hesseflux Documentation

Release 2.1

Matthias Cuntz

Jul 15, 2020

CONTENTS

CHAPTER 1

QUICKSTART

`hesseflux` provides functions used in the processing and post-processing of the Eddy covariance flux data of the ICOS ecosystem site FR-Hes.

1.1 About

`hesseflux` collects functions used for processing Eddy covariance data of the ICOS ecosystem site FR-Hes.

The package uses several functions of the JAMS Python package

> https://github.com/mcuntz/jams_python

The JAMS package and hesseflux are synchronised irregularly.

`hesseflux` includes a Python port of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

The post-processing functionality for Eddy flux data is similar to the R-package REddyProc and includes basically the steps described in Papale et al. (Biogeosciences, 2006) plus some extensions such as the daytime method of flux partitioning (Lasslop et al., Global Change Biology 2010).

The complete documentation for `hesseflux` is available from Read The Docs.

> http://hesseflux.readthedocs.org/en/latest/

1.2 Quick usage guide

An example script that makes all the steps described in Papale et al. (Biogeosciences, 2006) is given in the example directory. It is simply called:

```
python postproc_europe-fluxdata.py hesseflux_example.cfg
```

The script is governed by a configuration file in Python's standard `configparser` format. The example configuration file *hesseflux_example.cfg* is highly commented. See the User Guide for a step by step guide through the script and the configuration file.

1.3 Installation

The easiest way to install is via *pip*:

```
pip install hesseflux
```

See the installation instructions for more information.

1.4 License

`hesseflux` is distributed under the MIT License. See the LICENSE file for details.

The project structure is based on a template provided by Sebastian Müller .

1.5 Contributing to hesseflux

Users are welcome to submit bug reports, feature requests, and code contributions to this project through GitHub.

More information is available in the Contributing guidelines.

1.6 Indices and tables

- genindex
- modindex

CHAPTER 2

USER GUIDE

`hesseflux` collects functions used for processing Eddy covariance data of the ICOS ecosystem site FR-Hes.

The package uses several functions of the JAMS Python package

https://github.com/mcuntz/jams_python

The JAMS package and hesseflux are synchronised irregularly.

`hesseflux` includes a Python port of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

The post-processing functionality for Eddy flux data is similar to the R-package REddyProc and includes basically the steps described in Papale et al. (Biogeosciences, 2006) plus some extensions such as the daytime method of flux partitioning (Lasslop et al., Global Change Biology 2010) and the estimation of uncertainties on the fluxes as in Lasslop et al. (Biogeosci, 2008).

Only the post-processing steps are described here. We are happy to discuss any processing or post-processing directly. Contact us at mc (at) macu (dot) de.

2.1 europe-fluxdata.eu file format

The first processing steps at the ICOS ecosystem site FR-Hes (not shown) brings the data in a format that can be submitted to the database europe-fluxdata.eu. The database predates ICOS and is somewhat a precursor of the ICOS data processing.

The file format of europe-fluxdata.eu is hence very similar to the ICOS format. The only known difference to us is the unit of atmospheric pressure, which is in hPa in europe-fluxdata.eu and in kPa in ICOS ecosystems. The file format has notably one header line with variable names. There are no units in the file. `hesseflux` provides a little helper script `europe-fluxdata_units.py` in the *bin* directory that adds a second header line with units. The script can be run on the output as:

```
python europe-fluxdata_units.py output_file_of_postproc_europe-fluxdata.csv
```

2.2 Post-processing Eddy covariance data

The script `postproc_europe-fluxdata.py` in the *example* directory provides a template for post-processing data that is in the europe-fluxdata.eu file format. It basically makes all steps described in Papale et al. (Biogeosciences, 2006). The script is governed by a configuration file in Python's standard `configparser`

format. The example configuration file `hesseflux_example.cfg` in the *example* directory is highly commented and should be (almost) self-explanatory. The script is called like:

```
python postproc_europe-fluxdata.py hesseflux_example.cfg
```

This script should be taken as a template for one's own post-processing but includes most standard post-processing steps.

Here we describe the main parts of the post-processing script.

Reading the configuration file

The script `postproc_europe-fluxdata.py` starts by reading the configuration file `hesseflux_example.cfg`:

```
import sys
import configparser

# Read config file
if len(sys.argv) <= 1:
    raise IOError('Input configuration file must be given.')
configfile = sys.argv[1]
config = configparser.ConfigParser(interpolation=None)
config.read(configfile)
```

It then analyses the configuration options. The first section in the configuration file are the options controlling which steps shall be performed by the script. The section in the `hesseflux_example.cfg` looks like:

```
[POSTSWITCH]
# spike detection (Papale et al., Biogeoci 2006)
# bool
outlier   = True
# ustar filtering (Papale et al., Biogeoci 2006)
# bool
ustar     = True
# flux partitioning (Reichstein et al., GCB 2005; Lasslop et al., GCB 2010)
# bool
partition = True
# gap filling (Reichstein et al., GCB 2005)
# bool
fill      = True
# error estimate of Eddy fluxes (Lasslop et al., Biogeosci 2008)
# bool
fluxerr   = False
```

And the code in `postproc_europe-fluxdata.py` is:

```
# program switches
outlier   = config['POSTSWITCH'].getboolean('outlier',   True)
ustar     = config['POSTSWITCH'].getboolean('ustar',     True)
partition = config['POSTSWITCH'].getboolean('partition', True)
fill      = config['POSTSWITCH'].getboolean('fill',      True)
fluxerr   = config['POSTSWITCH'].getboolean('fluxerr',   True)
```

All options are boolean and set to *True* by default if they are not given in the configuration file. All post-processing steps except uncertainty estimation of flux data would be performed in the given example.

Read the data

The script would then read in the data. The section in the configuration file is:

```
[POSTIO]
# can be comma separated list or single file
# str
inputfile  = FR-Hes_europe-fluxdata_2016.txt
# see strftime documentation of Python's datetime module
# https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior
# str
timeformat  = %Y%m%d%H%M
# Delimter to use with pandas.read_csv. If None, Python's builtin sniffer tool is␣
↪used (slow)
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
# str
sep         = ,
# Line numbers to skip (0-indexed) or number of lines to skip (int) at the start␣
↪of the file.
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
# list-like, int
skiprows    = None
# values being NaN and undef will be ignored
# float
undef       = -9999.
# threshold of shortwave radiation for determining day/night. day is SW_IN > swthr
# float
swthr       = 10.
```

The analysis of the options in `postproc_europe-fluxdata.py` is:

```
# input file
eufluxfile  = config['POSTIO'].get('inputfile',  '')
timeformat  = config['POSTIO'].get('timeformat', '%Y%m%d%H%M')
sep         = config['POSTIO'].get('sep',        ',')
skiprows    = config['POSTIO'].get('skiprows',   None)
undef       = config['POSTIO'].getfloat('undef', -9999.)
swthr       = config['POSTIO'].getfloat('swthr', 10.)
```

Note that strings are given without quotes in the configuration file.

eufluxfile can be a single filename or a comma-separated list of filenames. If it is missing or empty, the script will try to open a GUI, where one can choose input files. The data will be appended if several input files are given.

The (first) input file is read as:

```
import pandas as pd

parser = lambda date: pd.datetime.strptime(date, timeformat)
infile = eufluxfile[0]
df     = pd.read_csv(infile, sep, skiprows=skiprows, parse_dates=[0], date_
↪parser=parser, index_col=0, header=0)
```

`pandas` will use the first column as index (*index_col=0*), assuming that these are dates (*parse_dates=[0]*) in the format *timeformat*, where columns are separated by *sep*. The defaults follow the europe-fluxdata.eu format but similar formats may be used, and script and/or configuration file can be adapted easily. Only variable names have to follow europe-fluxdata.eu, ICOS or Ameriflux format at the moment. If the input file has a second header line with units, one can skip it giving *skiprows=[1]* (not *skiprows=1*).

All input files are supposed to be in the same format, if *eufluxfile* is a comma-separated list of filenames, and they will be read with the same command above. The `pandas` dataframes (*df*) will simply be appended.

The flag dataframe

All Not-a-Number (NaN) values will be set to *undef* and will be ignored in the following.

This happens via a second dataframe (*dff*), having the same columns and index as the input dataframe *df*, representing quality flags. All cells that have a value other than *0* in the flag dataframe *dff* will be ignored in the dataframe *df*. This means all cells of *df* with *undef* will be set to *2* in *dff* immediately:

```
# NaN -> undef
df.fillna(undef, inplace=True)

# Flag
dff               = df.copy(deep=True).astype(int)
dff[:]            = 0
dff[df == undef] = 2
```

Day / night

Most post-processing routines differentiate between daytime and nighttime data. Papale et al. (Biogeosciences, 2006) use a threshold of 20 W m^{-2} of global radiation to distinguish between day and night. REddyProc uses incoming shortwave radiation greater than 10 W m^2 as daytime. The shortwave raditan threshold *swthr* (same name as in ReddyProc) can be used to define the appropriate threshold. The default is 10 W m^2. The column *SW_IN_1_1_1* has to exist in the input data.

```
# day / night
isday = df['SW_IN_1_1_1'] > swthr
```

Data check

`postproc_europe-fluxdata.py` checks the units of air temperature (i.e. the first column starting with *TA_*).

```
# Check Ta in Kelvin
hta = ['TA_']
hout = _findfirststart(hta, df.columns)
if df[hout[0]].max() < 100.:
    tkelvin = 273.15
else:
    tkelvin = 0.
df.loc[dff[hout[0]]==0, hout[0]] += tkelvin
```

`_findfirststart(starts, names)()` is a helper function that finds the first occurrence in *names* that starts with the string *starts*. This is used for the moment until it was implemented in *hesseflux*' that the user can give individual variable names.

The script calculates air vapour pressure deficit *VPD_PI_1_1_1* from air temperature and relative humidity (i.e. the first column starting with *RH_*) if not given in input data using the function `esat()` of `hesseflux` for saturation vapour pressure:

```
import numpy as np
import hesseflux as hf

# add vpd if not given
hvpd = ['VPD']
hout = _findfirststart(hvpd, df.columns)
if len(hout) == 0:
    hvpd = ['TA_', 'RH_']
    hout = _findfirststart(hvpd, df.columns)
    if len(hout) != 2:
        raise ValueError('Cannot calculate VPD.')
    ta_id = hout[0]
    rh_id = hout[1]
```

(continues on next page)

(continued from previous page)

```
    if df[ta_id].max() < 100.:
        tk = df[ta_id] + 273.15
    else:
        tk = df[ta_id]
    if df[rh_id].max() > 10.:
        rh = df[rh_id] / 100.
    else:
        rh = df[rh_id]
    vpd = (1.-rh) * hf.esat(tk)
    vpd_id = 'VPD_PI_1_1_1'
    df[vpd_id] = vpd
    df[vpd_id].where((df[ta_id]!=undef) | (df[rh_id]!=undef), other=undef,␣
↪inplace=True)
    dff[vpd_id] = np.where((dff[ta_id] + dff[rh_id]) > 0, 2, 0)
    df.loc[dff[vpd_id]==0, vpd_id] /= 100. # hPa as in europe-fluxdata.eu
```

It further checks assures that VPD is in Pa for further calculations.

```
# Check VPD in Pa
 hvpd = ['VPD']
 hout = _findfirststart(hvpd, df.columns)
 if df[hout[0]].max() < 10.: # kPa
     vpdpa = 1000.
 elif df[hout[0]].max() < 100.: # hPa
     vpdpa = 100.
 else:
     vpdpa = 1.
 df.loc[dff[hout[0]]==0, hout[0]] *= vpdpa
```

And finally determines the time intervals of the input data *dtsec* (s) and the number of time steps per day *ntday*.

```
# time stepping
dsec  = (df.index[1] - df.index[0]).seconds
ntday = np.rint(86400/dsec).astype(np.int)
```

Spike / outlier flagging

If *outlier=True* is set in the configuration file, spikes will be detected with the method given in Papale et al. (Biogeosciences, 2006). A median absolute deviation (MAD) filter will be used on the second derivatives of the time series in two-week chunks. The section in `hesseflux_example.cfg` looks like:

```
[POSTMAD]
# spike / outlier detection, see help(hesseflux.madspikes)
# scan window in days for spike detection
# int
nscan = 15
# fill window in days for spike detection
# int
nfill = 1
# spike will be set for values above z absolute deviations
# float
z     = 7.
# 0: mad on raw values; 1, 2: mad on first or second derivatives
# int
deriv = 2
```

nfill is the number of days that are treated at once. *nfill=1* means that the time series will be stepped through day by day. *nscan* are the days to be considered to calculate the mean absolute deviations. *nscan=15* means that 7 days before the fill day, the fill day itself and 7 days after the fill day will be used for the robust statistic. However, only spikes detected within the inner *nfill* days will be flagged in the *nscan* days. Spikes will be detected if they

deviate more than *z* mean absolute deviations from the median. *deriv=2* applies the MAD filter to the second derivatives. A spike has normally a strong curvature and hence a large second derivative. *deriv=1* is currently not implemented. *deriv=0* applies the filter to the raw time series. This might be useful to find outliers in smooth time series such as soil moisture. *deriv=0* is also used on the 20 Hz Eddy raw data in the quality and uncertainty strategy of Mauder et al. (Agric Forest Meteo, 2013).

The default values, if option are not given in the configuration file, are *nscan=15*, *nfill=1*, *z=7*, and *deriv=2*.

`postproc_europe-fluxdata.py` calls the spike detection like this:

```
houtlier = ['H_', 'LE', 'FC',       # assume *_PI variables after raw variables, e.
↪g. LE before LE_PI
            'H_PI', 'LE_PI', 'NEE'] # if available
hout = _findfirststart(houtlier, df.columns)
sflag = hf.madspikes(df[hout], flag=dff[hout], isday=isday, undef=undef,
                     nscan=nscan*ntday, nfill=nfill*ntday, z=z, deriv=deriv,
                     plot=False)
for ii, hh in enumerate(hout):
    dff.loc[sflag[hh]==2, hh] = 3
```

The function `madspikes()` returns flag columns for the input variables where spiked data is flagged as 2. The scripts sets the corresponding columns in the flag dataframe *dff* to 3 (3 just to keep track where the flag was set).

u* filtering

If *ustar=True* is set in the configuration file, a u*-filter will be applied following Papale et al. (Biogeosciences, 2006).

