution for a distance matrix and threshold

Computes the linkages for a distance matrix using SciPy's hierarchical (agglomerative) clustering methods and returns the event labels for the original detection list

distance_matrix [2darray] Two dimensional numpy array representing distance matrix

threshold [float] Threshold value defining linkage cutoff

linkage_method [str] Linkage method used by scipy.cluster.hierarchy.linkage

show_results [boolean] Boolean to plot dendrogram and sorted distance matrix to screeen

file_id [str] Prefix for output file is dendrogram and sorted distance matrix figure is saved

den_label_size [float] Font size off the dendrogram labels in the figure

mat_label_size [float] Font size of the distance matrix label in the figure

trim_indices [`list` of `int` pairs] Locations of linkages cut by trimming algorithm (only used in figure)

Returns: links : scipy.cluster.hierarchy.linkage

Output links from SciPy clustering analysis

labels [`list` of `int`] Labels of cluster memberships for each detection

distance_matrix_sorted [2darray] The distance matrix sorted to put clusters together `infrapy.association.hjl.`**`compute_assoc_pair`**(*det1*, *det2*, *bm_width=10.0*, *rng_max=10005.97260168349*, *rad_min=100.0*, *rad_max=1000.0*, *resol=180*, *prog_step=0*)

Computes the joint-likelihiood for a pair of detections

Projects finite width beams from each of the detecting arrays and looks for intersections of the primary (center) and secondary (edge) lines to define the integration region for computation of the joint-likelihiood between the detection pair

det1 [InfrasoundDetection] First detection (see infrapy.propagation.likelihoods)

det2 [InfrasoundDetection] Second detection (see infrapy.propagation.likelihoods)

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

resol [int] Number of radial and azimuthal points used in the polar projection of the likelihood PDFs

prog_step [int] Used to increment progress bar

Returns: jntlklhd : float

The joint-likelihood value for the pair of detections

`infrapy.association.hjl.`**`compute_assoc_pair_wrapper`**(*args*)

`infrapy.association.hjl.`**`run`**(*det_list*, *threshold*, *dist_max=10.0*, *bm_width=10.0*, *rng_max=10005.97260168349*, *rad_min=100.0*, *rad_max=1000.0*, *resol=180*, *show_result=None*, *file_id=None*, *linkage_method='weighted'*, *trimming_thresh=None*, *trim_thresh_scalar=1.0*, *pool=None*)

Run the Hierarchical Joint-Likelihood (HJL) association analysis

Runs the clustering analysis on a list of detecctions and returns the membership labels and sorted distance matrix for event identification

det_list [`list` of `InfrasoundDetection`] List of detections (see infrapy.propagation.likelihoods)

threshold [float] Threshold value defining linkage cutoff

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

resol [int] Number of radial and azimuthal points used in the polar projection of the likelihood PDFs

show_results [boolean] Boolean to plot dendrogram and sorted distance matrix to screeen

file_id [str] Prefix for output file is dendrogram and sorted distance matrix figure is saved

linkage_method [str] Linkage method used by scipy.cluster.hierarchy.linkage

trim_thresh_scalar [float] Scalar modifying the threshold value for linkage cutoff in the trimmed result

pool [pathos.multiprocessing.ProcessingPool] Multiprocessing pool for accelerating calculations

Returns: labels : `list` of `int`

Labels of cluster memberships for each detection

distance_matrix_sorted [2darray] The distance matrix sorted to put clusters together

`infrapy.association.hjl.`**`set_region`**(*det1*, *det2*, *bm_width=10.0*, *rng_max=10005.97260168349*, *rad_min=100.0*, *rad_max=1000.0*)

Defines the integration region for computation of the joint-likelihood for a pair of detections

Projects finite width beams from each of the detecting arrays and looks for intersections of the primary (center) and secondary (edge) lines to define the integration region for computation of the joint-likelihiood between the detection pair

det1 [InfrasoundDetection] First detection (see infrapy.propagation.likelihoods)

det2 [InfrasoundDetection] Second detection (see infrapy.propagation.likelihoods)

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

Returns: Successs : boolean

True if region was defined, False if not

Center [float] Center of the integration region as latitude, longitude pair [degrees]

Radius [float] Radius of the integration region [km]

`infrapy.association.hjl.`**`summarize_clusters`**(*labels*, *distance_matrix*, *population_min=3*, *show_result=False*)

Prints summary of cluster association solution to screen

Searches through labels to identify clusters with sufficient membership to declare events and summarizes cluster quality

labels [`list` of `int`] Labels of cluster memberships for each detection

distance_matrix [2darray] Two dimensional numpy array representing distance matrix

population_min [int] Minimum number of detections in a cluster to declare an event

`infrapy.association.hjl.`**`trim_clusters`**(*labels*, *distance_matrix*, *population_min=3*, *ratio_thresh=3.0*)

Trims linkages in poorly shaped clusters

Identifies poorly shaped clusters by the ratio of their mean inter-element distances and radius (maximum inter-element distance) and returns indices of the linkages that should be cut to improve clustering

labels [`list` of `int`] Labels of cluster memberships for each detection

distance_matrix [2darray] Two dimensional numpy array representing distance matrix

population_min [int] Minimum number of detections in a cluster to declare an event

ratio_thresh [float] Threshold for radius / mean inter-element distance to require trimming

Returns: trim_indices : `list` of `int`

Indices of linkages causing poor cluster shape `infrapy.association.hjl.`**`view_distance_matrix`**(*distance_matrix*, *file_id=None*, *ordering=None*)

View distance matrix used for clustering analysis

dist_matrix [2darray] Distance matrix describing joint-likelihood separations for all pairs

1.10.2 Characterization

1.10.2.1 SPYE

`infrapy.characterization.spye.`**`blastwave`**(*t*, *p0*, *t0*, *alpha=0.0*)

Acoustic blastwave that can be used as a source for surface explosions out to several scale kilometers.

Note: alpha = 0 produces the Friedlander blastwave model.

t [float] Time [s]

p0 [float] Peak overpressure

t0 [float] Time scale [s]

alpha [float] Shaping parameter (positive)

p [float] Overpressure at time t

`infrapy.characterization.spye.`**`blastwave_spectrum`**(*f*, *p0*, *t0*, *alpha=0.0*)

Fourier transform amplitude for the acoustic blastwave in sasm.acoustic.blastwave().

Note: alpha = 0 produces the Friedlander blastwave model.

Note: the peak of the spectrum occurs at f_0 =

rac{1}{2 pi t_0} rac{1}{sqrt{lpha + 1}

and t0 corresponding to a given peak frequency is t_0 =

rac{1}{2 pi f_0} rac{1}{sqrt{lpha + 1}

f [float] Frequency [Hz]

p0 [float] Peak overpressure

t0 [float] Time scale [s]

alpha [float] Shaping parameter (positive)

P [float] Spectral value at frequency f

`infrapy.characterization.spye.`**`extract_spectra`**(*det_list*, *st_list*, *win_buffer=0.25*, *ns_opt='pre'*)

Extract spectra for a list of detections using a list of obspy streams with a defined window buffer and noise window option

det_list [`list` of `InfrasoundDetection`] Iterable of detections with defined start and end times

st_list [`list` of `obspy.Stream`] Iterable of Obspy streams containing the array channels for each detection

win_buffer [float]

Scaling factor defining the window buffer (a 20 second detection with a buffer of 0.25 adds 5 seconds to the start and end of the window)

ns_opt [string]

Option defining how the noise window is defined. Options include "pre", "post", and "beam" to use the preceding, following, or beam residual

smn_spec [ndarray] Spectral amplitude of the signal-minus-noise

`infrapy.characterization.spye.`**`kg_op`**(*W*, *r*, *p_amb=101.325*, *T_amb=288.15*, *type='chemical'*)

Kinney & Graham scaling law peak overpressure model

W [float] Explosive yield of the source [kg eq. TNT]

r [float] Propagation distance [km]

p_amb [float] Ambient atmospheric pressure [kPa]

T_amb [float] Ambient atmospheric temperature [deg K]

type [string] Type of explosion modeled, options are "chemical" or "nuclear"

p0 [float] Peak overpressure [Pa]

`infrapy.characterization.spye.`**`kg_ppd`**(*W*, *r*, *p_amb=101.325*, *T_amb=288.15*, *type='chemical'*)

Kinney & Graham scaling law positive phase duration model

W [float] Explosive yield of the source [kg eq. TNT]

r [float] Propagation distance [km]

p_amb [float] Ambient atmospheric pressure [kPa]

T_amb [float] Ambient atmospheric temperature [deg K]

type [string] Type of explosion modeled, options are "chemical" or "nuclear"

t0 [float] Positive phase duration [s]

`infrapy.characterization.spye.`**`run`**(*det_list*, *smn_spec*, *src_loc*, *freq_band*, *tloss_models*, *resol=150*, *yld_rng=array([10., 10000.])*, *ref_src_rng=1.0*, *grnd_brst=True*)

Run Spectral Yield Estimation (SpYE) methods to estimate explosive yield

det_list [`list` of `InfrasoundDetection`] Iterable of detections with defined start and end times

smn_spec [ndarray] Spectral amplitude of the signal-minus-noise

freq_band [iterable] List or tuple with minimum and maximum frequency (e.g., [f_min, f_max])

tloss_models [list of frequencies and infrapy.propagation.TLossModel instances]

Propagation transmission loss model list with reference frequencies (see test/test_yield.py for construction example)

resol [int] Resolution used in analysis

yld_rng [iterable] List or tuple with minimum and maximum yields (e.g., [yld_min, yld_max])

ref_src_rng [float] Standoff distance used to define source model distance

grnd_brst [boolean]

Boolean declaring whether the source is a ground burst (interaction with source doubles the effective yield for a ground burst vs. air burst)

yld_vals: Values of explosive yield for the PDF

yld_pdf [] Probability of the observed source having a given yield

conf_bnds [ndarray] Limits for the 68% and 95% confidence bounds on yield

1.10.3 Database Processing

1.10.3.1 Database Processing Taskbase

Updated Fall 2019 @fkdd @omarcillo

class `infrapy.database.taskbase.fdet.`**FDet**(*conf_file=[]*, *ARRAY_NAME=None*)

Bases: `infrapy.database.taskbase.base.Base`

classdocs

data_processing()

Constructor

data_retrieving(*dayofyear*)

database_connecting()

Constructor

distfit(*x*, *F*, *dofnum*, *dofden*, *c*)

Returns the correlation coefficient between the theoretical F-distribution and the distribution of the input F-statistics, scaled by a c-value

Inputs: - x is the set of F-statistics for binning - F are the measured F-statistics - dofnum is the number of degrees of freedom on the numerator - dofden is the number of degrees of freedom on the denominator - c is the c value

Outputs: - corr is the correlation coefficient

event_detection()

event_detection2()

getdet(*time*, *fval_corr*, *pf*, *p*, *corr*, *corr_thres*, *min_no_samples*, *fval=None*, *c=None*)

Obtains the start and end points for significant detections given a p-value. Inputs: - pf is the array of p-values as a function of time - p is the threshold p-value Outputs: - start_pt is the list of indices corresponding to the starts of detections - stop_pt is the list of indices corresponding to the ends of detections

Created on Oct 31, 2014 Updated Jan 2020

@author: omarcillo, fkdd

class `infrapy.database.taskbase.assoc.`**AssocInfraPy_LANL**(*conf_file=[]*)

Bases: `infrapy.database.taskbase.base.Base`

classdocs

algorithm = 'Blom and Euler'

data_processing()

Constructor

data_processingASSOC(*t_start*, *t_end*, *src_win*, *max_prop_tm*)

data_retrievingS(*win_start*, *win_end*)

database_connecting()
Constructor

Created on Oct 31, 2014

@author: omarcillo

class infrapy.database.taskbase.loc.**LocInfraPy**(*conf_file=[]*)
Bases: infrapy.database.taskbase.base.Base

classdocs

algorithm = 'Blom/BISL III'

data_processing()
Constructor

data_processingLOC()

data_retrieving(*dayofyear*)

database_connecting()
Constructor

meanlocations()

infrapy.database.taskbase.loc.**run_bislSC_wrapper**(*args*)
wrapper for multicore and quality control

1.10.3.2 Database Schema

class infrapy.database.schema.**ASSOC_params**(**args*, ***kwargs*)
Bases: pisces.schema.util.Base

algorithm = Column(None, String(length=15), table=None)

beamwidth = Column(None, Float(precision=24), table=None)

clusterthresh = Column(None, Float(precision=24), table=None)

duration = Column(None, Float(precision=24), table=None)

classmethod from_string(*line*, *default_on_error=None*)
Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
```

(continues on next page)

(continued from previous page)

```
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

minarraypop = Column(None, Float(precision=24), table=None)

mindetpop = Column(None, Float(precision=24), table=None)

name = Column(None, Float(precision=53), table=None)

passocid = Column(None, Integer(), table=None)

rangemax = Column(None, Float(precision=24), table=None)

trimthresh = Column(None, String(length=15), table=None)

trimthreshscalar = Column(None, Float(precision=24), table=None)

class infrapy.database.schema.**ASSOC_results**(**args*, ***kwargs*)

Bases: pisces.schema.util.Base

associd = Column(None, Integer(), table=None)

eventid = Column(None, Integer(), table=None)

fdid = Column(None, Integer(), table=None)

fdtable = Column(None, String(length=15), table=None)

classmethod from_string(*line*, *default_on_error=None*)

Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

net = Column(None, String(length=8), table=None)

passocid = Column(None, Integer(), table=None)

qassoc = Column(None, Float(precision=53), table=None)

qdetcluster = Column(None, Float(precision=53), table=None)

sta = Column(None, String(length=6), table=None)

timeend = Column(None, Float(precision=53), table=None)

timeini = Column(None, Float(precision=53), table=None)

class `infrapy.database.schema.`**`EDetectParams`**(**args*, ***kwargs*)

Bases: `pisces.schema.util.Base`

`algorithm = Column(None, String(length=15), table=None)`

`filterh = Column(None, Float(), table=None)`

`filterl = Column(None, Float(), table=None)`

`filtertype = Column(None, String(length=15), table=None)`

`classmethod from_string`(*line*, *default_on_error=None*)

Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

`name = Column(None, Float(precision=53), table=None)`

`overlapwlen = Column(None, Float(), table=None)`

`pedetectid = Column(None, Integer(), table=None)`

`wlenlong = Column(None, Float(), table=None)`

`wlenshort = Column(None, Float(), table=None)`

class `infrapy.database.schema.`**`fd_params`**(**args*, ***kwargs*)

Bases: `pisces.schema.util.Base`

`additional1 = Column(None, Float(precision=24), table=None)`

`additional2 = Column(None, Float(precision=24), table=None)`

`algorithm = Column(None, String(length=15), table=None)`

`backazlim = Column(None, Integer(), table=None)`

`cthr = Column(None, Float(precision=53), table=None)`

`detwinlen = Column(None, Float(precision=53), table=None)`

`dsegmin = Column(None, Float(precision=53), table=None)`

`classmethod from_string`(*line*, *default_on_error=None*)

Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

minlen = Column(None, Float(precision=53), table=None)

name = Column(None, Float(precision=53), table=None)

numsources = Column(None, Integer(), table=None)

pfdid = Column(None, Integer(), table=None)

pthreshold = Column(None, Float(precision=53), table=None)

tb_prod = Column(None, Integer(), table=None)

class infrapy.database.schema.**fd_results**(**args*, ***kwargs*)
Bases: pisces.schema.util.Base

c = Column(None, Float(precision=24), table=None)

ebz = Column(None, Float(precision=53), table=None)

ec = Column(None, Float(precision=24), table=None)

etimeend = Column(None, Float(precision=53), table=None)

etimeini = Column(None, Float(precision=53), table=None)

etrvel = Column(None, Float(precision=24), table=None)

fdid = Column(None, Integer(), table=None)

fktablename = Column(None, String(length=15), table=None)

classmethod from_string(*line*, *default_on_error=None*)
Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
```

(continues on next page)

(continued from previous page)

```
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

maxfc = Column(None, Float(precision=24), table=None)

maxfo = Column(None, Float(precision=24), table=None)

pfdid = Column(None, Integer(), table=None)

pfkid = Column(None, Integer(), table=None)

sourcenum = Column(None, Integer(), table=None)

sta = Column(None, String(length=6), table=None)

tend = Column(None, Float(precision=53), table=None)

timeend = Column(None, Float(precision=53), table=None)

timeini = Column(None, Float(precision=53), table=None)

tonset = Column(None, Float(precision=53), table=None)

trvel = Column(None, Float(precision=24), table=None)

class `infrapy.database.schema.`**fk_params**(**args*, ***kwargs*)
Bases: `pisces.schema.util.Base`

additional1 = Column(None, Float(precision=53), table=None)

additional2 = Column(None, Float(precision=53), table=None)

algorithm = Column(None, String(length=15), table=None)

backazmax = Column(None, Float(precision=53), table=None)

backazmin = Column(None, Float(precision=53), table=None)

backazstep = Column(None, Float(precision=53), table=None)

beamwinlen = Column(None, Float(precision=53), table=None)

beamwinstep = Column(None, Integer(), table=None)

domain = Column(None, String(length=15), table=None)

filter = Column(None, String(length=15), table=None)

freqmax = Column(None, Float(precision=53), table=None)

freqmin = Column(None, Float(precision=53), table=None)

classmethod from_string(*line*, *default_on_error=None*)
Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

maxslowness = Column(None, Float(precision=53), table=None)

minslowness = Column(None, Float(precision=53), table=None)

name = Column(None, String(length=15), table=None)

numsources = Column(None, Integer(), table=None)

pfkid = Column(None, Integer(), table=None)

stepslowness = Column(None, Float(precision=53), table=None)

trvelmax = Column(None, Float(precision=53), table=None)

trvelmin = Column(None, Float(precision=53), table=None)

trvelstep = Column(None, Float(precision=53), table=None)

class `infrapy.database.schema.`**fk_results**(**args*, ***kwargs*)
Bases: `pisces.schema.util.Base`

additional1 = Column(None, Float(precision=24), table=None)

additional2 = Column(None, Float(precision=24), table=None)

bz = Column(None, Float(precision=53), table=None)

chan = Column(None, String(length=8), table=None)

coher = Column(None, Float(precision=24), table=None)

ebz = Column(None, Float(precision=53), table=None)

eslofk = Column(None, Float(precision=24), table=None)

esx = Column(None, Float(precision=24), table=None)

esy = Column(None, Float(precision=24), table=None)

etrvel = Column(None, Float(precision=24), table=None)

fkid = Column(None, Integer(), table=None)

classmethod from_string(*line*, *default_on_error=None*)
Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

fval = Column(None, Float(precision=24), table=None)

nchan = Column(None, Integer(), table=None)

pfkid = Column(None, Integer(), table=None)

rmsval = Column(None, Float(precision=24), table=None)

slofk = Column(None, Float(precision=24), table=None)

sourcenum = Column(None, Integer(), table=None)

sta = Column(None, String(length=6), table=None)

sx = Column(None, Float(precision=24), table=None)

sy = Column(None, Float(precision=24), table=None)

timeend = Column(None, Float(precision=53), table=None)

timeini = Column(None, Float(precision=53), table=None)

trvel = Column(None, Float(precision=24), table=None)

xcorrvalmax = Column(None, Float(precision=24), table=None)

xcorrvalmean = Column(None, Float(precision=24), table=None)

xcorrvalmin = Column(None, Float(precision=24), table=None)

class infrapy.database.schema.**loc_params**(**args*, ***kwargs*)
Bases: pisces.schema.util.Base

algorithm = Column(None, String(length=15), table=None)

classmethod from_string(*line*, *default_on_error=None*)
Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

maxlat = Column(None, Float(precision=53), table=None)

maxlon = Column(None, Float(precision=53), table=None)

minlat = Column(None, Float(precision=53), table=None)

minlon = Column(None, Float(precision=53), table=None)

name = Column(None, Float(precision=53), table=None)

plocid = Column(None, Integer(), table=None)

priors = Column(None, Integer(), table=None)

class infrapy.database.schema.**loc_results**(**args*, ***kwargs*)

Bases: pisces.schema.util.Base

additional1 = Column(None, Float(precision=24), table=None)

additional2 = Column(None, Float(precision=24), table=None)

additional3 = Column(None, Float(precision=24), table=None)

additional4 = Column(None, Float(precision=24), table=None)

eventid = Column(None, Integer(), table=None)

classmethod from_string(*line*, *default_on_error=None*)

Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

latlonorigcovar = Column(None, Float(precision=53), table=None)

latorigmap = Column(None, Float(precision=53), table=None)

latorigmean = Column(None, Float(precision=53), table=None)

latorigvar = Column(None, Float(precision=53), table=None)

locid = Column(None, Integer(), table=None)

lonorigmap = Column(None, Float(precision=53), table=None)

lonorigmean = Column(None, Float(precision=53), table=None)

lonorigvar = Column(None, Float(precision=53), table=None)

net = Column(None, String(length=8), table=None)

numstations = Column(None, Integer(), table=None)

par_a = Column(None, Float(precision=24), table=None)

par_b = Column(None, Float(precision=24), table=None)

par_theta = Column(None, Float(precision=24), table=None)

plocid = Column(None, Integer(), table=None)

timeend = Column(None, Float(precision=53), table=None)

timeini = Column(None, Float(precision=53), table=None)

timeorigmap = Column(None, Float(precision=53), table=None)

timeorigmax = Column(None, Float(precision=53), table=None)

timeorigmean = Column(None, Float(precision=53), table=None)

timeorigmin = Column(None, Float(precision=53), table=None)

timeorigvar = Column(None, Float(precision=53), table=None)

class infrapy.database.schema.**noise_params**(**args*, ***kwargs*)

Bases: pisces.schema.util.Base

algorithm = Column(None, String(length=15), table=None)

filterh = Column(None, Float(precision=24), table=None)

filterl = Column(None, Float(precision=24), table=None)

filtertype = Column(None, String(length=15), table=None)

classmethod from_string(*line*, *default_on_error=None*)

Construct a mapped table instance from correctly formatted flat file line.

Works with fixed-length fields, separated by a single whitespace.

line: str Flat file line (best to remove newline, but maybe not a problem).

default_on_error: list, optional Supply a list of column names that return default values if they produce an error during parsing (e.g. lddate).

ValueError: Can't properly parse the line.

default_on_error is useful for malformed fields, but it will also mask other problems with line parsing. It's better to pre-process tables to match the table specifications or catch exceptions and isolate these lines.

```
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line)
or
>>> with open('TA.site','r') as ffsite:
        for line in ffsite:
            isite = Site.from_string(line, default_on_error=['lddate'])
```

n_octave = Column(None, Integer(), table=None)

name = Column(None, Float(precision=53), table=None)

overlapwlen = Column(None, Integer(), table=None)

pnoiseid = Column(None, Integer(), table=None)

wlen = Column(None, Float(precision=53), table=None)

1.10.4 Beamforming/Detection

1.10.4.1 Beamforming

`infrapy.detection.beamforming.`**AIC**(*eigenV*, *n*, *m*)

`infrapy.detection.beamforming.`**MDL**(*eigenV*, *n*, *m*)

`infrapy.detection.beamforming.`**bfstat**(*beam*)

Returns the F-statistic using the formalism of Blandford (1974) for a particular window (FK sliding window)

Inputs: - beam is the time-aligned waveform in a particular window (FK sliding window)

Outputs: - F is the Blandford F-statistic

`infrapy.detection.beamforming.`**bfstat2**(*beam*)

Returns the F-statistic using the formalism of Blandford (1974) for a particular window (FK sliding window)

Inputs: - beam is the time-aligned waveform in a particular window (FK sliding window)

Outputs: - F is the Blandford F-statistic

`infrapy.detection.beamforming.`**bfstatT**(*beam*)

Returns the F-statistic using the formalism of Blandford (1974) for a particular window (FK sliding window)

Inputs: - beam is the time-aligned waveform in a particular window (FK sliding window)

Outputs: - F is the Blandford F-statistic

`infrapy.detection.beamforming.`**corrp**(*beam*)

Computes the maximum average cross correlation from all triplets of elements in an array for a particular window (FK sliding window)

Inputs: - beam is the time-aligned waveform in a particular window (FK sliding-window)

Outputs: - C is the minimum, maximum, mean average cross-correlation

`infrapy.detection.beamforming.`**corrpT**(*beam*)

Computes the maximum average cross correlation from all triplets of elements in an array for a particular window (FK sliding window)

Inputs: - beam is the time-aligned waveform in a particular window (FK sliding-window)

Outputs: - C is the minimum, maximum, mean average cross-correlation

`infrapy.detection.beamforming.`**detect_peaks**(*image*, *size=3*, *num_peaks=None*)

Takes an image and detect the peaks using the local maximum filter. Returns a boolean mask of the peaks (i.e. 1 when the pixel's value is the neighborhood maximum, 0 otherwise)

`infrapy.detection.beamforming.`**dist_az**(*X1*, *X2*)

Returns the angular distance and direction between two locations

`infrapy.detection.beamforming.`**fkPROC**(*method*, *stream*, *sps*, *slow*, *mult_vectors*, *fN*, *x*, *y*, *timeSTAMP*, *func*, *freqN*, *aux*, *num_sources=None*) `infrapy.detection.beamforming.`**`fkfromOStream`**(*St*, *wlen*, *overlap*, *freqmin*, *freqmax*, *slow=None*)

`infrapy.detection.beamforming.`**`getXY_array`**(*stream*)
Returns the site coordinates for a specific array in the format required by fk

`infrapy.detection.beamforming.`**`procPEAKS`**(*peaks*, *slow*)

`infrapy.detection.beamforming.`**`svdAV_wv`**(*stream*, *sps*, *slow*, *mult_vectors*, *fN*, *x*, *y*, *timeSTAMP*, *func*, *freqN*, *number_div=None*)

`infrapy.detection.beamforming.`**`tapering`**(*st_aux*)

`infrapy.detection.beamforming.`**`tdelay`**(*data*, *sps*, *bazimuth*, *slowness*, *x*, *y*)
Returns the time-delay-and-sum beam of data for a particular window (FK sliding window)

Inputs: - data is the non-time-aligned waveform in a particular window (FK sliding window), this may be already in real values - samprate is the sampling rate - x is the array of x coordinates - y is the array of y coordinates - azimuth is the azimuth for calculating the beam - slowness is the slowness for calculating the beam

Outputs: - beam is the time-aligned waveform in a particular window (FK sliding window)

`infrapy.detection.beamforming.`**`tdelayT`**(*data*, *sps*, *bazimuth*, *slowness*, *x*, *y*, *slowx*, *slowy*)
Returns the time-delay-and-sum beam of data for a particular window (FK sliding window)

Inputs: - data is the non-time-aligned waveform in a particular window (FK sliding window), this may be already in real values - samprate is the sampling rate - x is the array of x coordinates - y is the array of y coordinates - azimuth is the azimuth for calculating the beam - slowness is the slowness for calculating the beam

Outputs: - beam is the time-aligned waveform in a particular window (FK sliding window)

`infrapy.detection.beamforming.`**`xcorr_beam`**(*dat_st*)

1.10.4.2 Beamforming_new

infrapy.detection.beamforming_new.py

Methods for reading in time series data, analyzing it, and identifying infrasonic signals using various beamforming methods.