The section in `hesseflux_example.cfg` looks like:

```
[POSTUSTAR]
# ustar filtering, see help(hesseflux.ustarfilter)
# min ustar value. Papale et al. (Biogeosci 2006): 0.1 forest, 0.01 else
# float
ustarmin    = 0.1
# number of boostraps for determining uncertainty of ustar threshold. 1 = no␣
↪bootstrap
# int
nboot       = 1
# significant difference between ustar class and mean of classes above
# float
plateaucrit = 0.95
```

A minimum threhold *ustarmin* is defined under which data is flagged by default. Papale et al. (Biogeosciences, 2006) suggest 0.1 for forests and 0.01 for other land cover types. `postproc_europe-fluxdata.py` sets 0.01 as its default value. Uncertainty of the u*-threshold is calculated via bootstraping in Papale et al. *nboot* gives the number of bootstrepping for an the uncertainty estimate of the u*-threshold. The algorithm divides the input data (per season) in 7 temperature classes and in 20 u*-classes within each temperature class. It then determines the threshold as the average u* of the u*-class where the average CO2 flux is less than *plateaucrit* times the average of all CO2 fluxes with u* greater than the u*-class. Papale et al. (Biogeosciences, 2006) took *plateaucrit=0.99*, while REddyProc takes *plateaucrit=0.95*, which `postproc_europe-fluxdata.py` also takes as its default.

The u*-filtering is then performed as:

```
# The algorithm uses NEE to determine u*-threshold
hfilt = ['NEE', 'USTAR', 'TA_']
hout = _findfirststart(hfilt, df.columns)
# the algorithm does not work with carbon uptake at night -> flag it temporarily
ffsave = dff[hout[0]].to_numpy()
iic    = np.where((~isday) & (df[hout[0]] < 0.))[0]
```

(continues on next page)

(continued from previous page)

```
dff.iloc[iic, list(df.columns).index(hout[0])] = 4
ustars, flag = hf.ustarfilter(df[hout], flag=dff[hout], isday=isday, undef=undef,
                              ustarmin=ustarmin, nboot=nboot,␣
↪plateaucrit=plateaucrit,
                              plot=True)
dff[hout[0]] = ffsave # set NEE<0 @ night flags back
# The threshold is then applied to all eddy fluxes
hustar = ['H_', 'LE', 'FC',       # assume *_PI variables after raw variables, e.g.
↪ LE before LE_PI
          'H_PI', 'LE_PI', 'NEE'] # if available
hout = _findfirststart(hustar, df.columns)
for ii, hh in enumerate(hout):
    dff.loc[flag==2, hh] = 5
```

The function `ustarfilter()` returns the ustar 5, 50 and 95 percentile of the bootstrapped u*-thresholds and a flag columns, which is 0 except where u* is smaller than the median u*-threshold. The scripts sets the columns of the Eddy fluxes in the flag dataframe *dff* to 5 (5 just to keep track where the flag was set).

One might not want to do u*-filtering, but use for example Integral Turbulence Characteristics (ITC) that were calculated, for example, with EddyPro(R). These should be set right at the start after reading the input data into the dataframe *df* and producing the flag dataframe *dff* like:

```
dff.loc[df['FC_SSITC_TEST_1_1_1']>0, 'FC_1_1_1'] = 2
```

Partitioning of Net Ecosystem Exchange

If *partition=True* is set in the configuration file, two estimates of Gross Primary Productivity (GPP) and Ecosystem Respiration (RECO) are calculated: firstly with the method of Reichstein et al. (Glob Change Biolo, 2005) using nighttime data only, and secondly with the method of Lasslop et al. (Glob Change Biolo, 2010) using a light-response curve on 'daytime' data. The configuration `hesseflux_example.cfg` gives only one option in this section:

```
[POSTPARTITION]
# partitioning, see help(hesseflux.nee2gpp)
# if True, set GPP=0 at night
# bool
nogppnight = False
```

Many people find it unaesthetic that the 'daytime' method gives negative GPP at night. We esteem this the correct behaviour, reflecting the uncertainty in the gross flux estimates. However, one can set *nogppnight=True* to set GPP=0 at night and RECO=NEE in this case, the latter having then all variability of the net fluxes.

The partitioning is calculated as:

```
hpart = ['NEE', 'SW_IN', 'TA_', 'VPD']
hout  = _findfirststart(hpart, df.columns)
# nighttime method
dfpartn = hf.nee2gpp(df[hout], flag=dff[hout], isday=isday,
                     undef=undef, method='reichstein', nogppnight=nogppnight)
suff = hout[0][3:-1]
dfpartn.rename(columns=lambda c: c+suff+'1', inplace=True)

# daytime method
dfpartd = hf.nee2gpp(df[hout], flag=dff[hout], isday=isday,
                     undef=undef, method='lasslop', nogppnight=nogppnight)
dfpartd.rename(columns=lambda c: c+suff+'2', inplace=True)

# add new columns to dataframe
df = pd.concat([df, dfpartn, dfpartd],  axis=1)
```

(continues on next page)

(continued from previous page)

```
# take flags from NEE for the new columns
for dn in ['1', '2']:
    for gg in ['GPP', 'RECO']:
        dff[gg+suff+dn] = dff[hout[0]]
```

Gap-filling / Imputation

Marginal Distribution Sampling (MDS) of Reichstein et al. (Glob Change Biolo, 2005) is implemented as imputation or called gap-filling algorithm. The algorithm looks for similar conditions in the vicinity of a missing data point, if option *fill=True*. The configuration file is:

```
[POSTGAP]
# gap-filling with MDS, see help(hesseflux.gapfill)
# max deviation of SW_IN
# float
sw_dev  = 50.
# max deviation of TA
# float
ta_dev  = 2.5
# max deviation of VPD
# float
vpd_dev = 5.0
# avoid extrapolation in gaps longer than longgap days
longgap = 60
```

If a flux data point is missing, times with incoming shortwave radiation in the range of *sw_dev* around the actual shortwave radiation will be looked for, as well as air temperatures within *ta_dev* and air vapour pressure deficit within *vpd_dev*. The function does not fill long gaps longer than *longgap* days. A good summary is given in Fig. A1 of Reichstein et al. (Glob Change Biolo, 2005).

The script invokes MDS as:

```
hfill = ['H_', 'LE', 'FC',       # assume *_PI variables after raw variables, e.g.␣
↪LE before LE_PI
         'H_PI', 'LE_PI', 'NEE', # if available
         'GPP_1_1_1', 'RECO_1_1_1', 'GPP_1_1_2', 'RECO_1_1_2',
         'GPP_PI_1_1_1', 'RECO_PI_1_1_1', 'GPP_PI_1_1_2', 'RECO_PI_1_1_2',
         'SW_IN', 'TA_', 'VPD']
hout  = _findfirststart(hfill, df.columns)
df_f, dff_f = hf.gapfill(df[hout], flag=dff[hout],
                         sw_dev=sw_dev, ta_dev=ta_dev, vpd_dev=vpd_dev,
                         longgap=longgap, undef=undef, err=False, verbose=1)
# remove meteorology columns
hdrop = ['SW_IN', 'TA_', 'VPD']
hout = _findfirststart(hdrop, df.columns)
df_f.drop(columns=hout,  inplace=True)
dff_f.drop(columns=hout, inplace=True)
# we add _f to columns names of filled variables
df_f.rename(columns=lambda c: '_'.join(c.split('_')[:-3]+['f']+c.split('_')[-3:]),␣
↪ inplace=True)
dff_f.rename(columns=lambda c: '_'.join(c.split('_')[:-3]+['f']+c.split('_')[-3:]),
↪ inplace=True)
df  = pd.concat([df,  df_f],  axis=1)
dff = pd.concat([dff, dff_f], axis=1)
```

The function `gapfill()` returns the filled columns *df_f* as well as flag columns *dff_f* indicating fill quality. Fill quality A-C of Reichstein et al. are translated to quality flags 1-3.

Uncertainty estimates of flux data

Lasslop et al. (Biogeosci, 2008) presented an algorithm to estimate uncertainties of Eddy covariance fluxes using Marginal Distribution Sampling (MDS). The gap-filling function `gapfill()` can be used for uncertainty estimation giving the keyword *err=True*. The same thresholds as for gap-filling are used.

The script `postproc_europe-fluxdata.py` uses the function `gapfill()` to calculate flux uncertainties like:

```
hfill = ['H_', 'LE', 'FC',       # assume *_PI variables after raw variables, e.g. LE before LE_PI
         'H_PI', 'LE_PI', 'NEE', # if available
         'H_f', 'LE_f', 'FC_f',
         'H_PI_f', 'LE_PI_f', 'NEE_f', 'NEE_PI_f',
         'GPP_1_1_1', 'RECO_1_1_1', 'GPP_1_1_2', 'RECO_1_1_2',
         'GPP_f_1_1_1', 'RECO_f_1_1_1', 'GPP_f_1_1_2', 'RECO_f_1_1_2',
         'GPP_PI_1_1_1', 'RECO_PI_1_1_1', 'GPP_PI_1_1_2', 'RECO_PI_1_1_2',
         'GPP_PI_f_1_1_1', 'RECO_PI_f_1_1_1', 'GPP_PI_f_1_1_2', 'RECO_PI_f_1_1_2',
         'SW_IN', 'TA_', 'VPD']
hout = _findfirststart(hfill, df.columns)
df_f = hf.gapfill(df[hout], flag=dff[hout],
                  sw_dev=sw_dev, ta_dev=ta_dev, vpd_dev=vpd_dev,
                  longgap=longgap, undef=undef, err=True, verbose=1)
# remove meteorology columns
hdrop = ['SW_IN', 'TA_', 'VPD']
hout = _findfirststart(hdrop, df.columns)
df_f.drop(columns=hout, inplace=True)
colin = list(df_f.columns)
# we add _err to columns names of uncertainty estimates, such as: NEE_PI_err_1_1_1
df_f.rename(columns=lambda c: '_'.join(c.split('_')[:-3]+['err']+c.split('_')[-3:]),  inplace=True)
colout = list(df_f.columns)
df = pd.concat([df, df_f], axis=1)
# take flags of non-error columns
for cc in range(len(colin)):
    dff[colout[cc]] = dff[colin[cc]]
```

We recommend, however, to calculate flux uncertainties with the Eddy covariance raw data as described in Mauder et al. (Agric Forest Meteo, 2013). This is, for example, implemented in the processing software EddyPro(R).

Writing the output file

The dataframe is written to the output file with `pandas` `pandas.Dataframe.to_csv()`:

```
df.to_csv(outputfile, sep=sep, na_rep=str(undef), index=True, date_format=timeformat)
```

using the same *sep* and *timeformat* as the input.

The configuration for output is:

```
[POSTIO]
# if empty, write will ask for output path using the name of this config file with the suffix .csv
outputfile  = FR-Hes_europe-fluxdata_2016-post.txt
# if True, set variable to undef where flagged in output
# bool
outundef    = True
# if True, add flag columns prepended with flag_ for each variable
# bool
outflagcols = False
```

If *outputfile* is missing or empty, the script will try to open a GUI, where one can choose an output directory and the filename will then be name of the configuration file with the suffix '.csv'.

If *outundef=True* then all values in *df* with a flag value in *dff* greater than zero will be set to *undef*. The script can also add flag columns, prefixed with *flag_*, for each column in *df*, if *outflagcols=True*. The script will always output the columns with the flags for fill quality, if gap-filling was performed: option *fill=True*.

The code before `pandas.Dataframe.to_csv()` is then:

```
if outundef:
    for cc in df.columns:
        if cc.split('_')[-4] != 'f': # exclude gap-filled columns
            df[cc].where(dff[cc] == 0, other=undef, inplace=True)
if outflagcols:
    dff.rename(columns=lambda c: 'flag_'+c, inplace=True)
    df = pd.concat([df, dff], axis=1)
else:
    occ = []
    for cc in df.columns:
        if cc.split('_')[-4] == 'f': occ.append(cc)
    dff1 = dff[occ].copy(deep=True)
    dff1.rename(columns=lambda c: 'flag_'+c, inplace=True)
    df = pd.concat([df, dff1], axis=1)
```

That's all Folks!

CHAPTER 3

INSTALLATION

The easiest way to install `hesseflux` is via `pip`:

```
pip install hesseflux
```

3.1 Manual install

The latest version of `hesseflux` can be installed from source:

```
git clone https://github.com/mcuntz/hesseflux.git
cd hesseflux
pip install .
```

3.2 Local install

Users without proper privileges can append the *–user* flag to `pip` either while installing from the Python Package Index (PyPI):

```
pip install hesseflux --user
```

or from the top `hesseflux` directory:

```
git clone https://github.com/mcuntz/hesseflux.git
cd hesseflux
pip install . --user
```

If `pip` is not available, then *setup.py* can still be used:

```
python setup.py install --user
```

When using *setup.py* locally, it might be that one needs to append *–prefix=* to the command:

```
python setup.py install --user --prefix=
```

3.3 Dependencies

`hesseflux` uses the packages `numpy`, `scipy` and `pandas`. They are all available in PyPI and `pip` should install them automatically. Installations via *setup.py* might need to install the three dependencies first.

CHAPTER 4

API

4.1 Purpose

hesseflux provides functions used in the processing and post-processing of the Eddy covariance flux data of the ICOS ecosystem site FR-Hes.

The package uses several functions of the JAMS Python package https://github.com/mcuntz/jams_python The JAMS package and hesseflux are synchronised irregularly.

hesseflux includes a Python port of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

The post-processing functionality for Eddy flux data is similar to the R-package REddyProc and includes basically the steps described in Papale et al. (Biogeosciences, 2006) plus some extensions such as the daytime method of flux partitioning (Lasslop et al., Global Change Biology 2010).