Author Philip Blom (pblom@lanl.gov)

`infrapy.detection.beamforming_new.`**`build_slowness`**(*back_azs*, *trc_vels*)
Compute the slowness values for a polar grid

Computes the slowness grid usingg a polar grid defined by a series of back azimuth values and trave velocity values. Returns a grid specified such that grid[n] is the x and y component of the nth slowness vector.

back_azs [1darray] Back azimuth values for slowness grid, K_1 values

trc_vels [1darray] Trace velocity values for slowness grid K_2 values

grid [2darray] (K_1 x K_2) by 2 array of slowness vectors

`infrapy.detection.beamforming_new.`**`calc_det_thresh`**(*fstat_vals*, *det_thresh*, *TB_prod*, *channel_cnt*)

`infrapy.detection.beamforming_new.`**`compute_beam_power`**(*data*, *steering*, *method='bartlett'*, *ns_covar_inv=None*, *signal_cnt=1*)
Compute the beampower for a specific frequency

Cmoputes the beampower at a single frequency using either the FFT'd data, X(f), for Bartlett or GLS analysis or the covariance matrix, S(f), for Capon and MUSIC.

Generalized Least Square (GLS) analysis requires a noise covariance for the background which must be M x M where M is the length of X(f).

MUltiple SIgnal Classification (MUSIC) analysis requires knowledge of the number of coherent signals in the data specified as signal_cnt.

data [ndarray] Vector of length M, X_m(f_n), for "bartlett" and "gls" or matrix of dimension M x M, S(f_n), for covariance based methods

steering [2darray] Matrix representing K steering vectors each of length K

method [str] Beamforming method to be applied to the data (must match for of data)

ns_covar_inv [2darray] Noise covariance used in "gls" beamforming method

signal_cnt [int] Number of signals assumed in MUSIC algorithm

beam_power [1darray] Beam power for each steering vector (length K)

`infrapy.detection.beamforming_new.`**`compute_beam_power_wrapper`**(*args*)

`infrapy.detection.beamforming_new.`**`compute_delays`**(*dxdy*, *param_grid*, *param_opt='planar'*, *sph_vel=340.0*, *sph_src_ht=0.0*)

Compute the delays for a planewave

Computes the time delays for each pair in param_grid given the array geometry in dxdy. For planar parameterization, grid specifies s_x and s_y of the slowness. For spherical parameterization, it specifies the x,y location of the source and requires specification of the velocity of the wavefront.

For the slowness grid, use the build_slowness function to convert back azimuth and trace velocity values into a grid. Use np.meshgrid and flatten to produce a grid for the spherical wavefront source grid.

dxdy [2darray] M x 2 matrix describing the array geometry

param_grid [2darray] K x 2 matrix of parameterization vectors containing either the slowness components (for 'planar') or the source location (for 'spherical')

delays [2darray] K x M of time delays across the array for each slowness

`infrapy.detection.beamforming_new.`**`detect_signals`**(*times*, *beam_results*, *win_len*, *TB_prod*, *channel_cnt*, *det_thresh=0.99*, *min_seq=5*, *back_az_lim=15*, *fixed_thresh=None*)

Identify detections with beamforming results

Identify detection in the beamforming results using either Kernel Density Estimate (KDE) fits to the f-statistic distribution or the adaptive F-detector methods developed by Arrowsmith.

times [1darray] Times of beamforming results as numpy datetime64's

beam_results [2darray] Beamforming results consisting of back azimuth, trace velocity, and f-value at each time step. This is a 2D array with dimensions (len(times), 3), where the first column has back azimuth values, the second has trace velocity values, and the third has f-statistic values

win_len [float] Window length to define the adaptive fstat threshold

TB_prod [int] Time-bandwidth product needed to compute the Fisher statistic

channel_cnt [int] Number of channels on the array; needed to compute the Fisher statistic

det_thresh [float] Threshold for declaring a detection

min_seq [int] Threshold for the number of sequential above-threshold values to declare a detection

back_az_lim [float] Threshold below which the maximum separation of back azimuths must be in order to declare a detection

fixed_thresh [float]

A fixed detection threshold for fstat values (overrides adaptive threshold calculation)

dets [list] List of identified detections including detection time, relative start and end times of the detection, back azimuth, trace velocity, and f-stat.

`infrapy.detection.beamforming_new.`**`extract_signal`**(*X*, *f*, *slowness*, *dxdy*)

Extract the signal along the beam for a given slowness vector

Extract the "best beam" signal from the array data for a given slowness pair and array geometry. Returns both the extracted signal and the residual on each trace of the array

Note: following Laslo's work, the frequency domain Fisher ratio can be computed as: F[nf] = abs(sig_est)**2 / np.mean(np.abs(residual), axis=1)**2 * (X.shape[1] - 1)

f [1darray] Frequencies

X [2darray] FFT of data in analysis window, x(t) –> X(f)

slowness [2darray] Slowness components (either back azimuth and trace velocity or s_x and s_y depending on slowness option)

dxdy [2darray] Array geometry

sig_estimate [1darray] Extracted frequency domain signal along the beam

residual [2darray] Residual across the array once beamed signal is extracted

`infrapy.detection.beamforming_new.`**`fft_array_data`**(*x*, *t*, *window=None*, *sub_window_len=None*, *sub_window_overlap=0.5*, *fft_window='hanning'*, *normalize_windowing=False*)

Compute the Fourier transform of the array data to perform analysis

Compute the Fourier transform of the array data within an analysis window defined by window = [t1, t2] and potentially using subwindows to obtain a full rank covariance matrix for beamforming analyses requiring such data. Multiple FFT window options are available and a normalization option scales to account for the amplitude loss at the window edges.

x [2darray] M x N matrix of array data, x[m][n] = x_m(t_n)

t [1darray] Vector of N sampled points in time, t[n] = t_n

window [float] Start and end time of the window relative to times in t, [t_1, t_2]

sub_window_len [float] Duration of the subwindow in seconds

sub_window_overlap [float] Fraction of subwindow to overlap (limited range of 0.0 to 0.9)

fft_window [str] Fourier windowing method

normalize_windowing [boolean] Boolean to apply normalization of window scaling

X [2darray] M x N_f matrix of the FFT'd data, X[m][n] = X_m(f_n)

S [3darray] M x M x N_f cube of the covariance matrices, S[m1][m2][n] = mean(X_{m1}(f_n) conj(X_{m2}(f_n)))

f [1darray] Vector of N_f frequencies for the FFT'd data, f[n] = f_n

`infrapy.detection.beamforming_new.`**`find_peaks`**(*beam_power*, *slowness_vals1*, *slowness_vals2*, *signal_cnt=1*, *freq_weights=None*)

Identify the peak(s) in the beampower defined over a slowness grid

Finds the peaks of a distribution using a frequency averaged beamforming result over a defined slowness grid.

beam_power [2darray] Beam power for each steering vector at each frequency in the band (dimension K x N_f)

slowness_vals1 [1darray] Slowness values along first axis (polar or Cartesian grid)

slowness_vals2 [1darray] Slowness values along second axis (polar or Cartesian grid)

signal_cnt [int] Number of signals to identify in the slowness grid

freq_weights [string or 1darray] Weights or method to use in frequency averaging of the beam power

peaks [ndarray] signal_cnt x 3 array of the peaks identified containing slowness value 1 (back azimuth), slowness value 2 (trace velocity), and beam value

`infrapy.detection.beamforming_new.`**`project_ABA`**

Project matrix of K vectors, a_k, onto Hermitian matrix B

Projects each of K vectors, a_k, in matrix, A, of dimension K x M onto a Hermitian matrix, B, of dimension M x M producing a vector of scalars, c, of length K

A [2darray] K x M matrix representing a set of K vectors, a_k, each of length M

B [2darray] M x M Hermitian matrix

c [1darray] Vector c where each scalar c_k = a_k^dagger B a_k

`infrapy.detection.beamforming_new.`**`project_ABc`**

Project matrix of K vectors, a_k, through Hermitian matrix B and onto vector c

Projects each of K vectors, a_k, in matrix, A, of dimension K x M through a Hermitian matrix, B, of dimension M x M and onto a vector, c, producing a vector of scalars, d, of length K

A [2darray] K x M matrix representing a set of K vectors, a_k, each of length M

B [2darray] M x M Hermitian matrix

c [1darray] Vector of length M

d [1darray] Vector d where each scalar d_k = a_k^dagger B c

`infrapy.detection.beamforming_new.`**`project_Ab`**

Project matrix of K vectors, a_k, onto a vector b

Projects a vector, b, of length M onto a set of K vectors, a_k, each of length M producing a vector, c, of length K

A [2darray] K x M matrix representing a set of K vectors, a_k, each of length M

b [1darray] Vector of length M to project onto each a_k

c [1darray] Vector c where each scalar c_k = a_k^dagger b

`infrapy.detection.beamforming_new.`**`project_beam`**(*beam_power*, *back_az_vals*, *trc_vel_vals*, *freq_weights=None*, *method='max'*)

Project polar slowness grid onto only azimuth

Projects the polar slowness grid onto back azimuth and trace velocity in order to more easily view each. The method can either use the maximum value to project or average to approximate the marginal distribution

beam_power [2darray] Beam power for each steering vector at each frequency in the band (K x N_f)

back_az_vals [1darray] Back azimuth values defining polar slowness grid

trc_vel_vals [1darray] Trace velocity values defining polar slowness grid

freq_weights [1darray] Weights to use in frequency averaging of the beam power

method [str] Determines whether mean or maximum along trace velocity axis is used to define the projections

back_az_proj [1darray] Projection of the beam power onto the back azimuth axis

trc_vel_proj [1darray] Projection of the beam power onto the trace velocity axis

`infrapy.detection.beamforming_new.`**`pure_state_filter`**(*S*)

Compute the pure state filter applied to a Hermitian matrix, S(f)

Computes the pure state filter for a matrix. Here, the covariance matrix is utilized to measure the average coeherence across the entire array at a given frequency.

Pure state filter value are useful for weighting a multi-frequency beam average.

S [3darray] Covariance matrix of data in analysis window for all frequencies, x(t) –> S(f) = mean(X(f) X^dagger(f))

pure_state [1darray] Pure state filter value at each frequency

`infrapy.detection.beamforming_new.`**`run`**(*X*, *S*, *f*, *dxdy*, *delays*, *freq_band*, *method='bartlett'*, *ns_covar_inv=None*, *signal_cnt=1*, *normalize_beam=True*, *pool=None*)

Run beamforming analysis over frequencies of interest

Computes the beam at multiple frequencies within a specified band given data in X(f) and S(f) and frequencies f as produced by the fft_array_data function.

Normalization of the beam returns coherence in the case of Bartlett and a normalized version of the Capon beam but does not alter the output of the MUSIC algorithm as its result is a mathematical projection onto a noise subspace.

A multiprocessing pool can be used to accelerate calculation of different frequencies in parallel.