4.2 Subpackages

const	Purpose
functions	Purpose
argsort(a, *args, **kwargs)	Wrapper for numpy.argsort, numpy.ma.argsort, and using sorted for Python iterables.
ascii2ascii(edate[, full, en, fr, us, eng, YY])	Convert date notations between ascii DD.MM.YYYY hh:mm:ss, English YYYY-MM-DD hh:mm:ss, American MM/DD/YYYY hh:mm:ss, and French DD/MM/YYYY hh:mm:ss.
date2dec([calendar, units, excelerr, yr, ...])	Convert scalar and array_like with calendar dates into decimal dates.
dec2date(indata[, calendar, refdate, units, ...])	Converts scale and array_like with decimal dates into calendar dates.
division(a, b[, otherwise, prec])	Divide two arrays, return *otherwise* if division by 0.
esat(T[, liquid, formula])	Calculates the saturation vapour pressure of water and/or ice.
fgui	fgui : GUI dialogs to choose files and directories using Tkinter.

Continued on next page

Table 1 – continued from previous page

fread(infile[, nc, cname, skip, cskip, ...])	Read numbers into array with floats from a file.
fsread(infile[, nc, cname, snc, sname, ...])	Read from a file numbers into 2D float array as well as characters into 2D string array.
gapfill(dfin[, flag, date, timeformat, ...])	Fills gaps in flux data from Eddy covariance measurements with Marginal Distribution Sampling (MDS) according to Reichstein et al.
mad(datin[, z, deriv, nozero])	Median absolute deviation test, either on raw values, or on 1st or 2nd derivatives.
madspikes(dfin[, flag, isday, colhead, ...])	Spike detection for using a moving median absolute difference filter.
nee2gpp(dfin[, flag, isday, date, ...])	Calculate photosynthesis (GPP) and ecosystem respiration (RECO) from Eddy covariance CO2 flux data.
sread(infile[, nc, cname, skip, cskip, ...])	Read strings into string array from a file.
ustarfilter(dfin[, flag, isday, date, ...])	Flag Eddy Covariance data using a threshold of friction velocity (u*) below which u* correlates with a reduction in CO2 flux.
logtools	Purpose

4.3 hesseflux.argsort

argsort : argmax, argmin and argsort for array_like and Python iterables.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Dec 2014 by Matthias Cuntz (mc (at) macu (dot) de)
- Added argmin, argmax, Jul 2019, Matthias Cuntz
- Using numpy docstring format, extending examples from numpy docstrings, May 2020, Matthias Cuntz

The following functions are provided

argmax(a, *args, **kwargs)	Wrapper for numpy.argmax, numpy.ma.argmax, and using max for Python iterables.
argmin(a, *args, **kwargs)	Wrapper for numpy.argmin, numpy.ma.argmin, and using min for Python iterables.
argsort(a, *args, **kwargs)	Wrapper for numpy.argsort, numpy.ma.argsort, and using sorted for Python iterables.

argmax(*a*, **args*, ***kwargs*)

Wrapper for numpy.argmax, numpy.ma.argmax, and using max for Python iterables.

Passes all keywords directly to the individual routines, i.e.

numpy.argmax(a, axis=None, out=None)

numpy.ma.argmax(self, axis=None, fill_value=None, out=None)

No keyword will be passed to max routine for Python iterables.

Parameters

- **a** (*array_like*) – input array, masked array, or Python iterable
- ***args** (*optional*) – all arguments of numpy.argmax or numpy.ma.argmax
- ****kwargs** (*optional*) – all keyword arguments of numpy.argmax or numpy.ma.argmax

Returns **index_array** – Array of indices of the largest element in input array *a*. It has the same shape as *a.shape* with the dimension along *axis* removed. a[np.unravel_index(argmax(a), a.shape)] is the maximum value of a.

Return type ndarray, int

Examples

```
>>> import numpy as np
```

One-dimensional array >>> a = np.array([0,4,6,2,1,5,3,5]) >>> ii = argmax(a) >>> print(ii) 2 >>> print(a[ii]) 6

One-dimensional masked array >>> a = np.ma.array([0,4,6,2,1,5,3,5], mask=[0,0,1,1,0,0,0,0]) >>> ii = argmax(a) >>> print(ii) 5 >>> print(a[ii]) 5 >>> ii = argmax(a, fill_value=6) >>> print(ii) 2

List >>> a = [0,4,6,2,1,5,3,5] >>> ii = argmax(a) >>> print(ii) 2 >>> print(a[ii]) 6

```
>>> # from numpy.argmax docstring
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

Indexes of the maximal elements of a N-dimensional array: >>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape) >>> ind (1, 2) >>> a[ind] 15

```
>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b)  # Only the first occurrence is returned.
1
```

Notes

argmax for iterables was taken from https://stackoverflow.com/questions/16945518/finding-the-index-of-the-value-which-is-the-min-or-max-in-python

argmin(*a*, **args*, ***kwargs*)

Wrapper for numpy.argmin, numpy.ma.argmin, and using min for Python iterables.

Passes all keywords directly to the individual routines, i.e. numpy.argmin(a, axis=None, out=None)

numpy.ma.argmin(self, axis=None, fill_value=None, out=None)

No keyword will be passed to min routine for Python iterables.

Parameters

- **a** (*array_like*) – input array, masked array, or Python iterable
- ***args** (*optional*) – all arguments of numpy.argmin or numpy.ma.argmin
- ****kwargs** (*optional*) – all keyword arguments of numpy.argmin or numpy.ma.argmin

Returns **index_array** – Array of indices of the largest element in input array *a*. It has the same shape as *a.shape* with the dimension along *axis* removed. a[np.unravel_index(argmin(a), a.shape)] is the minimum value of a.

Return type ndarray, int

Examples

```
>>> import numpy as np
```

One-dimensional array >>> a = np.array([0,4,6,2,1,5,3,5]) >>> ii = argmin(a) >>> print(ii) 0 >>> print(a[ii]) 0

One-dimensional masked array >>> a = np.ma.array([0,4,6,2,1,5,3,5], mask=[1,0,1,1,0,0,0,0]) >>> ii = argmin(a) >>> print(ii) 4 >>> print(a[ii]) 1 >>> ii = argmin(a, fill_value=1) >>> print(ii) 0

List >>> a = [0,4,6,2,1,5,3,5] >>> ii = argmin(a) >>> print(ii) 0 >>> print(a[ii]) 0

```
>>> # from numpy.argmin docstring
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmin(a)
0
>>> np.argmin(a, axis=0)
array([0, 0, 0])
>>> np.argmin(a, axis=1)
array([0, 0])
```

Indices of the minimum elements of a N-dimensional array: >>> ind = np.unravel_index(np.argmin(a, axis=None), a.shape) >>> ind (0, 0) >>> a[ind] 10

```
>>> b = np.arange(6) + 10
>>> b[4] = 10
>>> b
array([10, 11, 12, 13, 10, 15])
>>> np.argmin(b)  # Only the first occurrence is returned.
0
```

Notes

argmin for iterables was taken from https://stackoverflow.com/questions/16945518/finding-the-index-of-the-value-which-is-the-min-or-max-in-python

argsort (*a*, **args*, ***kwargs*)

Wrapper for numpy.argsort, numpy.ma.argsort, and using sorted for Python iterables.

Passes all keywords directly to the individual routines, i.e. numpy.argsort(a, axis=-1, kind='quicksort', order=None) numpy.ma.argsort(a, axis=None, kind='quicksort', order=None, fill_value=None) sorted(iterable[, cmp[, key[, reverse]]])

Only key cannot be given for Python iterables because the input array is used as key in the sorted function.

Parameters

- **a** (*array_like*) – input array, masked array, or Python iterable
- ***args** (*optional*) – all arguments of numpy.argsort, numpy.ma.argsort, and sorted (except key argument)
- ****kwargs** (*optional*) – all keyword arguments of numpy.argsort, numpy.ma.argsort, and sorted (except key argument)

Returns **index_array** – Array of indices that sort *a* along the specified *axis*. If *a* is one-dimensional, `a[index_array]` yields a sorted *a*.

Return type ndarray, int

Examples

```
>>> import numpy as np
```

1D array >>> a = np.array([0,4,6,2,1,5,3,5]) >>> ii = argsort(a) >>> print(a[ii]) [0 1 2 3 4 5 5 6]

```
>>> ii = argsort(a, kind='quicksort')
>>> print(a[ii])
[0 1 2 3 4 5 5 6]
```

1D masked array >>> a = np.ma.array([0,4,6,2,1,5,3,5], mask=[0,0,1,1,0,0,0,0]) >>> ii = argsort(a) >>> print(a[ii]) [0 1 3 4 5 5 – –]

```
>>> ii = argsort(a, fill_value=1)
>>> print(a[ii])
[0 -- -- 1 3 4 5 5]
```

list >>> a = [0,4,6,2,1,5,3,5] >>> ii = argsort(a) >>> b = [a[i] for i in ii] >>> print(b) [0, 1, 2, 3, 4, 5, 5, 6]

```
>>> a = [0,4,6,2,1,5,3,5]
>>> ii = argsort(a, reverse=True)
>>> b = [ a[i] for i in ii ]
>>> print(b)
[6, 5, 5, 4, 3, 2, 1, 0]
```

from numpy.argsort docstring # One-dimensional array: >>> x = np.array([3, 1, 2]) >>> np.argsort(x) array([1, 2, 0])

```
>>> # Two-dimensional array:
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])
>>> ind = np.argsort(x, axis=0)  # sorts along first axis (down)
>>> ind
array([[0, 1],
       [1, 0]])
>>> np.take_along_axis(x, ind, axis=0)  # same as np.sort(x, axis=0)
array([[0, 2],
       [2, 3]])
>>> ind = np.argsort(x, axis=1)  # sorts along last axis (across)
>>> ind
array([[0, 1],
       [0, 1]])
>>> np.take_along_axis(x, ind, axis=1)  # same as np.sort(x, axis=1)
array([[0, 3],
       [2, 2]])
```

Indices of the sorted elements of a N-dimensional array: >>> ind = np.unravel_index(np.argsort(x, axis=None), x.shape) >>> ind (array([0, 1, 1, 0]), array([0, 0, 1, 1])) >>> x[ind] # same as np.sort(x, axis=None) array([0, 2, 2, 3])

```
>>> # Sorting with keys:
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])
>>> np.argsort(x, order=('x','y'))
array([1, 0])
>>> np.argsort(x, order=('y','x'))
array([0, 1])
```

Notes

argsort for iterables was taken from http://stackoverflow.com/questions/3382352/equivalent-of-numpy-argsort-in-basic-python http://stackoverflow.com/questions/3071415/efficient-method-to-calculate-the-rank-vector-of-a-list-in-python

4.4 hesseflux.ascii2ascii

ascii2ascii [Convert date notations between different regional variants] such as English YYYY-MM-DD hh:mm:ss and US-English MM/DD/YYYY hh:mm:ss formats.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Feb 2015 by Matthias Cuntz (mc (at) macu (dot) de)
- Removed usage of date2dec and dec2date, Nov 2016, Matthias Cuntz
- Adapted docstrings to Python 2 and 3, Nov 2016, Matthias Cuntz
- Added us and fr keywords, renamed eng to en, Mar 2018, Matthias Cuntz
- Added two-digit year, Nov 2018, Matthias Cuntz
- Removed bug from usage of en and old name eng, Jun 2019, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

ascii2ascii(edate[, full, en, fr, us, eng, YY])	Convert date notations between ascii DD.MM.YYYY hh:mm:ss, English YYYY-MM-DD hh:mm:ss, American MM/DD/YYYY hh:mm:ss, and French DD/MM/YYYY hh:mm:ss.
ascii2en(edate, **kwarg)	Wrapper function for *ascii2ascii()* with English date format output, i.e. *en=True*:
ascii2fr(edate, **kwarg)	Wrapper function for *ascii2ascii()* with French date format output, i.e. *fr=True*:
ascii2us(edate, **kwarg)	Wrapper function for *ascii2ascii()* with US-English date format output, i.e. *us=True*:
ascii2eng(edate, **kwarg)	Wrapper function for *ascii2ascii()* with English date format output, i.e. *en=True*:
en2ascii(edate, **kwarg)	Wrapper function for ascii2ascii with ascii date format output (default):
fr2ascii(edate[, full, YY])	Convert French date notation DD/MM/YYYY hh:mm:ss to ascii notation DD.MM.YYYY hh:mm:ss.
us2ascii(edate, **kwarg)	Wrapper function for *ascii2ascii()* with ascii date format output (default):
eng2ascii(edate, **kwarg)	Wrapper function for *ascii2ascii()* with ascii date format output (default):

ascii2ascii(*edate*, *full=False*, *en=False*, *fr=False*, *us=False*, *eng=False*, *YY=False*)

Convert date notations between ascii DD.MM.YYYY hh:mm:ss, English YYYY-MM-DD hh:mm:ss, American MM/DD/YYYY hh:mm:ss, and French DD/MM/YYYY hh:mm:ss. Input can only be ascii, English and American. Output can also be French DD/MM/YYYY hh:mm:ss. Use fr2ascii first for French input formats.

Parameters

- **edate** (*array_like*) – date strings in ascii, English or American format.
- **full** (*bool, optional*) – True: output dates arr all in full format DD.MM.YYYY hh:mm:ss; missing time inputs are 00 on output

  False: output dates are as long as input dates (default), e.g. [YYYY-MM-DD,

YYYY-MM-DD hh:mm] gives [DD.MM.YYYY, DD.MM.YYYY hh:mm]

- **en** (*bool, optional*) – True: output format is English YYYY-MM-DD hh:mm:ss (default: False)
- **fr** (*bool, optional*) – True: output format is French DD/MM/YYYY hh:mm:ss (default: False)
- **us** (*bool, optional*) – True: output format is American MM/DD/YYYY hh:mm:ss (default: False)
- **YY** (*bool, optional*) – Year in input file is 2-digit year. Every year that is above the current year will be taken as being in 1900 (default: False).
- **eng** (*bool, optional*) – Same as en: obsolete.

Returns **date** – date strings in chosen date format. If no optional keyword True then output is in ascii format: DD.MM.YYYY hh:mm:ss. The output type will be the same as the type of the input.

Return type array_like

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> print(", ".join(ascii2ascii(edate)))
12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990, 04.05.1786
```

```
>>> print(", ".join(ascii2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00, 04.05.1786␣
↪00:00:00
```

```
>>> print(", ".join(ascii2ascii(edate, en=True)))
2014-11-12 12:00, 2015-03-01 17:56:00, 1990-12-01, 1786-05-04
```

```
>>> print(", ".join(ascii2ascii(edate, en=True, full=True)))
2014-11-12 12:00:00, 2015-03-01 17:56:00, 1990-12-01 00:00:00, 1786-05-04␣
↪00:00:00
```

```
>>> print(ascii2ascii(list(edate)))
['12.11.2014 12:00', '01.03.2015 17:56:00', '01.12.1990', '04.05.1786']
```

```
>>> print(ascii2ascii(tuple(edate)))
('12.11.2014 12:00', '01.03.2015 17:56:00', '01.12.1990', '04.05.1786')
```

```
>>> print(ascii2ascii(np.array(edate)))
['12.11.2014 12:00' '01.03.2015 17:56:00' '01.12.1990' '04.05.1786']
```

```
>>> print(ascii2ascii(edate[0]))
12.11.2014 12:00
```

```
>>> print(", ".join(ascii2ascii(edate, us=True)))
11/12/2014 12:00, 03/01/2015 17:56:00, 12/01/1990, 05/04/1786
```

```
>>> print(", ".join(ascii2ascii(ascii2ascii(edate, en=True), us=True,␣
↪full=True)))
11/12/2014 12:00:00, 03/01/2015 17:56:00, 12/01/1990 00:00:00, 05/04/1786␣
↪00:00:00
```

```
>>> print(", ".join(ascii2ascii(edate, fr=True)))
12/11/2014 12:00, 01/03/2015 17:56:00, 01/12/1990, 04/05/1786
```

```
>>> print(", ".join(ascii2ascii(edate, fr=True, full=True)))
12/11/2014 12:00:00, 01/03/2015 17:56:00, 01/12/1990 00:00:00, 04/05/1786␣
↪00:00:00
```

YY=True >>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01'] >>> print(", ".join(ascii2ascii(edate, YY=True))) 12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990 >>> print(", ".join(ascii2ascii(edate, en=True, YY=True))) 2014-11-12 12:00, 2015-03-01 17:56:00, 1990-12-01 >>> print(", ".join(ascii2ascii(edate, us=True, YY=True))) 11/12/2014 12:00, 03/01/2015 17:56:00, 12/01/1990 >>> print(", ".join(ascii2ascii(ascii2ascii(edate, en=True, YY=True), us=True, full=True))) 11/12/2014 12:00:00, 03/01/2015 17:56:00, 12/01/1990 00:00:00 >>> print(", ".join(ascii2ascii(edate, fr=True, full=True, YY=True))) 12/11/2014 12:00:00, 01/03/2015 17:56:00, 01/12/1990 00:00:00

ascii2en(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with English date format output, i.e. *en=True*:
*ascii2ascii(edate, en=True, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> print(", ".join(ascii2en(edate)))
2014-11-12 12:00, 2015-03-01 17:56:00, 1990-12-01, 1786-05-04
```

```
>>> print(", ".join(ascii2en(edate, full=True)))
2014-11-12 12:00:00, 2015-03-01 17:56:00, 1990-12-01 00:00:00, 1786-05-04␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> print(", ".join(ascii2en(edate, full=True, YY=True)))
2014-11-12 12:00:00, 2015-03-01 17:56:00, 1990-12-01 00:00:00
```

ascii2fr(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with French date format output, i.e. *fr=True*:
*ascii2ascii(edate, fr=True, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> print(", ".join(ascii2fr(edate)))
12/11/2014 12:00, 01/03/2015 17:56:00, 01/12/1990, 04/05/1786
```

```
>>> print(", ".join(ascii2fr(edate, full=True)))
12/11/2014 12:00:00, 01/03/2015 17:56:00, 01/12/1990 00:00:00, 04/05/1786␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> print(", ".join(ascii2fr(edate, full=True, YY=True)))
12/11/2014 12:00:00, 01/03/2015 17:56:00, 01/12/1990 00:00:00
```

ascii2us(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with US-English date format output, i.e. *us=True*:
*ascii2ascii(edate, us=True, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> print(", ".join(ascii2ascii(edate, us=True)))
11/12/2014 12:00, 03/01/2015 17:56:00, 12/01/1990, 05/04/1786
```

```
>>> print(", ".join(ascii2ascii(ascii2ascii(edate, en=True), us=True,␣
↪full=True)))
11/12/2014 12:00:00, 03/01/2015 17:56:00, 12/01/1990 00:00:00, 05/04/1786␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> print(", ".join(ascii2ascii(ascii2ascii(edate, en=True, YY=True), us=True,␣
↪full=True)))
11/12/2014 12:00:00, 03/01/2015 17:56:00, 12/01/1990 00:00:00
```

ascii2eng(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with English date format output, i.e. *en=True*:
*ascii2ascii(edate, en=True, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> print(", ".join(ascii2eng(edate)))
2014-11-12 12:00, 2015-03-01 17:56:00, 1990-12-01, 1786-05-04
```

```
>>> print(", ".join(ascii2eng(edate, full=True)))
2014-11-12 12:00:00, 2015-03-01 17:56:00, 1990-12-01 00:00:00, 1786-05-04␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> print(", ".join(ascii2eng(edate, full=True, YY=True)))
2014-11-12 12:00:00, 2015-03-01 17:56:00, 1990-12-01 00:00:00
```

en2ascii(*edate*, ***kwarg*)

Wrapper function for ascii2ascii with ascii date format output (default): *ascii2ascii(edate, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> edate = ascii2ascii(edate, en=True)
>>> print(", ".join(en2ascii(edate)))
12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990, 04.05.1786
```

```
>>> print(", ".join(en2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00, 04.05.1786␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> edate = ascii2ascii(edate, en=True, YY=True)
>>> print(", ".join(en2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00
```

fr2ascii(*edate*, *full=False*, *YY=False*)

Convert French date notation DD/MM/YYYY hh:mm:ss to ascii notation DD.MM.YYYY hh:mm:ss. Simply replaces '/' with '.', assuring iterable type.

Parameters

- **edate** (*array_like*) – date strings in French date format DD/MM/YYYY [hh:mm:ss]
- **full** (*bool, optional*) – True: output dates arr all in full format DD.MM.YYYY hh:mm:ss; missing time inputs are 00 on output

  False: output dates are as long as input dates (default), e.g. [DD/MM/YYYY, DD/MM/YYYY hh:mm] gives [DD.MM.YYYY, DD.MM.YYYY hh:mm]
- **YY** (*bool, optional*) – Year in input file is 2-digit year. Every year that is above the current year will be taken as being in 1900 (default: False).

Returns **date** – date strings in ascii date format: DD.MM.YYYY hh:mm:ss. The output type will be the same as the type of the input.

Return type array_like

Examples

```
>>> edate = ['12/11/2014 12:00', '01/03/2015 17:56:00', '01/12/1990', '04/05/
↪1786']
>>> print(", ".join(fr2ascii(edate)))
12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990, 04.05.1786
```

```
>>> print(", ".join(fr2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00, 04.05.1786␣
↪00:00:00
```

```
>>> print(fr2ascii(list(edate)))
['12.11.2014 12:00', '01.03.2015 17:56:00', '01.12.1990', '04.05.1786']
```

```
>>> print(fr2ascii(tuple(edate)))
('12.11.2014 12:00', '01.03.2015 17:56:00', '01.12.1990', '04.05.1786')
```

```
>>> print(fr2ascii(np.array(edate)))
['12.11.2014 12:00' '01.03.2015 17:56:00' '01.12.1990' '04.05.1786']
```

```
>>> print(fr2ascii(edate[0]))
12.11.2014 12:00
```

YY=True >>> edate = ['12/11/14 12:00', '01/03/15 17:56:00', '01/12/90'] >>> print(", ".join(fr2ascii(edate, YY=True))) 12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990

```
>>> print(", ".join(fr2ascii(edate, full=True, YY=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00
```

us2ascii(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with ascii date format output (default): *ascii2ascii(edate, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> edate = ascii2ascii(edate, us=True)
>>> print(", ".join(us2ascii(edate)))
12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990, 04.05.1786
```

```
>>> print(", ".join(us2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00, 04.05.1786␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> edate = ascii2ascii(edate, us=True, YY=True)
>>> print(", ".join(us2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00
```

eng2ascii(*edate*, ***kwarg*)

Wrapper function for *ascii2ascii()* with ascii date format output (default): *ascii2ascii(edate, **kwarg)*

Examples

```
>>> edate = ['2014-11-12 12:00', '01.03.2015 17:56:00', '1990-12-01', '04.05.
↪1786']
>>> edate = ascii2ascii(edate, en=True)
>>> print(", ".join(eng2ascii(edate)))
12.11.2014 12:00, 01.03.2015 17:56:00, 01.12.1990, 04.05.1786
```

```
>>> print(", ".join(eng2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00, 04.05.1786␣
↪00:00:00
```

```
>>> edate = ['14-11-12 12:00', '01.03.15 17:56:00', '90-12-01']
>>> edate = ascii2ascii(edate, en=True, YY=True)
>>> print(", ".join(eng2ascii(edate, full=True)))
12.11.2014 12:00:00, 01.03.2015 17:56:00, 01.12.1990 00:00:00
```

4.5 hesseflux.const.const

const : Provides physical, mathematical, computational, isotope, and material constants.

Defines the following constants:

Mathematical Pi, Pi2, Pi3, TwoPi, Sqrt2, pi, pi2, pi3, Twopi

Physical Gravity, T0, P0, T25, sigma, R, R_air, R_H2O, Na, REarth

Isotope R13VPDB, R18VSMOW, R2VSMOW

Computational tiny, huge, eps

Material mmol_co2, mmol_h2o, mmol_air, density_quartz, cheat_quartz, cheat_water, cheat_air, latentheat_vaporization

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

Examples

```
>>> print({:.3f}.format(Pi))
3.142
```

```
>>> print({:.3f}.format(Sqrt2))
1.414
```

```
>>> print({:.3f}.format(Gravity))
9.810
```

```
>>> print({:.3f}.format(T0))
273.150
```

```
>>> print({:.3f}.format(sigma))
5.670e-08
```

```
>>> print({:.3f}.format(R13VPDB))
0.011
```

```
>>> print({:.3f}.format(tiny))
1.000e-06
```

```
>>> print({:.3f}.format(REarth))
6371000.000
```

```
>>> print({:.3f}.format(mmol_h2o))
18.015
```

```
>>> print({:.3f}.format(mmol_air))
28.964
```

```
>>> print({:.3f}.format(density_quartz))
2.650
```

```
>>> print({:.3f}.format(cheat_quartz))
800.000
```

```
>>> print({:.3f}.format(cheat_water))
4180.000
```

```
>>> print({:.3f}.format(cheat_air))
1010.000
```

```
>>> print({:.3f}.format(latentheat_vaporization))
2.450e+06
```


- Written Jan 2012 by Matthias Cuntz (mc (at) macu (dot) de)
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Added dielectric constant for water, Mar 2014, Arndt Piayda
- Added heat capacities for air, water and quartz as well as density of quartz, Sep 2014, Arndt Piayda
- Added Pi3=pi/3, R13VPDB, R18VSMOW, R2VSMOW, Mar 2015, Matthias Cuntz
- Renamed heat capacities, molar masses, density of quartz, Mar 2015, Matthias Cuntz
- Moved calculation of dielectric constant of water to own routine, Mar 2015, Matthias Cuntz
- Added computational constants such as tiny=np.finfo(np.float).tiny, Nov 2016, Matthias Cuntz
- Added gas constants for dry air and water, May 2017, RL
- Using numpy docstring format, May 2020, Matthias Cuntz
- Added lowercase version of pi constants, May 2020, Matthias Cuntz

`Pi`	Convert a string or number to a floating point number, if possible.
`Pi2`	Convert a string or number to a floating point number, if possible.
`Pi3`	Convert a string or number to a floating point number, if possible.
`TwoPi`	Convert a string or number to a floating point number, if possible.
`pi`	Convert a string or number to a floating point number, if possible.
`pi2`	Convert a string or number to a floating point number, if possible.
`pi3`	Convert a string or number to a floating point number, if possible.
`Twopi`	Convert a string or number to a floating point number, if possible.
`Sqrt2`	Convert a string or number to a floating point number, if possible.
`Gravity`	Convert a string or number to a floating point number, if possible.
`T0`	Convert a string or number to a floating point number, if possible.
`P0`	Convert a string or number to a floating point number, if possible.
`T25`	Convert a string or number to a floating point number, if possible.

Continued on next page

Table 4 – continued from previous page

sigma	Convert a string or number to a floating point number, if possible.
R	Convert a string or number to a floating point number, if possible.
R_air	Convert a string or number to a floating point number, if possible.
R_H2O	Convert a string or number to a floating point number, if possible.
Na	Convert a string or number to a floating point number, if possible.
REarth	Convert a string or number to a floating point number, if possible.
mmol_co2	Convert a string or number to a floating point number, if possible.
mmol_h2o	Convert a string or number to a floating point number, if possible.
mmol_air	Convert a string or number to a floating point number, if possible.
density_quartz	Convert a string or number to a floating point number, if possible.
cheat_quartz	Convert a string or number to a floating point number, if possible.
cheat_water	Convert a string or number to a floating point number, if possible.
cheat_air	Convert a string or number to a floating point number, if possible.
latentheat_vaporization	Convert a string or number to a floating point number, if possible.
R13VPDB	Convert a string or number to a floating point number, if possible.
R18VSMOW	Convert a string or number to a floating point number, if possible.
R2VSMOW	Convert a string or number to a floating point number, if possible.
tiny	
huge	
eps	

4.6 hesseflux.const

Purpose

const provides physical, mathematical, computational, isotope, and material constants, such as Pi = 3.14159265358979323846264338327950288419 7.