X [2darray] M x N_f matrix of the FFT'd data, X[m][n] = X_m(f_n)

S [3darray] M x M x N_f cube of the covariance matrices, S[m1][m2][n] = mean(X_{m1}(f_n) conj(X_{m2}(f_n)))

f [1darray] Frequencies

delays [1darray] Set of delays for the parameterization (length K)

freq_band [iterable] List or tuple with minimum and maximum frequency (e.g., [f_min, f_max])

method [str] Beamforming method to be applied to the data (must match form of data)

signal_cnt [int] Number of signals assumed in MUSIC algorithm

ns_covar_inv [2darray] Noise covariance used in “gls” beamforming method

normalize_beam [boolean] Option to normalize the beam and return coherence (value between 0 and 1)

pool [multiprocessing pool] Multiprocessing pool for accelerating caluclation (maps over frequency)

param_opt [string] Option for the solution parameterization: ‘planar’ or ‘spherical’

sph_vel [float] Velocity of the wavefront in the ‘spherical’ param_opt method

bmpwr [2darray] Beam power for each steering vector at each frequency in the band (dimension K x N_f)

`infrapy.detection.beamforming_new.`**`stream_to_array_data`**(*stream*, *latlon=None*, *t_start=None*, *t_end=None*)

Extract time series from ObsPy stream on common time samples and define the array geometry

Extracts the time series from individual traces of an Obspy stream and identifies a common set of time samples where all are defined. Interpolates the individual traces into a single numpy array (x) for which x[m] = x_m(t). The geometry of the array is also extracted to enable beamforming analysis.

stream [ObsPy stream] Obspy stream containing traces for all array elements

latlon [2darray] (M x 2) 2darray containing the latitudes and longitudes of the array elements if they aren’t in the stream

x [2darray] M x N matrix of array data, x[m][n] = x_m(t_n)

t [1darray] Vector of N sampled points in time, t[n] = t_n

t_ref [datetime64] Datetime corresponding to t[0]

dxdy [2darray] M x 2 matrix of slowness vectors

1.10.5 Location

1.10.5.1 BISL

`infrapy.location.bisl.`**`calc_conf_ellipse`**(*means*, *st_devs*, *conf_lvl*, *pnts=100*)

Compute the confidence ellipse around a latitude longitude point

Computes the points on an ellipse centered at a latitude, longitude point with standard deviations (E/W, N/S, and covariance) defiend in kilometers for a given confidence level.

means [float] Latitude and longitude of the center

st_devs [float] East/West and North/South standard deviations, and covariance for the ellipse

conf_lvl [float] Confidence level in percentage (e.g., 95.0 = 95%)

pnts [int] Number of points to return around the ellipse contour

latlon [float] Latitutde and longitude points of the ellipse

`infrapy.location.bisl.`**`find_confidence`**(*func*, *lims*, *conf_lvl*)

Computes the bounds for a function given a confidence level

Identifies the points for which int_{x_1}^{x_2}{f(x) dx} includes a given fraction of the overall integral such that f(x_1) = f(x_2)

func [function of single float parameter] Function to be analyzed

lims [float] Limits for the integration of the norm and bounds for possible limits

bnds [float] The values of x_1 and x_2 for the confidence bound

conf [float] Actual confidence value obtained

thresh [float] Value of the function at x_1 and x_2

`infrapy.location.bisl.`**`run`**(*det_list, path_geo_model=None, custom_region=None, resol=180, bm_width=10.0, rng_max=10005.97260168349, rad_min=100.0, angle=[-180, 180], rad_max=1000.0, MaP_mthd='grid'*)

Run analysis of the posterior pdf for BISL

Compute the marginal disribution…

det_list [iterable of InfrasoundDetection instances] Detections attributed to the event

path_geo_model [Propagation-based, stochastic path geometry model] Optional path geometry model if available

custom_region [float] Latitude, Longitude, and radius of a specific region to conduct analysis (overrides use of set_region function)

resol [int] Number of radial and azimuthal points used in the polar projection of the spatial PDFs

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

angle [float] Minimum and maximum angles for polar projection of the spatial PDFs

MaP_method [string] Method to use for searching for the maximum a posteriori solution (“grid” or “random”)

result [dictionary]

Dictionary containing all localization results: ‘lat_mean’: Mean latitude for the marginalized spatial distribution ‘lon_mean’ : Mean longitude for the marginalized spatial distribution ‘EW_stdev’: Standard deviation of the marginalized spatial distribution in the east/west direction in km ‘NS_stdev’: Standard deviation of the marginalized spatial distribution in the north/south direction in km ‘covar’: Relative covariance, sigma_{xy}^2 / (sigma_x sigma_y) ‘t_mean’: Mean marginalized temporal distribution ‘t_stdev’: Standard deviation of the marginalized temporal distribution ‘t_min’ : 95% confidence bound lower limit for the marginalized temporal distribution ‘t_max’ : 95% confidence bound upper limit for the marginalized temporal distribution ‘lat_MaP’: Maximum a Posteriori latitude ‘lon_MaP’: Maximum a Posteriori longitude ‘t_MaP’: Maximum a Posteriori origin time ‘MaP_val’ : Maximum a Posteriori value

`infrapy.location.bisl.`**`set_region`**(*det_list, bm_width=10.0, rng_max=10005.97260168349, rad_min=100.0, rad_max=1000.0*)

Defines the integration region for computation of the BISL probability distribution

Projects finite width beams from each of the detecting arrays and looks for intersections of the primary (center) and secondary (edge) lines to define the integration region for computation of the localization distribution

det_list [iterable of InfrasoundDetection instances] Detections attributed to the event

bm_width [float] Width of the projected beam [degrees]

rng_max [float] Maximmum range for beam projection [km]

rad_min [float] Minimum radius of the integration region [km]

rad_max [float] Maximum radius of the integration region [km]

Center [float] Center of the integration region as latitude, longitude pair [degrees]

Radius [float] Radius of the integration region [km]

`infrapy.location.bisl.`**`summarize`**(*result*, *confidence_level=95*)

Outputs results of BISL analysis

Prints all results to screen in a readable format

bisl_result [dictionary] Dictionary output of the BISL analysis methods

1.10.6 Propagation

1.10.6.1 Infrasound

class `infrapy.propagation.infrasound.`**`PathGeometryModel`**

Bases: `object`

`az_bin_cnt = 8`

`az_bin_wdth = 60.0`

`bnc_max = 10`

`bnd_overlap = 0.075`

`build`(*results_file*, *model_file*, *show_fits=False*, *file_id=None*, *verbose_output=False*, *rng_width=40.0*, *rng_spacing=10.0*, *data_format='new'*, *abs_lim=-100.0*, *trn_ht_min=2.0*)

`default_az_dev_var = 4.0`

`display`(*file_id=None*, *hold_fig=None*)

`eval_az_dev_mn`(*rng*, *az*)

`eval_az_dev_vr`(*rng*, *az*)

`eval_rcel_gmm`(*rng*, *rcel*, *az*)

`load`(*model_file*, *smooth=None*)

`min_az_dev_var = 0.9`

`rcel_vrs_min = 0.05`

`rng_max = 1000.0`

`strat_therm_bnd = 3.846153846153846`

`tropo_strat_bnd = 3.2258064516129035`

class `infrapy.propagation.infrasound.`**`TLossModel`**

Bases: `object`

`az_bin_cnt = 8`

`az_bin_wdth = 60.0`

`build`(*results_file*, *model_file*, *show_fits=False*, *file_id=None*, *pool=None*)

`display`(*title='Transmission Loss Statistics'*, *file_id=None*, *hold_fig=None*)

`eval`(*rng*, *tloss*, *az*)

load(*model_file*)

infrapy.propagation.infrasound.**canonical_rcel**(*rcel*)

infrapy.propagation.infrasound.**canonical_tloss**(*rng*, *tloss*)

infrapy.propagation.infrasound.**find_azimuth_bin**(*az*, *bin_cnt=8*)

1.10.6.2 Likelihoods

exception infrapy.propagation.likelihoods.**BadArrayDimensionError**

Bases: *infrapy.propagation.likelihoods.CustomError*

Raised when someone tries to set the array dimension to something that isnt an integer

exception infrapy.propagation.likelihoods.**BadLatitudeError**

Bases: *infrapy.propagation.likelihoods.CustomError*

Rasied when the latitude is out of range

exception infrapy.propagation.likelihoods.**BadLongitudeError**

Bases: *infrapy.propagation.likelihoods.CustomError*

Raised when the longitude is out of range

exception infrapy.propagation.likelihoods.**CustomError**

Bases: Exception

class infrapy.propagation.likelihoods.**Infrapy_Encoder**(**, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class infrapy.propagation.likelihoods.**InfrasoundDetection**(*lat_loc=None*, *lon_loc=None*, *time=None*, *azimuth=None*, *f_stat=None*, *array_d=None*)

Bases: object

property array_dim
(int) Number of elements in the detecting array

az_pdf(*lat*, *lon*, *path_geo_model=None*)

property back_azimuth
(float) Back azimuth of the detected signal

calc_kappa_etc()

property elevation
Elevation of the detecting array

property end
(float) End time in seconds relative to the peak F utc time of the detection

property event_id
Optional id of the event

fillFromDict(*dict*)

property frequency_range
(tuple) Frequency range of the measurement

generateDict()

get_array_dim()

get_back_azimuth()

get_ele()

get_end()

get_event_id()

get_freq_range()

get_lat()

get_lon()

get_method()

get_name()

get_note()

get_peakF_UTCtime(*type='numpy'*)

get_peakF_value()

get_start()

get_trace_velocity()

property latitude
Latitude of the array

property longitude
Longitude of the detecting array

property method
(string) Beamforming method used to generate the detection

property name
Optional name of the detection

property note
Optional note regarding the detection

pdf (*lat*, *lon*, *t*, *path_geo_model=None*)

property peakF_UTCtime
time of the peak F value, in UTC

property peakF_value
Peak F values

rng_pdf (*lat*, *lon*, *t*, *path_geo_model=None*)

set_array_dim (*ad*)

set_back_azimuth (*ba*)

set_ele (*ele*)

set_end (*end*)

set_event_id (*evt*)

set_freq_range (*frange*)

set_lat (*lat*)

set_lon (*lon*)

set_method (*method*)

set_name (*n*)

set_note (*note*)

set_peakF_UTCtime (*pf_UTCtime*)

set_peakF_value (*pfv*)

set_start (*start*)

set_trace_velocity (*tv*)

src_spec_pdf (*lat*, *lon*, *freqs*, *src_spec*, *smn_spec*, *tloss_models*)
Defines the probability of detection being produced by a given source

Computes the probability of a given source spectral amplitude, P(f), producing this detection given its signal-minus-noise spectrum and a transmission loss model

lat [float] Latitude of the hypothetical source

lon [float] Longitue of the hypothetical source

freqs [1darray] Frequencies at which to evaluate the source spectrum

src_spec [1darray] Values at which to evaluate the source spectrum

smn_spec [2darray] Numpy array containing frequencies and signal-minus-noise spectral amplitudes of the detection

tloss_models [frequencies and TLossModel instances] Transmission loss models to use and frequencies at which they are computed

freq_grid [1darray] Frequencies at which the pdf is evaluated

src_spec_grid [1darray] Spectral amplitudes at which the pdf is evaluated

pdf [2darray] Grid of probability for source spectrum at frequencies and source spectrum amplitudes specified by input

property start
(float) Start time in seconds relative to the peak F utc time of the detection

property trace_velocity
Trace velocity of the detection

class infrapy.propagation.likelihoods.**SeismicDetection**(*lat*, *lon*, *time*, *phase='p'*, *slowness=None*)
Bases: object

pdf(*lat*, *lon*, *t*)

sigma()

trvl_tm(*rng*)

infrapy.propagation.likelihoods.**db2dets**(*file_name*)

infrapy.propagation.likelihoods.**detection_list_to_json**(*filename*, *detections*)

infrapy.propagation.likelihoods.**file2dets**(*file_name*)

infrapy.propagation.likelihoods.**joint_pdf**(*lat*, *lon*, *t*, *det_list*, *path_geo_model=None*, *prog_step=0*)

infrapy.propagation.likelihoods.**joint_pdf_wrapper**(*args*)

infrapy.propagation.likelihoods.**json_to_detection_list**(*filename*)

infrapy.propagation.likelihoods.**marginal_spatial_pdf**(*lat*, *lon*, *det_list*, *path_geo_model=None*, *prog_step=0*, *resol=100*)

infrapy.propagation.likelihoods.**marginal_spatial_pdf_wrapper**(*args*)

1.10.6.3 Seismic

infrapy.propagation.seismic.**ak135_p_tr_time**(*rng*)

infrapy.propagation.seismic.**ak135_s_tr_time**(*rng*)

1.10.7 Utils

1.10.7.1 infrapy.utils.cart2pol module

infrapy.utils.cart2pol.**cart2pol**(*x*, *y*)
Transform Cartesian to polar coordinates

Inputs: x is the x coordinate y is the y coordinate

Outputs: th is the angle r is the range

1.10.7.2 infrapy.utils.confidence module

infrapy.utils.confidence.**find_confidence**(*func*, *lims*, *conf_aim*)

1.10.7.3 infrapy.utils.db2db module

1.10.7.4 infrapy.utils.db2sac module

1.10.7.5 infrapy.utils.files2db module

class infrapy.utils.files2db.**CoreTable**(*name*, *prototype*, *table*)

Bases: tuple

property name
Alias for field number 0

property prototype
Alias for field number 1

property table
Alias for field number 2

infrapy.utils.files2db.**apply_plugins**(*plugins*, ***rows*)

infrapy.utils.files2db.**dicts2rows**(*dicts*, *classes*)

infrapy.utils.files2db.**expand_glob**(*option*, *opt_str*, *value*, *parser*)
Returns an iglob iterator for file iteration. Good for large file lists.

infrapy.utils.files2db.**get_files**(*options*)
Return a sequence of WFDISC file names from either a list of file names (trivial) or a text file list (presumable because there are too many files to use normal shell expansion).

infrapy.utils.files2db.**get_or_create_tables**(*options*, *session*, *create=True*)
Load or create canonical ORM KB Core table classes.

options : argparse.ArgumentParser session : sqlalchemy.orm.Session

tables [dict] Mapping between canonical table names and SQLA ORM classes. e.g. {'origin': MyOrigin, . . . }

infrapy.utils.files2db.**get_parser**()
This is where the command-line options are defined, to be parsed from argv

argparse.ArgumentParser instance

Test the parser with this syntax:

```
>>> from sac2db import get_parser
>>> parser = get_parser()
>>> options = parser.parse_args(['--origin', 'origin', '--affiliation',
                                                            'myaffiliation',
 ↪'sqlite:///mydb.sqlite',
                                                            '*.sac'])
>>> print options
Namespace(affiliation='my.affiliation', arrival=None,
assoc=None, event=None, files=['*.sac'], instrument=None, lastid=None,
origin='origin', absolute_path=False, site=None, sitechan=None,
url='sqlite:///mydb.sqlite', wfdisc=None)
```

infrapy.utils.files2db.**get_plugins**(*options*)
Returns a list of imported plugin function objects.

`infrapy.utils.files2db.`**`get_session`**(*options*)

`infrapy.utils.files2db.`**`get_wfdisc`**(*options*)
Return a sequence of WFDISC file names from either a list of file names (trivial) or a text file list (presumable because there are too many files to use normal shell expansion).