The module is part of the JAMS Python package https://github.com/mcuntz/jams_python It will be synchronised with the JAMS package irregularily if used in other packages.

Subpackages

const	const : Provides physical, mathematical, computational, isotope, and material constants.

4.7 hesseflux.date2dec

date2dec : Converts calendar dates into decimal dates.

This module was written by Arndt Piayda and then enhanced and maintained by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jun 2010 by Arndt Piayda
- Input can be scalar, list, array, or mix of it, Feb 2012, Matthias Cuntz
- Changed checks, added calendars decimal and decimal360, Feb 2012, Matthias Cuntz
- Changed units of proleptic_gregorian calendar from days since 0001-01-01 00:00:00 to days since 0001-01-00 00:00:00, Dec 2010, Matthias Cuntz
- Deal with Excel leap year error, Feb 2013, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- ascii/eng without time defaults to 00:00:00, Jul 2013, Matthias Cuntz
- Excel starts at 1 not at 0 on 01 January 1900 or 1904, Oct 2013, Matthias Cuntz
- Bug: 01.01.0001 was substracted if Julian calendar even with units given, Oct 2013, Matthias Cuntz
- Removed remnant of time treatment before time check in eng keyword, Nov 2013, Matthias Cuntz
- Adapted to new netCDF4/netcdftime (>= v1.0) and datetime (>= Python v2.7.9), Jun 2015, Matthias Cuntz
- Call date2num with list instead of single netCDF4.datetime objects, Oct 2015, Matthias Cuntz
- mo for months always integer, Oct 2016, Matthias Cuntz
- 00, 01, etc. for integers not accepted by Python 3, removed from examples and code, Nov 2016, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz
- Succeed eng by en keyword as in ascii2ascii and dec2date, Jul 2020, Matthias Cuntz
- proleptic_gregorian instead of gregorian calendar for Excel dates, Jul 2020, Matthias Cuntz

The following functions are provided

date2dec([calendar, units, excelerr, yr, ...])	Convert scalar and array_like with calendar dates into decimal dates.

date2dec(*calendar='standard'*, *units=None*, *excelerr=True*, *yr=1*, *mo=1*, *dy=1*, *hr=0*, *mi=0*, *sc=0*, *ascii=None*, *en=None*, *eng=None*)

Convert scalar and array_like with calendar dates into decimal dates. Supported calendar formats are standard, gregorian, julian, proleptic_gregorian, excel1900, excel1904, 365_day, noleap, 366_day, all_leap, 360_day, decimal, or decimal360.

Input is year, month day, hour, minute, second or a combination of them. Input as date string is possible.

Output is decimal date with day as unit.

Parameters

- **yr** (*array_like, optional*) – years (default: 1)
- **mo** (*array_like, optional*) – month (default: 1)
- **dy** (*array_like, optional*) – days (default: 1)

- **hr** (*array_like, optional*) – hours (default: 0)
- **mi** (*array_like, optional*) – minutes (default: 0)
- **sc** (*array_like, optional*) – seconds (default: 0)
- **ascii** (*array_like, optional*) – strings of the format 'dd.mm.yyyy hh:mm:ss'. Missing hour, minutes and/or seconds are set to their default values (0).

  ascii overwrites all other keywords.

  ascii and *eng* are mutually exclusive.
- **en** (*array_like, optional*) – strings of the format 'yyyy-mm-dd hh:mm:ss'. Missing hour, minutes and/or seconds are set to their default values (0).

  en overwrites all other keywords.

  en and *ascii* are mutually exclusive.
- **eng** (*array_like, optional*) – Same as en: obsolete.
- **calendar** (*str, optional*) – Calendar of output dates (default: 'standard').

  Possible values are:

  'standard', 'gregorian' = julian calendar from 01.01.-4712 12:00:00 (BC) until 05.03.1583 00:00:00 and gregorian calendar from 15.03.1583 00:00:00 until now. Missing 10 days do not exsist.

  'julian' = julian calendar from 01.01.-4712 12:00:00 (BC) until now.

  'proleptic_gregorian' = gregorian calendar from 01.01.0001 00:00:00 until now.

  'excel1900' = Excel dates with origin at 01.01.1900 00:00:00.

  'excel1904' = Excel 1904 (Lotus) format. Same as excel1904 but with origin at 01.01.1904 00:00:00.

  '365_day', 'noleap' = 365 days format, i.e. common years only (no leap years) with origin at 01.01.0001 00:00:00.

  '366_day', 'all_leap' = 366 days format, i.e. leap years only (no common years) with origin at 01.01.0001 00:00:00.

  '360_day' = 360 days format, i.e. years with only 360 days (30 days per month) with origin at 01.01.0001 00:00:00.

  'decimal' = decimal year instead of decimal days.

  'decimal360' = decimal year with a year of 360 days, i.e. 12 month with 30 days each.
- **units** (*str, optional*) – User set units of output dates. Must be a string in the format 'days since yyyy-mm-dd hh:mm:ss'. Default values are set automatically depending on *calendar*.
- **excelerr** (*bool, optional*) – In Excel, the year 1900 is normally considered a leap year, which it was not. By default, this error is taken into account if *calendar=='excel1900'* (default: True).

  1900 is not considered a leap year if *excelerr==False*.

Returns array_like with decimal dates. The type of output will be the same as the input type.

Return type array_like

Notes

Most versions of *datetime* do not support negative years, i.e. Julian days < 1721423.5 = 01.01.0001 00:00:00.

There is an issue in *netcdftime* version < 0.9.5 in proleptic_gregorian for dates before year 301: dec2date(date2dec(ascii='01.01.0300 00:00:00', calendar='proleptic_gregorian'), calendar='proleptic_gregorian') [300, 1, 2, 0, 0, 0] dec2date(date2dec(ascii='01.01.0301 00:00:00', calendar='proleptic_gregorian'), calendar='proleptic_gregorian') [301, 1, 1, 0, 0, 0]

List input is only supported up to 2 dimensions.

Requires *netcdftime.py* from module *netcdftime* available at: http://netcdf4-python.googlecode.com

Examples

Some implementations of datetime do not support negative years >>> import datetime >>> if datetime.MINYEAR > 0: ... print('The minimum year in your datetime implementation is ', datetime.MINYEAR) ... print('i.e. it does not support negative years (BC).') The minimum year in your datetime implementation is 1 i.e. it does not support negative years (BC).

```
>>> if datetime.MINYEAR > 0:
...     year   = np.array([2000, 1810, 1630, 1510, 1271, 619, 2, 1])
... else:
...     year   = np.array([2000, 1810, 1630, 1510, 1271, 619, -1579, -4712])
>>> month  = np.array([1, 4, 7, 9, 3, 8, 8, 1])
>>> day    = np.array([5, 24, 15, 20, 18, 27, 23, 1])
>>> hour   = np.array([12, 16, 10, 14, 19, 11, 20, 12])
>>> minute = np.array([30, 15, 20, 35, 41, 8, 3, 0])
>>> second = np.array([15, 10, 40, 50, 34, 37, 41, 0])
>>> decimal = date2dec(calendar = 'standard', yr=year, mo=month, dy=day,␣
↪hr=hour, mi=minute, sc=second)
>>> nn = year.size
>>> print('{:.14e} {:.14e} {:.14e} {:.14e}'.format(*decimal[:nn//2]))
2.45154902100695e+06 2.38226217719907e+06 2.31660093101852e+06 2.
↪27284810821759e+06
>>> print('{:.14e} {:.14e}'.format(*decimal[nn//2:nn-2]))
2.18536732053241e+06 1.94738596431713e+06
>>> decimal = date2dec(calendar='standard', yr=year, mo=6, dy=15, hr=12,␣
↪mi=minute, sc=second)
>>> print('{:.14e} {:.14e} {:.14e} {:.14e}'.format(*decimal[:nn//2]))
2.45171102100695e+06 2.38231401053241e+06 2.31657101435185e+06 2.
↪27275102488426e+06
>>> print('{:.14e} {:.14e}'.format(*decimal[nn//2:nn-2]))
2.18545602886574e+06 1.94731300598380e+06
```

ascii input >>> if datetime.MINYEAR > 0: ... a = np.array(['05.01.2000 12:30:15', '24.04.1810 16:15:10', '15.07.1630 10:20:40', '20.09.1510 14:35:50', ... '18.03.1271 19:41:34', '27.08. 619 11:08:37', '23.08.0002 20:03:41', '01.01.0001 12:00:00']) ... else: ... a = np.array(['05.01.2000 12:30:15', '24.04.1810 16:15:10', '15.07.1630 10:20:40', '20.09.1510 14:35:50', ... '18.03.1271 19:41:34', '27.08. 619 11:08:37', '23.08.-1579 20:03:41', '01.01.-4712 12:00:00']) >>> decimal = date2dec(calendar='standard', ascii=a) >>> nn = a.size >>> print('{:.14e} {:.14e} {:.14e} {:.14e}'.format(*decimal[:nn//2])) 2.45154902100695e+06 2.38226217719907e+06 2.31660093101852e+06 2.27284810821759e+06 >>> print('{:.14e} {:.14e}'.format(*decimal[nn//2:nn-2])) 2.18536732053241e+06 1.94738596431713e+06

calendar = 'julian' >>> decimal = date2dec(calendar='julian', ascii=a) >>> print('{:.14e} {:.14e} {:.14e} {:.14e}'.format(*decimal[:nn//2])) 2.45156202100695e+06 2.38227417719907e+06 2.31661093101852e+06 2.27284810821759e+06 >>> print('{:.14e} {:.14e}'.format(*decimal[nn//2:nn-2])) 2.18536732053241e+06 1.94738596431713e+06

calendar = 'proleptic_gregorian' >>> decimal = date2dec(calendar='proleptic_gregorian', ascii=a) >>> print('{:.7f} {:.7f} {:.7f} {:.7f}'.format(*decimal[:nn//2])) 730123.5210069 660836.6771991 595175.4310185 551412.6082176 >>> print('{:.7f} {:.7f}'.format(*decimal[nn//2:nn-2])) 463934.8205324 225957.4643171

calendar = ‘excel1900’ WITH excelerr=True -> 1900 considered as leap year >>> d = np.array([‘05.01.2000 12:30:15’, ‘27.05.1950 16:25:10’, ‘13.08.1910 10:40:55’, … ‘01.03.1900 00:00:00’, ‘29.02.1900 00:00:00’, ‘28.02.1900 00:00:00’, … ‘01.01.1900 00:00:00’]) >>> decimal = date2dec(calendar=’excel1900’, ascii=d) >>> nn = d.size >>> print(‘{:.7f} {:.7f} {:.7f}’.format(*decimal[:nn//2])) 36530.5210069 18410.6841435 3878.4450810 >>> print(‘{:.1f} {:.1f} {:.1f} {:.1f}’.format(*decimal[nn//2:])) 61.0 60.0 59.0 1.0

calendar = ‘excel1900’ WITH excelerr = False -> 1900 is NO leap year >>> decimal = date2dec(calendar=’excel1900’, ascii=d, excelerr=False) >>> print(‘{:.7f} {:.7f} {:.7f}’.format(*decimal[:nn//2])) 36529.5210069 18409.6841435 3877.4450810 >>> print(‘{:.1f} {:.1f} {:.1f} {:.1f}’.format(*decimal[nn//2:])) 60.0 60.0 59.0 1.0

calendar = ‘excel1904’ >>> decimal = date2dec(calendar=’excel1904’, ascii=d[:nn//2]) >>> print(‘{:.7f} {:.7f} {:.7f}’.format(*decimal[:nn//2])) 35069.5210069 16949.6841435 2417.4450810

calendar = ‘365_day’ >>> g = np.array([‘18.08.1972 12:30:15’, ‘25.10.0986 12:30:15’, ‘28.11.0493 22:20:40’, ‘01.01.0001 00:00:00’]) >>> decimal = date2dec(calendar=‘365_day’, ascii=g) >>> nn = g.size >>> print(‘{:.7f} {:.7f} {:.7f} {:.7f}’.format(*decimal[:nn])) 719644.5210069 359822.5210069 179911.9310185 0.0000000

calendar = ‘366_day’ >>> decimal = date2dec(calendar=‘366_day’, ascii=g) >>> print(‘{:.7f} {:.7f} {:.7f} {:.7f}’.format(*decimal[:nn])) 721616.5210069 360808.5210069 180404.9310185 0.0000000

360_day does not work with netcdftime.py version equal or below 0.9.2 # calendar = ‘360_day’ >>> decimal = date2dec(calendar=‘360_day’, ascii=g) >>> print(‘{:.7f} {:.7f} {:.7f} {:.7f}’.format(*decimal[:nn])) 709787.5210069 354894.5210069 177447.9310185 0.0000000

```
>>> print('{:.7f}'.format(date2dec(yr=1992, mo=1, dy=26, hr=2, mi=0, sc=0,␣
↪calendar='decimal')))
1992.0685337
>>> print('{:.7f}'.format(date2dec(ascii='26.01.1992 02:00', calendar=
↪'decimal360')))
1992.0696759
>>> print('{:.7f} {:.7f}'.format(*date2dec(ascii=['26.01.1992 02:00','26.01.
↪1992 02:00'], calendar='decimal360')))
1992.0696759 1992.0696759
>>> print('{:.7f} {:.7f}'.format(*date2dec(yr=[1992,1992], mo=1, dy=26, hr=2,␣
↪mi=0, sc=0, calendar='decimal360')))
1992.0696759 1992.0696759
>>> print('{:.7f} {:.7f}'.format(*date2dec(yr=np.array([1992,1992]), mo=1,␣
↪dy=26, hr=2, mi=0, sc=0, calendar='decimal360')))
1992.0696759 1992.0696759
>>> decimal = date2dec(ascii=[['26.01.1992 02:00','26.01.1992 02:00'],
...                           ['26.01.1992 02:00','26.01.1992 02:00'],
...                           ['26.01.1992 02:00','26.01.1992 02:00']],
...                    calendar='decimal360')
>>> print('{:.7f} {:.7f}'.format(*decimal[0]))
1992.0696759 1992.0696759
>>> print('{:.7f} {:.7f}'.format(*decimal[2]))
1992.0696759 1992.0696759
>>> print((date2dec(ascii='01.03.2003 00:00:00') - date2dec(ascii='01.03.2003
↪')) == 0.)
True
```

en >>> decimal = date2dec(en=‘1992-01-26 02:00’, calendar=’decimal360’) >>> print(‘{:.7f}’.format(decimal)) 1992.0696759 >>> decimal = date2dec(eng=‘1992-01-26 02:00’, calendar=’decimal360’) >>> print(‘{:.7f}’.format(decimal)) 1992.0696759

4.8 hesseflux.dec2date

dec2date : Converts decimal dates to calendar dates.

This module was written by Arndt Piayda and then enhanced and maintained by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jun 2010 by Arndt Piayda
- Input can be scalar, list, array, or mix of it, Feb 2012, Matthias Cuntz
- Changed checks, added calendars decimal and decimal360, Feb 2012, Matthias Cuntz
- fulldate=True default, Feb 2012, Matthias Cuntz
- Change keyword name: units -> refdate, Jun 2012, Matthias Cuntz
- units has now same meaning as in netcdftime, Jun 2012, Matthias Cuntz
- Include cdo absolute date units 'day as %Y%m%d.%f', Jun 2012, Matthias Cuntz
- Solved Excel leap year problem, Feb 2013, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Small bug in eng output, May 2013, Arndt Piayda
- Excel starts at 1 not at 0 on 01 January 1900 or 1904, Oct 2013, Matthias Cuntz
- Bug: 01.01.0001 was substracted if Julian calendar even with units given, Oct 2013, Matthias Cuntz
- Include units 'month as %Y%m.%f' and 'year as %Y.%f', May 2016, Matthias Cuntz
- Adapted to new netCDF4/netcdftime (>= v1.0) and datetime (>= Python v2.7.9), Jun 2015, Matthias Cuntz
- leap always integer, Oct 2016, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz
- Succeed eng by en keyword as in ascii2ascii and date2dec, Jul 2020, Matthias Cuntz
- proleptic_gregorian instead of gregorian calendar for Excel dates, Jul 2020, Matthias Cuntz

The following functions are provided

dec2date(indata[, calendar, refdate, units, ...])	Converts scale and array_like with decimal dates into calendar dates.

dec2date(*indata*, *calendar='standard'*, *refdate=None*, *units=None*, *excelerr=True*, *fulldate=None*, *yr=False*, *mo=False*, *dy=False*, *hr=False*, *mi=False*, *sc=False*, *ascii=False*, *en=False*, *eng=False*)

Converts scale and array_like with decimal dates into calendar dates. Supported time formats are: standard, gregorian, julian, proleptic_gregorian, excel1900, excel1904, 365_day, noleap, 366_day, all_leap, and 360_day.

Input is decimal date in units of days.

Output is year, month, day, hour, minute, second or any combination of them. Output in string format is possible.

Parameters

- **indata** (*array_like*) – Input decimal dates. Dates must be positive.
- **calendar** (*str, optional*) – Calendar of input dates (default: 'standard').

Possible values are:

'standard', 'gregorian' = julian calendar from 01.01.-4712 12:00:00 (BC) until 05.03.1583 00:00:00 and gregorian calendar from 15.03.1583 00:00:00 until now. Missing 10 days do not exsist.

'julian' = julian calendar from 01.01.-4712 12:00:00 (BC) until now.

'proleptic_gregorian' = gregorian calendar from 01.01.0001 00:00:00 until now.

'excel1900' = Excel dates with origin at 01.01.1900 00:00:00.

'excel1904' = Excel 1904 (Lotus) format. Same as excel1904 but with origin at 01.01.1904 00:00:00.

'365_day', 'noleap' = 365 days format, i.e. common years only (no leap years) with origin at 01.01.0001 00:00:00.

'366_day', 'all_leap' = 366 days format, i.e. leap years only (no common years) with origin at 01.01.0001 00:00:00.

'360_day' = 360 days format, i.e. years with only 360 days (30 days per month) with origin at 01.01.0001 00:00:00.

'decimal' = decimal year instead of decimal days.

'decimal360' = decimal year with a year of 360 days, i.e. 12 month with 30 days each.

4.9 hesseflux.division

division : Divide two arrays, return *otherwise* if division by 0.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jan 2012 by Matthias Cuntz (mc (at) macu (dot) de)
- Added wrapper div, May 2012, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Do not return masked array if no masked array given, Oct 2014, Matthias Cuntz
- Added two-digit year, Nov 2018, Matthias Cuntz
- Removed bug that non-masked array was returned if masked array given, Sep 2015, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

division(a, b[, otherwise, prec])	Divide two arrays, return *otherwise* if division by 0.
div(*args, **kwargs)	Wrapper function for *division()*.

division (*a*, *b*, *otherwise=nan*, *prec=0.0*)

Divide two arrays, return *otherwise* if division by 0.

Parameters

- **a** (*array_like*) – enumerator
- **b** (*array_like*) – denominator
- **otherwise** (*float*) – value to return if *b=0* (default: *np.nan*)
- **prec** (*float*) – if |*b*|<|*prec*| then *otherwise*

Returns

ratio – a/b if |*b*| > |*prec*|

otherwise if |*b*| <= |*prec*|

Output is numpy array. It is a masked array if at least one of *a* or *b* is a masked array.

Return type numpy array or masked array

Examples

```
>>> a = [1., 2., 3.]
>>> b = 2.
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b)))
0.5 1.0 1.5
```

```
>>> a = [1., 1., 1.]
>>> b = [2., 1., 0.]
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b)))
0.5 1.0 nan
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b, 0.)))
0.5 1.0 0.0
```

(continues on next page)

(continued from previous page)

```
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b, otherwise=0.)))
0.5 1.0 0.0
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b, prec=1.)))
0.5 nan nan
```

```
>>> import numpy as np
>>> a = np.array([1., 1., 1.])
>>> b = [2., 1., 0.]
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b)))
0.5 1.0 nan
```

```
>>> b = np.array([2., 1., 0.])
>>> print('{:.1f} {:.1f} {:.1f}'.format(*division(a, b)))
0.5 1.0 nan
```

```
>>> mask = [0, 0, 1]
>>> b = np.ma.array([2., 1., 0.], mask=mask)
>>> ratio = division(a, b)
>>> print(ratio)
[0.5 1.0 --]
```

div(**args*, ***kwargs*)

Wrapper function for *division()*.

Examples

```
>>> a = [1., 2., 3.]
>>> b = 2.
>>> print('{:.1f} {:.1f} {:.1f}'.format(*div(a, b)))
0.5 1.0 1.5
```

```
>>> a = [1., 1., 1.]
>>> b = [2., 1., 0.]
>>> print('{:.1f} {:.1f} {:.1f}'.format(*div(a, b)))
0.5 1.0 nan
>>> print('{:.1f} {:.1f} {:.1f}'.format(*div(a, b, 0.)))
0.5 1.0 0.0
>>> print('{:.1f} {:.1f} {:.1f}'.format(*div(a, b, otherwise=0.)))
0.5 1.0 0.0
>>> print('{:.1f} {:.1f} {:.1f}'.format(*div(a, b, prec=1.)))
0.5 nan nan
```

4.10 hesseflux.esat

esat : Saturation vapour pressure of water and ice.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jan 2012 by Matthias Cuntz (mc (at) macu (dot) de)
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Changed handling of masked arrays, Oct 2013, Matthias Cuntz
- Assert T>0, Apr 2014, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

esat(T[, liquid, formula])	Calculates the saturation vapour pressure of water and/or ice.

esat (*T*, *liquid=False*, *formula='GoffGratch'*)

Calculates the saturation vapour pressure of water and/or ice.

For temperatures above (and equal) 0 degree C (273.15 K), the vapour pressure over liquid water is calculated. For temperatures below 0 degree C, the vapour pressure over ice is calculated.

The optional parameter liquid=True changes the calculation to vapour pressure over liquid water over the entire temperature range.

Parameters

- **T** (*float or array_like*) – Temperature [K]
- **liquid** (*bool, optional*) – If True, use liquid formula for all temperatures.
- **formula** (*str, optional*) – Name of reference to use for calculations, case-insensitive (default: GoffGratch).

  Note that several formulations do not provide a vapour pressure formulation over ice and Marti and Mauersberger do not provide a formula over liquid: GoffGratch is used in theses cases.

  GoffGratch: Smithsonian Tables, 1984; after Goff and Gratch, 1946 (default)

  MartiMauersberger: Marti and Mauersberger, 1993

  MagnusTeten: Murray, 1967

  Buck_original: Buck, 1981

  Buck: Buck Research Manual, 1996

  WMO: Goff, 1957; WMO 1988, 2000

  Wexler: Wexler, 1977

  Sonntag: Sonntag, 1994

  Bolton: Bolton, 1980

  Fukuta: Fukuta, N. and C. M. Gramada, 2003

  HylandWexler: Hyland and Wexler, 1983

IAPWS: Wagner and Pruss, 2002

MurphyKoop: Murphy and Koop, 2005

Returns Saturation water pressure at temperature T in Pascal [Pa].

Return type float or array_like

Notes

From Holger Voemel: http://cires.colorado.edu/~voemel/vp.html

Referred literature cited in code.

Examples

```
>>> print('{:.3f}'.format(esat(293.15)))
2335.847
>>> print('{:.3f}'.format(esat(253.15)))
103.074
```

```
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15])))
2335.847 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='GoffGratch')))
2335.847 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula=
↪'MartiMauersberger')))
2335.847 103.650
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='MagnusTeten')))
2335.201 102.771
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='buck')))
2338.340 103.286
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='Buck_original
↪')))
2337.282 103.267
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='wmo')))
2337.080 103.153
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='WEXLER')))
2323.254 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='Sonntag')))
2339.249 103.249
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='Bolton')))
2336.947 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='Fukuta')))
2335.847 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='HylandWexler
↪')))
2338.804 103.260
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='IAPWS')))
2339.194 103.074
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='MurphyKoop')))
2339.399 103.252
```

```
>>> print('{:.3f} {:.3f}'.format(*esat(np.array([293.15,253.15]),␣
↪liquid=True)))
2335.847 125.292
>>> print('{:.3f} {:.3f}'.format(*esat([293.15,253.15],formula='Fukuta',␣
↪liquid=True)))
2335.847 125.079
```

```
>>> print('{:.3f} {:.3f}'.format(*esat(np.array([293.15,393.15]))))
esat.py: UserWarning: T>373.15 K - something might be wrong with T.
2335.847 198473.378
>>> print('{:.3f} {:.3f}'.format(*esat(np.array([293.15,93.15]))))
esat.py: UserWarning: T<100 - T probably given in Celsius instead of Kelvin.
2335.847 0.000
```

```
>>> out = esat(np.ma.array([253.15,-9999.], mask=[False,True]))
>>> print('{:.3f} {:.3f}'.format(*out.filled(-9999.)))
103.074 -9999.000
```

4.11 hesseflux.fgui

fgui : GUI dialogs to choose files and directories using Tkinter.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jun 2014 by Matthias Cuntz (mc (at) macu (dot) de)
- Added directories_from_gui, Oct 2015, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

directory_from_gui([initialdir, title])	Opens dialog to select directory.
directories_from_gui([initialdir, title])	Open dialog to select several directories.
file_from_gui([initialdir, title, multiple])	Wrapper for *files_from_gui()* with multiple=False, i.e.
files_from_gui([initialdir, title, multiple])	Open dialog to select one or several files.

directory_from_gui (*initialdir='.'*, *title='Choose directory.'*)

Opens dialog to select directory.

Parameters

- **initialdir** (*str, optional*) – Initial directory, in which opens GUI (default: '.')
- **title** (*str, optional*) – Title of GUI (default: 'Choose directory.')

Returns Selected directory.

Return type str

Examples

```
if not idir:
    idir = directory_from_gui()
    if not idir:
        raise ValueError('Error: no directory given.')
```

directories_from_gui (*initialdir='.'*, *title='Choose one or several directories.'*)

Open dialog to select several directories.

Parameters

- **initialdir** (*str, optional*) – Initial directory, in which opens GUI (default: '.')
- **title** (*str, optional*) – Title of GUI (default: 'Choose one or several directories.'

Returns Selected directories.

Return type list

Examples

```
if not dirs:
    dirs = directories_from_gui()
    if not dirs:
        raise ValueError('Error: no directories given.')
```

file_from_gui (*initialdir='.', title='Choose file', multiple=False*)
Wrapper for *files_from_gui()* with multiple=False, i.e. open dialog to select one file.

Examples

```
if not file:
    file = file_from_gui()
    if not file:
        raise ValueError('Error: no input file given.')
```

files_from_gui (*initialdir='.', title='Choose file(s).', multiple=True*)
Open dialog to select one or several files.

Parameters

- **initialdir** (*str, optional*) – Initial directory, in which opens GUI (default: '.')
- **title** (*str, optional*) – Title of GUI (default: 'Choose file(s).')
- **multiple** (*bool, optional*) – True: allow selection of multiple files (default). False: only one single file possible to select.

Returns Selected files.

Return type list

Note: It always returns a list even with *multiple=False*.