`infrapy.utils.files2db.`**`main`**(*argv=None*)
Command-line arguments are created and parsed, fed to functions.

`infrapy.utils.files2db.`**`make_atomic`**(*last*, ***rows*)
Unify related table instances/row, including: ids, dir, and dfile

`infrapy.utils.files2db.`**`ms2db`**(*files*, *url*, *tables*, *plugins=None*, *abs_paths=False*)

1.10.7.6 infrapy.utils.get_arraywaveforms module

`infrapy.utils.get_arraywaveforms.`**`get_arraywaveforms`**(*session*, *Site*, *Wfdisc*, *array*, *t0=None*, *te=None*, *channel=None*)

`infrapy.utils.get_arraywaveforms.`**`get_channel`**(*session*, *Site*, *Wfdisc*, *array*)

1.10.7.7 infrapy.utils.get_header_table module

`infrapy.utils.get_header_table.`**`get_header_table`**(*table_name*)

1.10.7.8 infrapy.utils.get_mean_locations module

`infrapy.utils.get_mean_locations.`**`get_mean_locations`**(*session*, *net*, *Affiliation*, *Site*, *t0_E_jday*, *te_E_jday*)
Get mean location information for a given network.

session : SQLAlchemy Session instance net : str

Network code.

Affiliation, Site [SQLAlchemy ORM mapped table classes] Affiliation, Site classes (not instances).

t0_E_jday, te_E_jday [int] Julian date (YYYJJJ) Site.ondate, offdate.

refsta [list of dict] One dict for each unique refsta, each dict contains mean lat, lon, elev, values in ‘lat’, ‘lon’, ‘elev’ keys, name of refsta in ‘name’ key, and number of stations in the means in ‘numsta’.

1.10.7.9 infrapy.utils.latlon module

`infrapy.utils.latlon.`**`azdiff`**(*az1*, *az2*)
AZDIFF Returns the angle between azimuths

Usage: d=azdiff(az1,az2)

Description: D=AZDIFF(AZ1,AZ2) calculates the angle D from AZ1 to AZ2. D is always within +/-180.

Notes:

Examples: # 340 –> 20 should give 40: >>> azdiff(340,20) array([[40.]])

# 20 –> 340 should give -40: >>> azdiff(20,340) array([[-40.]])

# Several examples with equivalent azimuths: >>> azdiff([180,0,540],[-180,720,-180]) array([[0., 0., 0.]])

See also: AZMEAN, AZINRNG, INLONRNG, LONMOD, INLATLONBOX, LATMOD

`infrapy.utils.latlon.`**`azinrng`**(*azrng*, *az*)

AZINRNG Returns TRUE for azimuths within azimuth range

Usage: tf=azinrng(azrng,az)

Description: TF=AZINRNG(AZRNG,AZ) returns a logical matrix TF of equal shape to AZ with each element set to TRUE for the elements of AZ that are within the range AZRNG as [MIN MAX] (e.g., including MIN & MAX). Note that wrapping is done on all azimuths. So if the range is 10 to 20 deg then values within that range and 370 to 380, -350 to -340, etc all are within the range. Also note that the range always extends clockwise from AZMIN to AZMAX. To get the full azimuth range set the limits such that AZMAX-AZMIN equals 360. If you set AZMIN=0 and AZMAX=361 then you will actually have a range from 0 to 1.

Notes:

- This is just a wrapper for INLONRNG.

Examples: % "Dateline" check: >>> azinrng([178,-178],np.arange(177,184)) array([False, True, True, True, True, True, False], dtype=bool)

See also: INLONRNG, AZDIFF, AZMEAN, LONMOD, LATMOD, FIXLATLON, INLATLONBOX

`infrapy.utils.latlon.`**`azmean`**(*az*, *dim=None*)

AZMEAN Returns the mean azimuth of a set of azimuths

Usage: avg=azmean(az) avg=azmean(az,dim)

Description: AVG=AZMEAN(AZ) returns the average azimuth of the azimuths in AZ. AZ is expected in degrees. Operates down the first non-singleton dimension.

AVG=AZMEAN(AZ,DIM) takes the mean across along dimension DIM.

Notes:

- NaNs are allowed and are ignored.

Examples: # Average North scattered azimuths: >>> azmean([-5,-15,5]) array([-5.])

# Test dimension input: >>> azmean([-5,-15,5],0) array([-5., -15., 5.]) >>> azmean([-5,-15,5],1) array([-5.]) >>> azmean([[-5,-15,5],[20,40,0]],0) array([7.5, 12.5, 2.5]) >>> azmean([[-5,-15,5],[20,40,0]],1) array([-5., 20.])

See also: AZDIFF, AZINRNG, ARRAYCENTER

`infrapy.utils.latlon.`**`fixlatlon`**(*latlon*)

FIXLATLON Returns latitudes & longitudes in reasonable ranges

Usage: latlon=fixlatlon(latlon)

Description: LATLON=FIXLATLON(LATLON) returns latitudes within the range +/-90 and longitudes within +/-180 taking care to preserve the actual corresponding location. The input LATLON must be Nx2 real array.

Notes:

Examples: # Some dumb programs may go "over the pole" in terms of latitude. # FIXLATLON can fix this while handling the accompanying shift in # longitude: >>> fixlatlon([100, 0]) array([[80., 180.]])

# Bring some longitudes from 0-360 to -180-180: >>> fixlatlon([[80, 210],[-45,10],[-5,315]]) array([[80., -150.],

[-45., 10.], [-5., -45.]])

See also: LATMOD, LONMOD, INLONRNG, INLATLONBOX

`infrapy.utils.latlon.`**`gc_intersect`**(*latlona0*, *latlona1*, *latlonb0*, *latlonb1*)

GC_INTERSECT Return intersection points between great circles

Usage: latloni0,latloni1=gc_intersect(latlona0,latlona1, latlonb0,latlonb1)

Description: LATLONI0,LATLONI1=GC_INTERSECT(LATLONA0,LATLONA1,LATLONB0,LATLONB1) finds the intersection points of great circles given by points LATLONA0/1 with great circles given by points LATLONB0/1. Great circles either intersect twice or are equal. LATLONI0 has 1 intersection point and LATLONI1 gives the other (antipodal to the first). When two great circles are equal both intersection points are set to NaNs. All LATLON must either be scalar points (1x2) or arrays (Nx2) with the same shape. This allows finding intersections between one great circle and several others or to find intersections between distinct pairs. All inputs must be in degrees! Outputs are in degrees.

Notes:

- Assumes positions are given in geocentric coordinates.

Examples: #

See also: DEGDIST_FROM_GC, CLOSEST_POINT_ON_GC, GC2LATLON, GCARC2LATLON, GCARC_INTERSECT

`infrapy.utils.latlon.`**`gcarc_intersect`**(*latlona0*, *latlona1*, *latlonb0*, *latlonb1*)

GCARC_INTERSECT Return intersection points between great circle arcs

Usage: latloni=gcarc_intersect(latlona0,latlona1,latlonb0,latlonb1)

Description: LATLONI=GCARC_INTERSECT(LATLONA0,LATLONA1,LATLONB0,LATLONB1) finds the intersection points of great circle arcs given by points LATLONA0/1 with great circle arcs given by points LATLONB0/1. Arcs are the shortest path between points (no arcs are >180deg in length)! Great circle arcs either intersect once or not at all (equal arcs are always treated as non-intersecting). LATLONI gives the intersection points. When two great circle arcs do not intersect (or are equal) the corresponding intersection is set to NaNs. All LATLON must either be scalar points (1x2) or arrays (Nx2) with the same shape. This allows finding intersections between one great circle arc and several others or to find intersections between distinct pairs. All inputs must be in degrees! Outputs are in degrees.

Notes:

- Arcs are always <=180deg!
- Assumes positions are given in geocentric coordinates.

Examples: # Simple case: >>> gcarc_intersect([0,0],[0,10],[-5,5],[5,5]) array([[-0., 5.]])

# Do 2 great circle arcs on the same great circle intersect? >>> gcarc_intersect([0,0],[0,10],[0,5],[0,15]) array([[nan, nan]])

# How about endpoints? >>> gcarc_intersect([0,0],[0,10],[0,0],[10,0]) array([[nan, nan]]) >>> gcarc_intersect([0,0],[0,10],[-10,0],[10,0]) array([[nan, nan]])

# You can also do several intersections: >>> gcarc_intersect([0,0],[0,10],[[0,5],[-5,5]],[[0,15],[5,5]]) array([[nan, nan],

[-0., 5.]])

See also: DEGDIST_FROM_GC, CLOSEST_POINT_ON_GC, GC2LATLON, GCARC2LATLON, GC_INTERSECT

`infrapy.utils.latlon.`**`geocentric2geographiclat`**(*lat*, *ecc=0.08181919084262149*)
GEOCENTRIC2GEOGRAPHICLAT Convert latitude from geocentric to geographic

Usage: lat=geocentric2geographiclat(lat) lat=geocentric2geographiclat(lat,ecc)

Description: LAT=GEOCENTRIC2GEOGRAPHICLAT(LAT) converts geocentric latitudes LAT to geographic latitudes. LAT is in degrees. Assumes the WGS-84 reference ellipsoid.

LAT=GEOCENTRIC2GEOGRAPHICLAT(LAT,ECC) specifies the eccentricity for the ellipsoid to use in the conversion.

Notes:

- If the location is not on the surface use GEOCENTRIC2GEOGRAPHIC.

Examples: # At what geographic latitude is 45N & 60N?: >>> geocentric2geographiclat((45,60)) array([45.19242322, 60.16636419])

# Show the difference in latitudes (geographic pushes to the poles): import matplotlib.pyplot as plt x=np.arange(-90,91) plt.plot(x,geocentric2geographiclat(x)-x) plt.xlabel('geocentric latitude (^o)') plt.ylabel('geographic adjustment (^o)') plt.show()

See also: GEOGRAPHIC2GEOCENTRICLAT, GEOGRAPHICLAT2RADIUS

`infrapy.utils.latlon.`**`geocentric2xyz`**(*latlon*, *radius=1.0*)
GEOCENTRIC2XYZ Converts coordinates from geocentric to cartesian

Usage: xyz=geocentric2xyz(latlon) xyz=geocentric2xyz(latlon,radius)

Description: XYZ=GEOCENTRIC2XYZ(LATLON) converts coordinates in geocentric latitude, longitude, radius to Earth-centered, Earth-Fixed (ECEF). LATLON is in degrees and should be a Nx2 array. XYZ will contain unit vectors.

XYZ=GEOCENTRIC2XYZ(LATLON,RADIUS) converts coordinates in geocentric latitude, longitude, radius to Earth-centered, Earth-Fixed (ECEF). LATLON is in degrees and should be a Nx2 array. XYZ will match the units of RADIUS.

Notes:

- The ECEF coordinate system has the X axis passing through the equator at the prime meridian, the Z axis through the north pole and the Y axis through the equator at 90 degrees longitude.

Examples: # Test the ECEF definition: >>> geocentric2xyz([[0,0],[0,90],[90,0]],[[1],[1],[1]]) array([[1.00000000e+00, 0.00000000e+00, 0.00000000e+00],

[6.12323400e-17, 1.00000000e+00, 0.00000000e+00], [6.12323400e-17, 0.00000000e+00, 1.00000000e+00]])

See also: XYZ2GEOCENTRIC

`infrapy.utils.latlon.`**`geographic2geocentriclat`**(*lat*, *ecc=0.08181919084262149*)
GEOGRAPHIC2GEOCENTRICLAT Convert latitude from geographic to geocentric

Usage: lat=geographic2geocentriclat(lat) lat=geographic2geocentriclat(lat,ecc)

Description: LAT=GEOGRAPHIC2GEOCENTRICLAT(LAT) converts geographic latitudes LAT to geocentric latitudes. LAT is in degrees. Assumes the WGS-84 reference ellipsoid.

LAT=GEOGRAPHIC2GEOCENTRICLAT(LAT,ECC) specifies the eccentricity for the ellipsoid to use in the conversion.

Notes:

- If the location is not on the surface use GEOGRAPHIC2GEOCENTRIC.

Examples: # Get the geocentric latitude for St. Louis, MO & Los Alamos, NM: >>> geographic2geocentriclat((38.649,35.891)) array([38.46142446, 35.70841385])

% Show the difference in latitudes (geocentric pushes to the equator): import matplotlib.pyplot as plt x=np.arange(-90,91) plt.plot(x,geographic2geocentriclat(x)-x) plt.xlabel('geographic latitude (^o)') plt.ylabel('geocentric adjustment (^o)') plt.show()

See also: GEOCENTRIC2GEOGRAPHICLAT, GEOGRAPHICLAT2RADIUS

`infrapy.utils.latlon.`**`geographiclat2radius`**`(`*lat*, *ellipsoid=None*`)`

GEOGRAPHICLAT2RADIUS Returns the radius at a geographic latitude

Usage: radius=geographiclat2radius(lat) radius=geographiclat2radius(lat,ellipsoid)

Description: RADIUS=GEOGRAPHICLAT2RADIUS(LAT) returns the radii at geographic latitudes in LAT. LAT must be in degrees. Assumes the WGS-84 reference ellipsoid.

RADIUS=GEOGRAPHICLAT2RADIUS(LAT,ELLIPSOID) allows specifying the ellipsoid as [A, F] where the parameters A (equatorial radius in kilometers) and F (flattening) must be scalar and float.