Examples

```
if not files:
    files = files_from_gui()
    if not files:
        raise ValueError('Error: no input file(s) given.')
```

4.12 hesseflux.fread

fread : Read numbers into array from a file.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jul 2009 by Matthias Cuntz (mc (at) macu (dot) de)
- Keyword transpose, Feb 2012, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Removed bug when nc is list and contains 0, Nov 2014, Matthias Cuntz
- Keyword hskip, Nov 2014, Matthias Cuntz
- Speed improvements: everything list until the very end, Feb 2015, Matthias Cuntz
- range instead of np.arange, Nov 2017, Matthias Cuntz
- Keywords cname, sname, hstrip, rename file to infile, Nov 2017, Matthias Cuntz
- Ignore unicode characters on read, Jun 2019, Matthias Cuntz
- Make ignoring unicode characters campatible with Python 2 and Python 3, Jul 2019, Matthias Cuntz
- Keywords encoding, errors with codecs module, Aug 2019, Matthias Cuntz
- Keyword return_list, Dec 2019, Stephan Thober
- return_list=False default, Jan 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

fread(infile[, nc, cname, skip, cskip, ...])	Read numbers into array with floats from a file.

fread(*infile*, *nc=0*, *cname=None*, *skip=0*, *cskip=0*, *hskip=0*, *hstrip=True*, *separator=None*, *squeeze=False*, *reform=False*, *skip_blank=False*, *comment=None*, *fill=False*, *fill_value=0*, *strip=None*, *encoding='ascii'*, *errors='ignore'*, *header=False*, *full_header=False*, *transpose=False*, *strarr=False*, *return_list=False*)

Read numbers into array with floats from a file.

Lines or columns can be skipped. Columns can be picked specifically.

Blank (only whitespace) and comment lines can be excluded.

The header of the file can be read separately.

This routines is exactly the same as sread but transforms everything to floats, handling NaN and Inf.

Parameters

- **infile** (*str*) – source file name
- **nc** (*int or iterable, optional*) – number of columns to be read [default: all (*nc<=0*)].

  nc can be an int or a vector of column indexes, starting with 0; *cskip* will be ignored in the latter case.
- **cname** (*iterable of str, optional*) – columns can be chosen by the values in the first header line; must be iterable with strings.

- **skip** (*int, optional*) – number of lines to skip at the beginning of file (default: 0)
- **cskip** (*int, optional*) – number of columns to skip at the beginning of each line (default: 0)
- **hskip** (*int, optional*) – number of lines in skip that do not belong to header (default: 0)
- **hstrip** (*bool, optional*) – True: strip header cells to match with cname (default: True)
- **separator** (*str, optional*) – column separator. If not given, columns separators are (in order): comma (‘,’), semicolon (‘;’), whitespace.
- **comment** (*iterable, optional*) – line gets excluded if first character of line is in comment sequence. Sequence must be iterable such as string, list and tuple.
- **fill_value** (*float, optional*) – value to fill in array in empty cells or if not enough columns in line and *fill==True* (default: 0, and ‘’ for header).
- **strip** (*str, optional*) – Strip strings with str.strip(strip).

  None: strip quotes ” and ‘ (default).

  False: no strip (~30% faster).

  str: strip character given by *strip*.
- **encoding** (*str, optional*) – Specifies the encoding which is to be used for the file (default: ‘ascii’). Any encoding that encodes to and decodes from bytes is allowed.
- **errors** (*str, optional*) – Errors may be given to define the error handling during encoding of the file (default: ‘ignore’).

  Possible values: ‘strict’, ‘replace’, ‘ignore’.
- **squeeze** (*bool, optional*) – True: 2-dim array will be cleaned of degenerated dimension, i.e. results in a vector.

  False: array will be two-dimensional as read (default)
- **reform** (*bool, optional*) – Same as squeeze.
- **skip_blank** (*bool, optional*) – True: continues reading after blank line.

  False: stops reading at first blank line (default).
- **fill** (*bool, optional*) – True: fills in *fill_value* if not enough columns in line.

  False: stops execution and returns None if not enough columns in line (default).
- **header** (*bool, optional*) – True: header strings will be returned.

  False: numbers in file will be returned (default).
- **full_header** (*bool, optional*) – True: header is a string vector of the skipped rows.

  False: header will be split in columns, exactly as the data, and will hold only the selected columns (default).
- **transpose** (*bool, optional*) – True: column-major format *output(0:ncolumns,0:nlines)*.

  False: row-major format *output(0:nlines,0:ncolumns)* (default).
- **strarr** (*bool, optional*) – True: return header as numpy array of strings.

  False: return header as list (default).

- **return_list** (*bool, optional*) – True: return file content as list.

  False: return file content as numpy array (default).

Returns

Depending on options:

Array of floats if *header==False*.

List of floats if *return_list==True*.

List with file header strings if *header==True*.

String array of file header if *header==True* and *strarr==True*.

List of lines of strings if *header=True* and *full_header=True*.

Return type array of floats

Notes

If *header==True* then skip is counterintuitive because it is actually the number of header rows to be read. This is to be able to have the exact same call of the function, once with *header=False* and once with *header=True*.

If *fill==True*, blank lines are not filled but are taken as end of file.

transpose=True has no effect on 1D output such as 1 header line.

Examples

```
>>> # Create some data
>>> filename = 'test.dat'
>>> ff = open(filename,'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 2.2 2.3 2.4\n')
>>> ff.close()
```

```
>>> # Read sample file in different ways
>>> # header
>>> print(fread(filename, nc=2, skip=1, header=True))
['head1', 'head2']
>>> print(fread(filename, nc=2, skip=1, header=True, full_header=True))
['head1 head2 head3 head4']
>>> print(fread(filename, nc=1, skip=2, header=True))
[['head1'], ['1.1']]
>>> print(fread(filename, nc=1, skip=2, header=True, squeeze=True))
['head1', '1.1']
>>> print(fread(filename, nc=1, skip=2, header=True, strarr=True))
[['head1']
 ['1.1']]
```

```
>>> # data
>>> print(fread(filename, skip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=2))
[[2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=1, cskip=1))
[[1.2 1.3 1.4]
 [2.2 2.3 2.4]]
```

(continues on next page)

(continued from previous page)

```
>>> print(fread(filename, nc=2, skip=1, cskip=1))
[[1.2 1.3]
 [2.2 2.3]]
>>> print(fread(filename, nc=[1,3], skip=1))
[[1.2 1.4]
 [2.2 2.4]]
>>> print(fread(filename, nc=1, skip=1))
[[1.1]
 [2.1]]
>>> print(fread(filename, nc=1, skip=1, reform=True))
[1.1 2.1]
```

```
>>> # skip blank lines
>>> ff = open(filename, 'a')
>>> ff.writelines('\n')
>>> ff.writelines('3.1 3.2 3.3 3.4\n')
>>> ff.close()
>>> print(fread(filename, skip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=1, skip_blank=True, comment='#!'))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]
 [3.1 3.2 3.3 3.4]]
```

```
>>> # skip comment lines
>>> ff = open(filename, 'a')
>>> ff.writelines('# First comment\n')
>>> ff.writelines('! Second 2 comment\n')
>>> ff.writelines('4.1 4.2 4.3 4.4\n')
>>> ff.close()
>>> print(fread(filename, skip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=1, nc=[2], skip_blank=True, comment='#'))
[[1.3]
 [2.3]
 [3.3]
 [2. ]
 [4.3]]
>>> print(fread(filename, skip=1, skip_blank=True, comment='#!'))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]
 [3.1 3.2 3.3 3.4]
 [4.1 4.2 4.3 4.4]]
>>> print(fread(filename, skip=1, skip_blank=True, comment=('#','!')))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]
 [3.1 3.2 3.3 3.4]
 [4.1 4.2 4.3 4.4]]
>>> print(fread(filename, skip=1, skip_blank=True, comment=['#','!']))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]
 [3.1 3.2 3.3 3.4]
 [4.1 4.2 4.3 4.4]]
```

```
>>> # fill missing columns
>>> ff = open(filename, 'a')
>>> ff.writelines('5.1 5.2\n')
>>> ff.close()
```

(continues on next page)

(continued from previous page)

```
>>> print(fread(filename, skip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=1, skip_blank=True, comment='#!', fill=True,␣
↪fill_value=-1))
[[ 1.1  1.2  1.3  1.4]
 [ 2.1  2.2  2.3  2.4]
 [ 3.1  3.2  3.3  3.4]
 [ 4.1  4.2  4.3  4.4]
 [ 5.1  5.2 -1.  -1. ]]
```

```
>>> # transpose
>>> print(fread(filename, skip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename, skip=1, transpose=True))
[[1.1 2.1]
 [1.2 2.2]
 [1.3 2.3]
 [1.4 2.4]]
```

```
>>> # Create some more data with Nan and Inf
>>> filename1 = 'test1.dat'
>>> ff = open(filename1, 'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 nan Inf "NaN"\n')
>>> ff.close()
```

```
>>> # Treat Nan and Inf with automatic strip of " and '
>>> print(fread(filename1, skip=1, transpose=True))
[[1.1 2.1]
 [1.2 nan]
 [1.3 inf]
 [1.4 nan]]
```

```
>>> # Create some more data with escaped numbers
>>> filename2 = 'test2.dat'
>>> ff = open(filename2, 'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('"1.1" "1.2" "1.3" "1.4"\n')
>>> ff.writelines('2.1 nan Inf "NaN"\n')
>>> ff.close()
```

```
>>> # Strip
>>> print(fread(filename2,  skip=1,  transpose=True,  strip='"'))
[[1.1 2.1]
 [1.2 nan]
 [1.3 inf]
 [1.4 nan]]
```

```
>>> # Create some more data with an extra (shorter) header line
>>> filename3 = 'test3.dat'
>>> ff = open(filename3, 'w')
>>> ff.writelines('Extra header\n')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 2.2 2.3 2.4\n')
>>> ff.close()
```

```
>>> print(fread(filename3, skip=2, hskip=1))
[[1.1 1.2 1.3 1.4]
 [2.1 2.2 2.3 2.4]]
>>> print(fread(filename3, nc=2, skip=2, hskip=1, header=True))
['head1', 'head2']
```

```
>>> # cname
>>> print(fread(filename, cname='head2', skip=1, skip_blank=True, comment='#!',
↪ squeeze=True))
[1.2 2.2 3.2 4.2 5.2]
>>> print(fread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!'))
[[1.1 1.2]
 [2.1 2.2]
 [3.1 3.2]
 [4.1 4.2]
 [5.1 5.2]]
>>> print(fread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', header=True))
['head1', 'head2']
>>> print(fread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', header=True, full_header=True))
['head1 head2 head3 head4']
>>> print(fread(filename, cname=['  head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', hstrip=False))
[[1.2]
 [2.2]
 [3.2]
 [4.2]
 [5.2]]
```

```
>>> # Clean up doctest
>>> import os
>>> os.remove(filename)
>>> os.remove(filename1)
>>> os.remove(filename2)
>>> os.remove(filename3)
```

4.13 hesseflux.fsread

fsread : Read from a file numbers into 2D float array as well as characters into 2D string array.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Feb 2015 by Matthias Cuntz (mc (at) macu (dot) de)
- nc<=-1 rmoved in case of nc is list, Nov 2016, Matthias Cuntz
- range instead of np.arange, Nov 2017, Matthias Cuntz
- Keywords cname, sname, hstrip, rename file to infile, Nov 2017, Matthias Cuntz
- full_header=True returns vector of strings, Nov 2017, Matthias Cuntz
- Ignore unicode characters on read, Jun 2019, Matthias Cuntz
- Keywords encoding, errors with codecs module, Aug 2019, Matthias Cuntz
- return_list=False default, Jan 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

fsread(infile[, nc, cname, snc, sname, ...])	Read from a file numbers into 2D float array as well as characters into 2D string array.

fsread(*infile*, *nc=0*, *cname=None*, *snc=0*, *sname=None*, *skip=0*, *cskip=0*, *hskip=0*, *hstrip=True*, *separator=None*, *squeeze=False*, *reform=False*, *skip_blank=False*, *comment=None*, *fill=False*, *fill_value=0*, *sfill_value=''*, *strip=None*, *encoding='ascii'*, *errors='ignore'*, *header=False*, *full_header=False*, *transpose=False*, *strarr=False*)

Read from a file numbers into 2D float array as well as characters into 2D string array.

Parameters

- **infile** (*str*) – source file name
- **nc** (*int or iterable, optional*) – number of columns to be read as floats [default: all (*nc<=0*)].

  nc can be an int or a vector of column indexes, starting with 0; *cskip* will be ignored in the latter case.

  If *snc!=0*: *nc* must be iterable or -1 to read all other columns as floats.
- **cname** (*iterable of str, optional*) – float columns can be chosen by the values in the first header line; must be iterable with strings.
- **snc** (*int or iterable, optional*) – number of columns to be read as strings [default: none (*snc=0*)].

  snc can be an int or a vector of column indexes, starting with 0; *cskip* will be ignored in the latter case.

  If *nc!=0*: *snc* must be iterable or -1 to read all other columns as strings.
- **sname** (*iterable of str, optional*) – string columns can be chosen by the values in the first header line; must be iterable with strings.
- **skip** (*int, optional*) – number of lines to skip at the beginning of file (default: 0)

- **cskip** (*int, optional*) – number of columns to skip at the beginning of each line (default: 0)
- **hskip** (*int, optional*) – number of lines in skip that do not belong to header (default: 0)
- **hstrip** (*bool, optional*) – True: strip header cells to match with cname (default: True)
- **separator** (*str, optional*) – column separator. If not given, columns separator are (in order): comma (‘,’), semicolon (‘;’), whitespace.
- **comment** (*iterable, optional*) – line gets excluded if first character of line is in comment sequence. Sequence must be iterable such as string, list and tuple.
- **fill_value** (*float, optional*) – value to fill in float array in empty cells or if not enough columns in line and *fill==True* (default: 0).
- **sfill_value** (*str, optional*) – value to fill in string array in empty cells or if not enough columns in line and *fill==True* (default: ‘’).
- **strip** (*str, optional*) – Strip strings with str.strip(strip).