Notes:

Examples: # Get the radius for St. Louis, MO & Los Alamos, NM: >>> geographiclat2radius((38.649,35.891)) array([6369.83836784, 6370.82788438])

# What is the radius at the pole & equator?: >>> geographiclat2radius((90,0)) array([6356.75231425, 6378.137])

See also: GEOCENTRIC2GEOGRAPHICLAT, GEOGRAPHIC2GEOCENTRICLAT

`infrapy.utils.latlon.`**`haversine`**`(`*latlon0*, *latlon1*`)`

HAVERSINE Returns distance between 2 points using the Haversine formula

Usage: gcdist=haversine(latlon0,latlon1)

Description: GCDIST=HAVERSINE(LATLON0,LATLON1) finds the spherical great-circle-arc degree distance between two points GCDIST. LATLON0 & LATLON1 must all be in degrees and the latitudes must be geocentric.

Notes:

- 'half versed sine' is better suited for accuracy at small distances compared to SPHERICALINV as it uses a half-versed sine function rather than a cosine which becomes inefficient at small distances.

Examples: # Plot distance discrepancy for SPHERICALINV and HAVERSINE: d2m=1000*6371*np.pi/180 d0=10**(np.linspace(-1,7.2,num=83))/d2m d0=d0[:,np.newaxis] pos=np.zeros((83,2)) pos[:,[1]]=d0 dc,_,_=sphericalinv((0,0),pos) dh=haversine((0,0),pos) hsph,=plt.plot(d2m*d0,d2m*abs(d0-dc),'r',label='SPHERICALINV') hhav,=plt.plot(d2m*d0,d2m*abs(d0-dh),'g',label='HAVERSINE') plt.xscale('log') plt.yscale('log') plt.xlabel('distance (m)') plt.ylabel('discrepancy (m)') plt.legend(handles=[hsph,hhav]) plt.show() # demonstrates convincingly this function is more accurate!

# St. Louis, MO to Yaounde, Cameroon: >>> haversine((38.649,-90.305),(3.861,11.521)) array([[96.75578437]])

# St. Louis, MO to Isla Isabella, Galapagos: >>> haversine((38.649,-90.305),(-0.823,-91.097)) array([[39.47872366]])

# St. Louis, MO to Los Alamos, NM: >>> haversine((38.649,-90.305),(35.891,-106.298)) array([[13.00431661]])

See also: SPHERICALINV, VINCENTYINV, SPHERICALFWD, VINCENTYFWD

`infrapy.utils.latlon.`**`inlatlonbox`**(*latrng*, *lonrng*, *latlon*)

INLATLONBOX Returns TRUE for positions within the specified region

Usage: tf=inlatlonbox(latrng,lonrng,latlon)

Description: TF=INLATLONBOX(LATRNG,LONRNG,LATLON) returns TRUE or FALSE depending on if the positions given by LATLON are within the region specified by LATRNG & LONRNG. LATRNG & LONRNG should be specified as [MIN MAX] and LATLON should be given as [LAT LON]. TF is a NROWSx1 logical array where NROWS is the number of rows in LATLON.

Notes: - Does not call FIXLATLON!

Examples: # Make a region covering 1/4th the Earth and see how close to 1/4th # of a random set of positions is in the region: np.sum(inlatlonbox([0 90],[0 180],randlatlon(1000)))/1000

# Longitude wrapping just works: >>> inlatlonbox([0,45],[160, 200],[[30,-170],[0,-180],[20,190]]) array([True, True, True], dtype=bool) >>> inlatlonbox([0,45],[160, 200],[[30,-160],[45,180],[-20,-170]]) array([True, True, False], dtype=bool)

# Test pole handling: >>> inlatlonbox([90,90],[180,180],[90,0]) array([False], dtype=bool) >>> inlatlonbox([90,90],[180,180],[90,180]) array([True], dtype=bool) >>> inlatlonbox([90,90],[180,180],[90,-180]) array([True], dtype=bool)

See also: INLONRNG, FIXLATLON, LATMOD, LONMOD, AZINRNG

`infrapy.utils.latlon.`**`inlonrng`**(*rng*, *lon*)

INLONRNG Returns TRUE for longitudes within the specified range

Usage: tf=inlonrng(rng,lon)

Description: TF=INLONRNG(RNG,LON) returns TRUE or FALSE depending on if LON is in the longitude range specified by RNG. RNG must be [MIN MAX] and handles wraparound of longitudes.

Notes:

Examples: % A few tests that should return TRUE: >>> inlonrng([-10, 10],360) array([True], dtype=bool) >>> inlonrng([170, 190],-180) array([True], dtype=bool) >>> inlonrng([170, 190],180) array([True], dtype=bool) >>> inlonrng([-190, -170],-180) array([True], dtype=bool) >>> inlonrng([-190, -170],180) array([True], dtype=bool) >>> inlonrng([350, 370],0) array([True], dtype=bool)

% A few tougher cases that should return TRUE: >>> inlonrng([0, 0],360) array([True], dtype=bool) >>> inlonrng([180, 180],-180) array([True], dtype=bool) >>> inlonrng([180, 180],180) array([True], dtype=bool) >>> inlonrng([-180, -180],-180) array([True], dtype=bool) >>> inlonrng([-180, -180],180) # FIXME IT FAILS array([True], dtype=bool) >>> inlonrng([360, 360],0) array([True], dtype=bool)

% Yet more cases that should return TRUE: >>> inlonrng([0, 360],0) array([True], dtype=bool) >>> inlonrng([0, 360],360) array([True], dtype=bool) >>> inlonrng([-180, 180],-180) array([True], dtype=bool) >>> inlonrng([-180, 180],180) array([True], dtype=bool)

See also: AZINRNG, LONMOD, LATMOD, FIXLATLON, INLATLONBOX

`infrapy.utils.latlon.`**`latmod`**`(`*lat*, *wrap=90*`)`

LATMOD Returns a latitude modulus (i.e., wraps latitudes into range)

Usage: wlat,px=latmod(lat) wlat,px=latmod(lat,wrap)

Description: WLAT,PX=LATMOD(LAT) returns latitudes LAT to be within the range +/-90 and also returns the number of pole-crossings each latitude in LAT made in PX. For example, if LAT=100 & WRAP=90 then WLAT=80 & PX=1. This is useful for coupling LATMOD with LONMOD to preserve the positions while reducing the values to reasonable ranges. See the Examples section below for instructions on how to use this output. Note that LATMOD expects floats and so integers are converted to floats such that WLAT and PX are always float. LAT may be a numpy array.

WLAT,PX=LATMOD(LAT,WRAP) is S.*(LAT-N.*WRAP) where N=round(LAT./WRAP) if WRAP~=0 and S=1-2.*MOD(N,2). Thus LATMOD(LAT,WRAP) is always within the range +/-WRAP and forms a continuous function (but is discontinuous in the 1st derivative) that looks like a “triangle wave”. The function is primarily intended for wrapping latitude values back to a valid range. The inputs LAT and WRAP must be equal in shape or scalar. WRAP is optional and defaults to 90.

Notes:

By convention: C,PX=latmod(A,0) returns C=A, PX=inf C,PX=latmod(A,A) returns C=A, PX=0 C,PX=latmod(0,0) returns C=0, PX=nan

Examples: # Modifying the latitude should also take into account the longitude # shift necessary to preserve the actual position. This may be done # by utilizing the second output to shift the longitude by 180 degrees # if there are an odd number of pole-crossings: lat,px=latmod(lat) lon=lonmod(lon+np.mod(px,2)*180)

# Try a few cases: >>> latmod((-89,-90,-91)) (array([-89., -90., -89.]), array([-0., 0., 1.])) >>> latmod((-269,-270,-271)) (array([89., 90., 89.]), array([1., 1., 2.])) >>> latmod((89,90,91)) (array([89., 90., 89.]), array([-0., 0., 1.])) >>> latmod((269,270,271)) (array([-89., -90., -89.]), array([1., 1., 2.]))

# Now playing with wrap=0: >>> latmod((180, 0, 54.3),0) (array([180. , 0. , 54.3]), array([inf, nan, inf]))

See also: LONMOD, FIXLATLON, INLONRNG, INLATLONBOX

`infrapy.utils.latlon.`**`lonmod`**`(`*lon*, *wrap=360*`)`

LONMOD Returns a longitude modulus (i.e., wraps longitudes into range)

Usage: wlon=lonmod(lon) wlon=lonmod(lon,wrap)

Description: WLON=LONMOD(LON) returns the longitudes LON within the range +/-180. For example, LON=181 is output as -179. See the other usage form for more algorithm details. LON must be array_like and composed of floats or integers. WLON is an ndarray of floats.

WLON=LONMOD(LON,WRAP) is LON-N.*WRAP where N=round(LON./WRAP) if WRAP~=0. Thus LONMOD(LON,WRAP) is always within the range +/-(WRAP/2). The inputs LON and WRAP must be equal in shape or scalar. WRAP is optional and defaults to 360.

Notes:

- **By convention:** C=lonmod(A,0) returns C=A C=lonmod(A,A) returns C=0

Examples: # Try a few cases: >>> lonmod((181, 180, -180, -181, 360, -360, 540, -540)) array([-179., 180., -180., 179., 0., 0., 180., -180.])

# Now playing with wrap=0: >>> lonmod((180, 0, 54.3),0) array([180. , 0. , 54.3])

# wrap=lon: >>> lonmod((180, 0, 54.3),(180, 0, 54.3)) array([0., 0., 0.])

See also: LATMOD, FIXLATLON, INLONRNG, INLATLONBOX

`infrapy.utils.latlon.`**`randlatlon`**(*n*)

RANDLATLON Returns lat/lon points uniformly distributed on a sphere

Usage: latlon=randlatlon(n)

Description: LATLON=RANDLATLON(N) returns the random position of N points uniformly distributed on a sphere (not within). The output LATLON is formatted as a Nx2 array of [LAT LON] where the latitudes and longitudes are in degrees. Longitudes are in the range -180 to 180 degrees.

Notes:

Examples: # Get 5 random points in latitude & longitude: randlatlon(5)

# Map 1000 random lat/lon points: p=randlatlon(1000) from mpl_toolkits.basemap import Basemap import matplotlib.pyplot as plt map=Basemap(projection='hammer',lon_0=0) map.drawcoastlines() x,y=map(p[:,1],p[:,0]) map.plot(x,y,'ro') plt.show()

# Output is Nx2: >>> np.shape(randlatlon(1000)) (1000, 2)

See also: RANDSPHERE

`infrapy.utils.latlon.`**`randsphere`**(*n*, *dim=3*)

RANDSPHERE Returns points uniformly distributed within a sphere

Usage: x=randsphere(n) x=randsphere(n,dim)

Description: X=RANDSPHERE(N) returns the random positions of N points uniformly distributed within a unit sphere. The output X is a Nx3 array of cartesian coordinates.

X=RANDSPHERE(N,DIM) allows getting points in an DIM dimensional sphere. DIM should be a scalar integer >=2.

Notes:

Examples: # Return 1000 points and plot in 3D: x=randsphere(1000) import matplotlib.pyplot as plt from mpl_toolkits.mplot3d import Axes3D fig = plt.figure() ax = Axes3D(fig) ax.scatter(x[:,0],x[:,1],x[:,2]) plt.show()

# output is NxM >>> np.shape(randsphere(1000)) (1000, 3) >>> np.shape(randsphere(1000,5)) (1000, 5)

See also: RANDLATLON

`infrapy.utils.latlon.`**`sphericalfwd`**(*latlon0*, *gcdist*, *az*)

SPHERICALFWD Finds a point on a sphere relative to another point

Usage: latlon1,baz=sphericalfwd(latlon0,gcdist,az)

Description: LATLON1,BAZ=SPHERICALFWD(LATLON0,GCDIST,AZ) finds geocentric latitudes and longitudes in LATLON1 of destination point(s), as well as the back azimuths BAZ, given the great circle distances GCDIST and forward azimuths AZ from initial point(s) with geocentric latitudes and longitudes in LATLON0. Inputs must all be in degrees. Outputs are also all in degrees.

Notes:

- Latitudes are geocentric (0 deg lat == equator, range +/-90)
- Longitudes are returned in the range +/-180
- Backazimuth is returned in the range 0-360

Examples: # Heading 45deg NW of St. Louis, MO to ???: >>> sphericalfwd((38.649,-90.305),45,-30) (array([[66.90805234, -154.65352366]]), array([[95.35931627]]))

# Go 10deg South of Los Alamos, NM to ???: >>> sphericalfwd((35.891,106.298),10,180) (array([[25.891, 106.298]]), array([[0.]]))

See also: SPHERICALINV

`infrapy.utils.latlon.`**`sphericalinv`**(*latlon0*, *latlon1*)

SPHERICALINV Return distance and azimuths between 2 locations on sphere

Usage: gcdist,az,baz=sphericalinv(latlon0,latlon1)

Description: GCDIST,AZ,BAZ=SPHERICALINV(LATLON0,LATLON1) returns the great-circle-arc degree distances GCDIST, forward azimuths AZ and backward azimuths BAZ between initial point(s) with geocentric latitudes and longitudes LATLON0 and final point(s) with geocentric latitudes and longitudes LATLON1 on a sphere. Inputs must be in degrees. Outputs are also all in degrees.

Notes:

- Will always return the shorter distance (GCDIST<=180)
- GCDIST accuracy degrades when < 3000km (see HAVERSINE example!)
- Azimuths are returned in the range 0<=az<=360

Examples: # St. Louis, MO to Yaounde, Cameroon: >>> sphericalinv((38.649,-90.305),(3.861,11.521)) (array([[96.75578437]]), array([[79.53972827]]), array([[309.66814964]]))

# St. Louis, MO to Isla Isabella, Galapagos: >>> sphericalinv((38.649,-90.305),(-0.823,-91.097)) (array([[39.47872366]]), array([[181.24562107]]), array([[0.97288389]]))

# St. Louis, MO to Los Alamos, NM: >>> sphericalinv((38.649,-90.305),(35.891,-106.298)) (array([[13.00431661]]), array([[262.71497244]]), array([[72.98729659]]))

See also: SPHERICALFWD

`infrapy.utils.latlon.`**`xyz2geocentric`**(*xyz*)

XYZ2GEOCENTRIC Converts coordinates from cartesian to geocentric

Usage: latlon,radius=xyz2geocentric(xyz)

Description: LATLON,RADIUS=XYZ2GEOCENTRIC(XYZ) converts arrays of coordinates in Earth-centered, Earth-Fixed (ECEF) to geocentric latitude, longitude, & radius. LAT and LON are in degrees. X, Y and Z of XYZ (a Nx3 array of floats) must have the same units (RADIUS will will be in those units).

Notes:

- The ECEF coordinate system has the X axis passing through the equator at the prime meridian, the Z axis through the north pole and the Y axis through the equator at 90 degrees longitude.

Examples: # Test the ECEF definition: >>> xyz2geocentric([[1, 0, 0],[0, 1, 0],[0, 0, 1]]) (array([[0., 0.],

[0., 90.], [90., 0.]]), array([[1.], [1.], [1.]]))

See also: GEOCENTRIC2XYZ

1.10.7.10 infrapy.utils.ms2db module

class infrapy.utils.ms2db.**CoreTable**(*name*, *prototype*, *table*)
Bases: tuple

property name
Alias for field number 0

property prototype
Alias for field number 1

property table
Alias for field number 2

infrapy.utils.ms2db.**apply_plugins**(*plugins*, ***rows*)

infrapy.utils.ms2db.**dicts2rows**(*dicts*, *classes*)

infrapy.utils.ms2db.**expand_glob**(*option*, *opt_str*, *value*, *parser*)
Returns an iglob iterator for file iteration. Good for large file lists.

infrapy.utils.ms2db.**get_files**(*options*)
Return a sequence of SAC file names from either a list of file names (trivial) or a text file list (presumable because there are too many files to use normal shell expansion).

infrapy.utils.ms2db.**get_or_create_tables**(*options*, *session*, *create=True*)
Load or create canonical ORM KB Core table classes.

options : argparse.ArgumentParser session : sqlalchemy.orm.Session

tables [dict] Mapping between canonical table names and SQLA ORM classes. e.g. {'origin': MyOrigin, . . . }

infrapy.utils.ms2db.**get_parser**()
This is where the command-line options are defined, to be parsed from argv

argparse.ArgumentParser instance

Test the parser with this syntax:

```
>>> from sac2db import get_parser
>>> parser = get_parser()
>>> options = parser.parse_args(['--origin', 'origin', '--affiliation',
                                                        'myaffiliation',
↪'sqlite:///mydb.sqlite',
                                                        '*.sac'])
>>> print options
Namespace(affiliation='my.affiliation', arrival=None,
assoc=None, event=None, files=['*.sac'], instrument=None, lastid=None,
origin='origin', absolute_path=False, site=None, sitechan=None,
url='sqlite:///mydb.sqlite', wfdisc=None)
```

infrapy.utils.ms2db.**get_plugins**(*options*)
Returns a list of imported plugin function objects.

infrapy.utils.ms2db.**get_session**(*options*)

infrapy.utils.ms2db.**main**(*argv=None*)
Command-line arguments are created and parsed, fed to functions.

infrapy.utils.ms2db.**make_atomic**(*last*, ***rows*)
Unify related table instances/row, including: ids, dir, and dfile

infrapy.utils.ms2db.**ms2db**(*files*, *url*, *tables*, *plugins=None*, *abs_paths=False*)

1.10.7.11 infrapy.utils.obspy_conversion module

`infrapy.utils.obspy_conversion.`**`stream2array`**`(`*str*`)`

turn obspy stream into 2D array

the data from stream is extracted and copied to a 2D numpy array

- **** parameters**, types, return, and return types::**

  param arg1 stream to be converted

  type arg1 obspy stream

  return 2D array with time series for all channels

  rtype numpy array

1.10.7.12 infrapy.utils.prog_bar module

`infrapy.utils.prog_bar.`**`close`**`()`

`infrapy.utils.prog_bar.`**`increment`**`(`*n=1*`)`

`infrapy.utils.prog_bar.`**`prep`**`(`*length*`)`

`infrapy.utils.prog_bar.`**`set_step`**`(`*n*, *N*, *len*`)`

1.10.7.13 infrapy.utils.ref2sac module

`infrapy.utils.ref2sac.`**`beta`**`(`*a*, *b*, *size=None*`)`

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a,b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalization, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

Note: New code should use the `beta` method of a `default_rng()` instance instead; see *random-quick-start*.

a [float or array_like of floats] Alpha, positive (>0).

b [float or array_like of floats] Beta, positive (>0).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized beta distribution.

Generator.beta: which should be used for new code.

`infrapy.utils.ref2sac.`**`binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Note: New code should use the `binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

n [int or array_like of ints] Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

p [float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

scipy.stats.binom [probability density function, distribution or] cumulative density function, etc.

Generator.binomial: which should be used for new code.

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N}p^N(1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

`infrapy.utils.ref2sac.`**`chisquare`**`(`*df*, *size=None*`)`

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Note: New code should use the `chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

df [float or array_like of floats] Number of degrees of freedom, must be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized chi-square distribution.

ValueError When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

Generator.chisquare: which should be used for new code.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272]) # random
```

`infrapy.utils.ref2sac.`**`choice`**(*a*, *size=None*, *replace=True*, *p=None*)

Generates a random sample from a given 1-D array

New in version 1.7.0.

Note: New code should use the `choice` method of a `default_rng()` instance instead; see *random-quick-start*.

a [1-D array-like or int] If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a were np.arange(a)

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

replace [boolean, optional] Whether the sample is with or without replacement

p [1-D array-like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

samples [single item or ndarray] The generated random samples

ValueError If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

randint, shuffle, permutation Generator.choice: which should be used in new code

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

`infrapy.utils.ref2sac.`**`dirichlet`**`(`*alpha*, *size=None*`)`

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. The Dirichlet distribution is a conjugate prior of a multinomial distribution in Bayesian inference.

Note: New code should use the `dirichlet` method of a `default_rng()` instance instead; see *random-quick-start*.

alpha [array] Parameter of the distribution (k dimension for sample of dimension k).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

samples [ndarray,] The drawn samples, of shape (size, alpha.ndim).

ValueError If any value in alpha is less than or equal to zero

Generator.dirichlet: which should be used for new code.

The Dirichlet distribution is a distribution over vectors x that fulfil the conditions $x_i > 0$ and $\sum_{i=1}^{k} x_i = 1$.

The probability density function p of a Dirichlet-distributed random vector X is proportional to

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1},$$

where α is a vector containing the positive concentration parameters.

The method uses the following property for computation: let Y be a random vector which has components that follow a standard gamma distribution, then $X = \frac{1}{\sum_{i=1}^{k} Y_i} Y$ is Dirichlet-distributed

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```

`infrapy.utils.ref2sac.`**`exponential`**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

Note: New code should use the `exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

scale [float or array_like of floats] The scale parameter, $\beta = 1/\lambda$. Must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized exponential distribution.

Generator.exponential: which should be used for new code.

`infrapy.utils.ref2sac.f(dfnum, dfden, size=None)`

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters must be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Note: New code should use the `f` method of a `default_rng()` instance instead; see *random-quick-start*.

dfnum [float or array_like of floats] Degrees of freedom in numerator, must be > 0.

dfden [float or array_like of float] Degrees of freedom in denominator, must be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Fisher distribution.

scipy.stats.f [probability density function, distribution or] cumulative density function, etc.

Generator.f: which should be used for new code.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> np.sort(s)[-10]
7.61988120985 # random
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`infrapy.utils.ref2sac.`**`gamma`**`(`*shape*, *scale=1.0*, *size=None*`)`

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

Note: New code should use the `gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

shape [float or array_like of floats] The shape of the gamma distribution. Must be non-negative.

scale [float or array_like of floats, optional] The scale of the gamma distribution. Must be non-negative. Default is equal to 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized gamma distribution.

scipy.stats.gamma [probability density function, distribution or] cumulative density function, etc.

Generator.gamma: which should be used for new code.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`geometric`**`(`*p*, *size=None*`)`

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where *p* is the probability of success of an individual trial.

Note: New code should use the `geometric` method of a `default_rng()` instance instead; see *random-quick-start*.

p [float or array_like of floats] The probability of success of an individual trial.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized geometric distribution.

Generator.geometric: which should be used for new code.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

`infrapy.utils.ref2sac.`**`get_state`**`()`

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [{tuple(str, ndarray of 624 uints, int, int, float), dict}] The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If *legacy* is False, or the BitGenerator is not NT19937, then state is returned as a dictionary.

legacy [bool] Flag indicating the return a legacy tuple state when the BitGenerator is MT19937.

set_state

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

`infrapy.utils.ref2sac.`**`gumbel`**`(`*loc=0.0*, *scale=1.0*, *size=None*`)`

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Note: New code should use the `gumbel` method of a `default_rng()` instance instead; see *random-quick-start*.

loc [float or array_like of floats, optional] The location of the mode of the distribution. Default is 0.

scale [float or array_like of floats, optional] The scale parameter of the distribution. Default is 1. Must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Gumbel distribution.

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme weibull Generator.gumbel: which should be used for new code.

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with “exponential-like” tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a “fat-tailed” distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...     a = np.random.normal(mu, beta, 1000)
...     means.append(a.mean())
...     maxima.append(a.max())
```

(continues on next page)

(continued from previous page)

```
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, *ngood* (ways to make a good selection), *nbad* (ways to make a bad selection), and *nsample* (number of items sampled, which is less than or equal to the sum `ngood + nbad`).

Note: New code should use the `hypergeometric` method of a `default_rng()` instance instead; see *random-quick-start*.

ngood [int or array_like of ints] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like of ints] Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if *ngood*, *nbad*, and *nsample* are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized hypergeometric distribution. Each sample is the number of good items within a randomly selected subset of size *nsample* taken from a set of *ngood* good items and *nbad* bad items.

scipy.stats.hypergeom [probability density function, distribution or] cumulative density function, etc.

Generator.hypergeometric: which should be used for new code.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \le x \le n$ and $n - b \le x \le g$

for P(x) the probability of `x` good results in the drawn sample, g = *ngood*, b = *nbad*, and n = *nsample*.

Consider an urn with black and white marbles in it, *ngood* of them are black and *nbad* are white. If you draw *nsample* balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

`infrapy.utils.ref2sac.`**`laplace`**`(`*loc=0.0*, *scale=1.0*, *size=None*`)`

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Note: New code should use the `laplace` method of a `default_rng()` instance instead; see *random-quick-start*.

loc [float or array_like of floats, optional] The position, μ, of the distribution peak. Default is 0.

scale [float or array_like of floats, optional] λ, the exponential decay. Default is 1. Must be non- negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Laplace distribution.

Generator.laplace: which should be used for new code.

It has the probability density function

$$f(x;\mu,\lambda) = \frac{1}{2\lambda}\exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
```

(continues on next page)

(continued from previous page)

```
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```

`infrapy.utils.ref2sac.`**`logistic`**`(`*loc=0.0*, *scale=1.0*, *size=None*`)`

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Note: New code should use the `logistic` method of a `default_rng()` instance instead; see *random-quick-start*.

loc [float or array_like of floats, optional] Parameter of the distribution. Default is 0.

scale [float or array_like of floats, optional] Parameter of the distribution. Must be non-negative. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized logistic distribution.

scipy.stats.logistic [probability density function, distribution or] cumulative density function, etc.

Generator.logistic: which should be used for new code.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return np.exp((loc-x)/scale)/(scale*(1+np.exp((loc-x)/scale))**2)
>>> lgst_val = logist(bins, loc, scale)
>>> plt.plot(bins, lgst_val * count.max() / lgst_val.max())
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`lognormal`**`(`*mean=0.0*, *sigma=1.0*, *size=None*`)`

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Note: New code should use the `lognormal` method of a `default_rng()` instance instead; see *random-quick-start*.

mean [float or array_like of floats, optional] Mean value of the underlying normal distribution. Default is 0.

sigma [float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Must be non-negative. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized log-normal distribution.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

Generator.lognormal: which should be used for new code.

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.standard_normal(100)
...     b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`logseries`**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < `p` < 1.

Note: New code should use the `logseries` method of a `default_rng()` instance instead; see *random-quick-start*.

p [float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 1).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized logarithmic series distribution.

scipy.stats.logser [probability density function, distribution or] cumulative density function, etc.

Generator.logseries: which should be used for new code.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*np.log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
...          logseries(bins, a).max(), 'r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`multinomial`**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

Note: New code should use the `multinomial` method of a `default_rng()` instance instead; see *random-quick-start*.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the `p` different outcomes. These must sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Generator.multinomial: which should be used for new code.

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]]) # random
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3], # random
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26]) # random
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62]) # random
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
Traceback (most recent call last):
ValueError: pvals < 0, pvals > 1 or pvals contains NaNs
```

`infrapy.utils.ref2sac.`**`multivariate_normal`**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-8*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Note: New code should use the `multivariate_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

size [int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

check_valid [{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

tol [float, optional] Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Generator.multivariate_normal: which should be used for new code.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True] # random
```

`infrapy.utils.ref2sac.`**`negative_binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is > 0 and *p* is in the interval [0, 1].

Note: New code should use the `negative_binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

n [float or array_like of floats] Parameter of the distribution, > 0.

p [float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

Generator.negative_binomial: which should be used for new code.

The probability mass function of the negative binomial distribution is

$$P(N;n,p) = \frac{\Gamma(N+n)}{N!\Gamma(n)}p^{n}(1-p)^{N},$$

where n is the number of successes, p is the probability of success, $N+n$ is the number of trials, and Γ is the gamma function. When n is an integer, $\frac{\Gamma(N+n)}{N!\Gamma(n)} = \binom{N+n-1}{N}$, which is the more common form of this term in the the pmf. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print(i, "wells drilled, probability of one success =", probability)
```

`infrapy.utils.ref2sac.`**`noncentral_chisquare`**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalization of the χ^2 distribution.

Note: New code should use the `noncentral_chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

df [float or array_like of floats] Degrees of freedom, must be > 0.

Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

nonc [float or array_like of floats] Non-centrality, must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

Generator.noncentral_chisquare: which should be used for new code.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

infrapy.utils.ref2sac.**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

Note: New code should use the `noncentral_f` method of a `default_rng()` instance instead; see *random-quick-start*.

dfnum [float or array_like of floats] Numerator degrees of freedom, must be > 0.

Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

dfden [float or array_like of floats] Denominator degrees of freedom, must be > 0.

nonc [float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, must be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

Generator.noncentral_f: which should be used for new code.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

infrapy.utils.ref2sac.**normal**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

Note: New code should use the `normal` method of a `default_rng()` instance instead; see *random-quick-start*.

loc [float or array_like of floats] Mean ("centre") of the distribution.

scale [float or array_like of floats] Standard deviation (spread or "width") of the distribution. Must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized normal distribution.

scipy.stats.norm [probability density function, distribution or] cumulative density function, etc.

Generator.normal: which should be used for new code.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0  # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
```

(continues on next page)

(continued from previous page)

```
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Two-by-four array of samples from N(3, 6.25):

```
>>> np.random.normal(3, 2.5, size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`infrapy.utils.ref2sac.`**`pareto`**`(`*a*, *size=None*`)`

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Note: New code should use the `pareto` method of a `default_rng()` instance instead; see *random-quick-start*.

a [float or array_like of floats] Shape of the distribution. Must be positive.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Pareto distribution.

scipy.stats.lomax [probability density function, distribution or] cumulative density function, etc.

scipy.stats.genpareto [probability density function, distribution or] cumulative density function, etc.

Generator.pareto: which should be used for new code.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]_. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`permutation`**(*x*)

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

Note: New code should use the `permutation` method of a `default_rng()` instance instead; see *random-quick-start*.

x [int or array_like] If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

Generator.permutation: which should be used for new code.

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12]) # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8], # random
       [0, 1, 2],
       [3, 4, 5]])
```

`infrapy.utils.ref2sac.`**`poisson`**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

Note: New code should use the `poisson` method of a `default_rng()` instance instead; see *random-quick-start*.

lam [float or array_like of floats] Expectation of interval, must be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Poisson distribution.

Generator.poisson: which should be used for new code.

The Poisson distribution

$$f(k;\lambda)=\frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k;\lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C int64 type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```

Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

`infrapy.utils.ref2sac.`**`power`**(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Note: New code should use the `power` method of a `default_rng()` instance instead; see *random-quick-start*.

a [float or array_like of floats] Parameter of the distribution. Must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized power distribution.

ValueError If a < 1.

Generator.power: which should be used for new code.

The probability density function is

$$P(x;a)=ax^{a-1},0\le x\le 1,a>0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

`infrapy.utils.ref2sac.`**`rand`**`(`*d0, d1, ..., dn*`)`
Random values in a given shape.

Note: This is a convenience function for users porting code from Matlab, and wraps *random_sample*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, . . . , dn [int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

`infrapy.utils.ref2sac.`**`randint`**(*low*, *high=None*, *size=None*, *dtype='l'*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Note: New code should use the `integers` method of a `default_rng()` instance instead; see *random-quick-start*.

low [int or array-like of ints] Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

high [int or array-like of ints, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

dtype [dtype, optional] Desired dtype of the result. All dtypes are determined by their name, i.e., ‘int64’, ‘int’, etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is *np.int_*.

New in version 1.11.0.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

Generator.integers: which should be used for new code.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

`infrapy.utils.ref2sac.`**`randn`**`(`*d0, d1, ..., dn*`)`

Return a sample (or samples) from the "standard normal" distribution.

Note: This is a convenience function for users porting code from Matlab, and wraps *standard_normal*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Note: New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

If positive int_like arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

d0, d1, . . . , dn [int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Z [ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

standard_normal : Similar, but takes a tuple as its argument. normal : Also accepts mu and sigma arguments. Generator.standard_normal: which should be used for new code.

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`infrapy.utils.ref2sac.`**`random`**`(`*size=None*`)`

Return random floats in the half-open interval [0.0, 1.0). Alias for *random_sample* to ease forward-porting to the new random API.

`infrapy.utils.ref2sac.`**`random_integers`**`(`*low, high=None, size=None*`)`

Random integers of type *np.int_* between *low* and *high*, inclusive.

Return random integers of type *np.int_* from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The *np.int_* type translates to the C long integer type and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

```
>>> np.random.random_integers(5)
4 # random
>>> type(np.random.random_integers(5))
<class 'numpy.int64'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4], # random
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ]) # random
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, density=True)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`random_sample`**`(`*size=None*`)`

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Note: New code should use the `random` method of a `default_rng()` instance instead; see *random-quick-start*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Generator.random: which should be used for new code.

```
>>> np.random.random_sample()
0.47108547995356098 # random
>>> type(np.random.random_sample())
<class 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # random
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984], # random
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`infrapy.utils.ref2sac.`**`rayleigh`**`(`*scale=1.0*, *size=None*`)`

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Note: New code should use the `rayleigh` method of a `default_rng()` instance instead; see *random-quick-start*.

scale [float or array_like of floats, optional] Scale, also equals the mode. Must be non-negative. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Rayleigh distribution.

Generator.rayleigh: which should be used for new code.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003 # random
```

`infrapy.utils.ref2sac.`**`seed`**(*self*, *seed=None*)

Reseed a legacy MT19937 BitGenerator

This is a convenience, legacy function.

The best practice is to **not** reseed a BitGenerator, rather to recreate a new one. This method is here for legacy reasons. This example demonstrates best practice.

```
>>> from numpy.random import MT19937
>>> from numpy.random import RandomState, SeedSequence
>>> rs = RandomState(MT19937(SeedSequence(123456789)))
# Later, you want to restart the stream
>>> rs = RandomState(MT19937(SeedSequence(987654321)))
```

`infrapy.utils.ref2sac.`**`set_state`**(*state*)

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the bit generator used by the RandomState instance. By default, RandomState uses the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [{tuple(str, ndarray of 624 uints, int, int, float), dict}] The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If state is a dictionary, it is directly set using the BitGenerators *state* property.

out [None] Returns 'None' on success.

get_state

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

`infrapy.utils.ref2sac.`**`shuffle`**(*x*)

Modify a sequence in-place by shuffling its contents.

This function only shuffles the array along the first axis of a multi-dimensional array. The order of sub-arrays is changed but their contents remains the same.

Note: New code should use the `shuffle` method of a `default_rng()` instance instead; see *random-quick-start*.

x [array_like] The array or list to be shuffled.

None

Generator.shuffle: which should be used for new code.

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8] # random
```

Multi-dimensional arrays are only shuffled along the first axis:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5], # random
       [6, 7, 8],
       [0, 1, 2]])
```

`infrapy.utils.ref2sac.`**`standard_cauchy`**`(`*size=None*`)`

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Note: New code should use the `standard_cauchy` method of a `default_rng()` instance instead; see *random-quick-start*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The drawn samples.

Generator.standard_cauchy: which should be used for new code.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \bigl[1+ (\frac{x-x_0}{\gamma})^2 \bigr] }$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`standard_exponential`**(*size=None*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Note: New code should use the `standard_exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

Generator.standard_exponential: which should be used for new code.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

`infrapy.utils.ref2sac.`**`standard_gamma`**(*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

Note: New code should use the `standard_gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

shape [float or array_like of floats] Parameter, must be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized standard gamma distribution.

scipy.stats.gamma [probability density function, distribution or] cumulative density function, etc.

Generator.standard_gamma: which should be used for new code.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
...                        (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`standard_normal`**(*size=None*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

Note: New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

out [float or ndarray] A floating-point array of shape `size` of drawn samples, or a single sample if `size` was not specified.

normal : Equivalent function with additional `loc` and `scale` arguments for setting the mean and standard deviation.

Generator.standard_normal: which should be used for new code.

For random samples from $N(\mu, \sigma^2)$, use one of:

```
mu + sigma * np.random.standard_normal(size=...)
np.random.normal(mu, sigma, size=...)
```

```
>>> np.random.standard_normal()
2.1923875335537315 #random
```

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311,  # random
       -0.38672696, -0.4685006 ])                                # random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

Two-by-four array of samples from $N(3, 6.25)$:

```
>>> 3 + 2.5 * np.random.standard_normal(size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`infrapy.utils.ref2sac.`**`standard_t`**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

Note: New code should use the `standard_t` method of a `default_rng()` instance instead; see *random-quick-start*.

df [float or array_like of floats] Degrees of freedom, must be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized standard Student's t distribution.

Generator.standard_t: which should be used for new code.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\Gamma(\frac{df}{2})}\Bigl(1+\frac{x^2}{df} \Bigr)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, density=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

`infrapy.utils.ref2sac.`**`triangular`**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Note: New code should use the `triangular` method of a `default_rng()` instance instead; see *random-quick-start*.

left [float or array_like of floats] Lower limit.

mode [float or array_like of floats] The value where the peak of the distribution occurs. The value must fulfill the condition `left <= mode <= right`.

right [float or array_like of floats] Upper limit, must be larger than *left*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized triangular distribution.

Generator.triangular: which should be used for new code.

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Note: New code should use the `uniform` method of a `default_rng()` instance instead; see *random-quick-start*.

low [float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float or array_like of floats] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized uniform distribution.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

interval `[low, high]`.

random_sample : Floats uniformly distributed over `[0, 1)`. random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

`rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Generator.uniform: which should be used for new code.

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`vonmises`**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Note: New code should use the `vonmises` method of a `default_rng()` instance instead; see *random-quick-start*.

mu [float or array_like of floats] Mode ("center") of the distribution.

kappa [float or array_like of floats] Dispersion of the distribution, has to be >=0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized von Mises distribution.

scipy.stats.vonmises [probability density function, distribution, or] cumulative density function, etc.

Generator.vonmises: which should be used for new code.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`wald`**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Note: New code should use the `wald` method of a `default_rng()` instance instead; see *random-quick-start*.

mean [float or array_like of floats] Distribution mean, must be > 0.

scale [float or array_like of floats] Scale parameter, must be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Wald distribution.

Generator.wald: which should be used for new code.

The probability density function for the Wald distribution is

$$P(x;mean,scale) = \sqrt{\frac{scale}{2\pi x^3}}e^{\frac{-scale(x-mean)^2}{2\cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`weibull`**`(`*a*, *size=None*`)`

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Note: New code should use the `weibull` method of a `default_rng()` instance instead; see *random-quick-start*.

a [float or array_like of floats] Shape parameter of the distribution. Must be nonnegative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Weibull distribution.

scipy.stats.weibull_max scipy.stats.weibull_min scipy.stats.genextreme gumbel Generator.weibull: which should be used for new code.

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

`infrapy.utils.ref2sac.`**`zipf`**`(`*a*, *size=None*`)`

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

Note: New code should use the `zipf` method of a `default_rng()` instance instead; see *random-quick-start*.

a [float or array_like of floats] Distribution parameter. Must be greater than 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

out [ndarray or scalar] Drawn samples from the parameterized Zipf distribution.

scipy.stats.zipf [probability density function, distribution, or] cumulative density function, etc.

Generator.zipf: which should be used for new code.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50], 50, density=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

1.10.7.14 infrapy.utils.sac_stats module

1.10.7.15 infrapy.utils.seed2sac module

1.10.7.16 infrapy.utils.short_time module

`infrapy.utils.short_time.`**`short_time`**(*timeSTR*)

1.10.7.17 infrapy.utils.skew_norm module

`infrapy.utils.skew_norm.`**`pdf`**(*x*, *x0=0.0*, *sigma=1.0*, *alpha=0.0*)

`infrapy.utils.skew_norm.`**`pdf_2comp`**(*x*, *x1*, *s1*, *a1*, *x2*, *s2*, *a2*, *w*)

`infrapy.utils.skew_norm.`**`pdf_3comp`**(*x*, *x1*, *s1*, *a1*, *w1*, *x2*, *s2*, *a2*, *w2*, *x3*, *s3*, *a3*, *w3*)

1.10.7.18 infrapy.utils.zip2ref module

PYTHON MODULE INDEX

INDEX