  None: strip quotes ” and ‘ (default).

  False: no strip (~30% faster).

  str: strip character given by *strip*.
- **encoding** (*str, optional*) – Specifies the encoding which is to be used for the file (default: ‘ascii’). Any encoding that encodes to and decodes from bytes is allowed.
- **errors** (*str, optional*) – Errors may be given to define the error handling during encoding of the file (default: ‘ignore’).

  Possible values: ‘strict’, ‘replace’, ‘ignore’.
- **squeeze** (*bool, optional*) – True: 2-dim array will be cleaned of degenerated dimension, i.e. results in a vector.

  False: array will be two-dimensional as read (default).
- **reform** (*bool, optional*) – Same as squeeze.
- **skip_blank** (*bool, optional*) – True: continues reading after blank line.

  False: stops reading at first blank line (default).
- **fill** (*bool, optional*) – True: fills in *fill_value* if not enough columns in line.

  False: stops execution and returns None if not enough columns in line (default).
- **header** (*bool, optional*) – True: header strings will be returned.

  False: numbers in file will be returned (default).
- **full_header** (*bool, optional*) – True: header is a string vector of the skipped rows.

  False: header will be split in columns, exactly as the data, and will hold only the selected columns (default).
- **transpose** (*bool, optional*) – True: column-major format *output(0:ncolumns,0:nlines)*.

  False: row-major format *output(0:nlines,0:ncolumns)* (default).
- **strarr** (*bool, optional*) – True: return header as numpy array of strings.

  False: return header as list.

Returns

1 output: array of floats (*nc!=0* and *snc=0*)

1 output: array of strings (*nc=0* and *snc!=0*)

2 outputs: array of floats, array of strings (*nc!=0* and *snc!=0*)

1 output: list/string array of header ((*nc=0* or *snc=0*) and *header=True*)

2 outputs: list/string array of header for float array, list/string array of header for strarr ((*nc!=0* and *snc!=0*) and *header=True*)

1 output: String vector of full file header (*header=True* and *full_header=True*)

Return type array(s)

Notes

If *header==True* then skip is counterintuitive because it is actually the number of header rows to be read. This is to be able to have the exact same call of the function, once with *header=False* and once with *header=True*.

If *fill==True*, blank lines are not filled but are taken as end of file.

transpose=True has no effect on 1D output such as 1 header line.

Passes file to `fread()` if *snc==0*.

Passes file to `sread()` if *nc==0*.

Examples

```
>>> # Create some data
>>> filename = 'test.dat'
>>> ff = open(filename,'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 2.2 2.3 2.4\n')
>>> ff.close()
```

```
>>> # Read sample with fread - see fread for more examples
>>> print(fsread(filename, nc=[1,3], skip=1))
[[1.2 1.4]
 [2.2 2.4]]
>>> print(fsread(filename, nc=2, skip=1, header=True))
['head1', 'head2']
```

```
>>> # Read sample with sread - see sread for more examples
>>> print(fsread(filename, snc=[1,3], skip=1))
[['1.2', '1.4'], ['2.2', '2.4']]
```

```
>>> # Some mixed data
>>> ff = open(filename,'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('01.12.2012 1.2 name1 1.4\n')
>>> ff.writelines('01.01.2013 2.2 name2 2.4\n')
>>> ff.close()
```

```
>>> # Read columns
>>> print(fsread(filename, nc=[1,3], skip=1))
[[1.2 1.4]
```

(continues on next page)

(continued from previous page)

```
 [2.2 2.4]]
>>> a, sa = fsread(filename, nc=[1,3], snc=[0,2], skip=1)
>>> print(a)
[[1.2 1.4]
 [2.2 2.4]]
>>> print(sa[0][0])
01.12.2012
>>> print(sa[0][1])
name1
>>> print(sa[1][0])
01.01.2013
>>> print(sa[1][1])
name2
>>> a, sa = fsread(filename, nc=[1,3], snc=-1, skip=1)
>>> print(a)
[[1.2 1.4]
 [2.2 2.4]]
>>> print(sa[0][0])
01.12.2012
>>> print(sa[0][1])
name1
>>> print(sa[1][0])
01.01.2013
>>> print(sa[1][1])
name2
>>> a, sa = fsread(filename, nc=-1, snc=[0,2], skip=1)
>>> print(a)
[[1.2 1.4]
 [2.2 2.4]]
```

```
>>> # Read header
>>> a, sa = fsread(filename, nc=[1,3], snc=[0,2], skip=1, header=True)
>>> print(a)
['head2', 'head4']
>>> print(sa)
['head1', 'head3']
```

```
>>> # Some mixed data with missing values
>>> ff = open(filename,'w')
>>> ff.writelines('head1,head2,head3,head4\n')
>>> ff.writelines('01.12.2012,1.2,name1,1.4\n')
>>> ff.writelines('01.01.2013,,name2,2.4\n')
>>> ff.close()
```

```
>>> print(fsread(filename, nc=[1,3], skip=1, fill=True, fill_value=-1))
[[ 1.2  1.4]
 [-1.   2.4]]
```

```
>>> # cname, sname
>>> a, sa = fsread(filename, cname='head2', snc=[0,2], skip=1, fill=True, fill_
↪value=-1, squeeze=True)
>>> print(a)
[ 1.2 -1. ]
>>> print(sa)
[['01.12.2012', 'name1'],
 ['01.01.2013', 'name2']]
>>> a, sa = fsread(filename, cname=['head2','head4'], snc=-1, skip=1,␣
↪fill=True, fill_value=-1)
>>> print(a)
[[ 1.2  1.4]
```

(continues on next page)

(continued from previous page)

```
 [-1.   2.4]]
>>> print(sa)
[['01.12.2012', 'name1'],
 ['01.01.2013', 'name2']]
>>> a, sa = fsread(filename, nc=[1,3], sname=['head1','head3'], skip=1,␣
↪fill=True, fill_value=-1, strarr=True, header=True)
>>> print(a)
['head2' 'head4']
>>> print(sa)
['head1' 'head3']
>>> print(fsread(filename, cname=['head2','head4'], snc=-1, skip=1,␣
↪header=True, full_header=True))
['head1,head2,head3,head4']
>>> print(fsread(filename, cname=['head2','head4'], snc=-1, skip=1, fill=True,␣
↪fill_value=-1, header=True, full_header=True))
['head1,head2,head3,head4']
>>> a, sa = fsread(filename, cname=['  head2','head4'], snc=-1, skip=1,␣
↪fill=True, fill_value=-1, hstrip=False)
>>> print(a)
[[1.4]
 [2.4]]
```

```
>>> # Clean up doctest
>>> import os
>>> os.remove(filename)
```

4.14 hesseflux.functions.fit_functions

Module defines common functions that are used in curve_fit or fmin parameter estimations.

For all fit functions, it defines the functions in two forms (ex. of 3 params):

func(x, p1, p2, p3)

func_p(x, p) with *p[0:3]*

The first form can be used, for example, with *scipy.optimize.curve_fit* (ex. function f1x=a+b/x):

p, cov = scipy.optimize.curve_fit(functions.f1x, x, y, p0=[p0,p1])

It also defines two cost functions along with the fit functions, one with the absolute sum, one with the squared sum of the deviations:

cost_func sum(abs(obs-func))

*cost2_func sum((obs-func)**2)*

These cost functions can be used, for example, with *scipy.optimize.minimize*:

p = scipy.optimize.minimize(jams.functions.cost_f1x, np.array([p1,p2]), args=(x,y), method='Nelder-Mead', options={'disp':False})

Note the different argument orders:

curvefit needs *f(x,*args)* with the independent variable as the first argument and the parameters to fit as separate remaining arguments.

minimize is a general minimiser with respect to the first argument, i.e. *func(p,*args)*.

The module provides also two common cost functions (absolute and squared deviations) where any function in the form *func(x, p)* can be used as second argument:

cost_abs(p, func, x, y)

cost_square(p, func, x, y)

This means, for example *cost_f1x(p, x, y)* is the same as *cost_abs(p, functions.f1x_p, x, y)*. For example:

p = scipy.optimize.minimize(jams.functions.cost_abs, np.array([p1,p2]), args=(functions.f1x_p,x,y), method='Nelder-Mead', options={'disp':False})

The current functions are (the functions have the name in the first column. The seond form has a '_p' appended to the name. The cost functions, which have 'cost_' and 'cost2_' prepended to the name.):

arrhenius 1 param: Arrhenius temperature dependence of biochemical rates: *exp((T-TC25)*E/(T25*R*(T+T0)))*, parameter: E

f1x 2 params: General 1/x function: *a + b/x*

fexp 3 params: General exponential function: *a + b * exp(c*x)*

gauss 2 params: Gauss function: *1/(sig*sqrt(2*pi)) *exp(-(x-mu)**2/(2*sig**2))*, parameter: mu, sig

lasslop 6 params: Lasslop et al. (2010) a rectangular, hyperbolic light-response GPP with Lloyd & Taylor (1994) respiration and the maximum canopy uptake rate at light saturation decreases exponentially with VPD as in Koerner (1995)

line0 1 params: Straight line: *a*x*

line 2 params: Straight line: *a + b*x*

lloyd_fix 2 params: Lloyd & Taylor (1994) Arrhenius type with T0=-46.02 degC and Tref=10 degC

lloyd_only_rref 1 param: Lloyd & Taylor (1994) Arrhenius type with fixed exponential term

logistic 3 params: Logistic function: *a/(1+exp(-b(x-c)))*

logistic_offset 4 params: Logistic function with offset: *a/(1+exp(-b(x-c))) + d*

logistic2_offset 7 params: Double logistic function with offset *L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a*

poly n params: General polynomial: *c0 + c1*x + c2*x**2 + ... + cn*x**n*

sabx 2 params: sqrt(f1x), i.e. general sqrt(1/x) function: *sqrt(a + b/x)*

see 3 params: Sequential Elementary Effects fitting function: *a*(x-b)**c*

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Dec 2012 by Matthias Cuntz (mc (at) macu (dot) de)
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Added general cost functions cost_abs and cost_square, May 2013, Matthias Cuntz
- Added line0, Feb 2014, Matthias Cuntz
- Removed multiline_p, May 2020, Matthias Cuntz
- Changed to Sphinx docstring and numpydoc, May 2020, Matthias Cuntz

The following functions are provided:

`cost_abs`(p, func, x, y)	General cost function for robust optimising *func(x,p)* vs.
`cost_square`(p, func, x, y)	General cost function for optimising *func(x,p)* vs.
`arrhenius`(T, E)	Arrhenius temperature dependence of rates.
`arrhenius_p`(T, p)	Arrhenius temperature dependence of rates.
`cost_arrhenius`(p, T, rate)	Sum of absolute deviations of obs and arrhenius function.
`cost2_arrhenius`(p, T, rate)	Sum of squared deviations of obs and arrhenius.
`f1x`(x, a, b)	General 1/x function: a + b/x
`f1x_p`(x, p)	General 1/x function: a + b/x
`cost_f1x`(p, x, y)	Sum of absolute deviations of obs and general 1/x function: a + b/x
`cost2_f1x`(p, x, y)	Sum of squared deviations of obs and general 1/x function: a + b/x
`fexp`(x, a, b, c)	General exponential function: a + b * exp(c*x)
`fexp_p`(x, p)	General exponential function: a + b * exp(c*x)
`cost_fexp`(p, x, y)	Sum of absolute deviations of obs and general exponential function: a + b * exp(c*x)
`cost2_fexp`(p, x, y)	Sum of squared deviations of obs and general exponential function: a + b * exp(c*x)
`gauss`(x, mu, sig)	Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))
`gauss_p`(x, p)	Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))
`cost_gauss`(p, x, y)	Sum of absolute deviations of obs and Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))
`cost2_gauss`(p, x, y)	Sum of squared deviations of obs and Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))
`lasslop`(Rg, et, VPD, alpha, beta0, k, Rref)	Lasslop et al.
`lasslop_p`(Rg, et, VPD, p)	Lasslop et al.
`cost_lasslop`(p, Rg, et, VPD, NEE)	Sum of absolute deviations of obs and Lasslop.
`cost2_lasslop`(p, Rg, et, VPD, NEE)	Sum of squared deviations of obs and Lasslop.

Continued on next page

Table 13 – continued from previous page

line(x, a, b)	Straight line: a + b*x
line_p(x, p)	Straight line: a + b*x
cost_line(p, x, y)	Sum of absolute deviations of obs and straight line: a + b*x
cost2_line(p, x, y)	Sum of squared deviations of obs and straight line: a + b*x
line0(x, a)	Straight line through origin: a*x
line0_p(x, p)	Straight line through origin: a*x
cost_line0(p, x, y)	Sum of absolute deviations of obs and straight line through origin: a*x
cost2_line0(p, x, y)	Sum of squared deviations of obs and straight line through origin: a*x
lloyd_fix(T, Rref, E0)	Lloyd & Taylor (1994) Arrhenius type with T0=-46.02 degC and Tref=10 degC
lloyd_fix_p(T, p)	Lloyd & Taylor (1994) Arrhenius type with T0=-46.02 degC and Tref=10 degC
cost_lloyd_fix(p, T, resp)	Sum of absolute deviations of obs and Lloyd & Taylor (1994) Arrhenius type.
cost2_lloyd_fix(p, T, resp)	Sum of squared deviations of obs and Lloyd & Taylor (1994) Arrhenius type.
lloyd_only_rref(et, Rref)	If E0 is know in Lloyd & Taylor (1994) then one can calc the exponential term outside the routine and the fitting becomes linear.
lloyd_only_rref_p(et, p)	If E0 is know in Lloyd & Taylor (1994) then one can calc the exponential term outside the routine and the fitting becomes linear.
cost_lloyd_only_rref(p, et, resp)	Sum of absolute deviations of obs and Lloyd & Taylor with known exponential term.
cost2_lloyd_only_rref(p, et, resp)	Sum of squared deviations of obs and Lloyd & Taylor with known exponential term.
sabx(x, a, b)	Square root of general 1/x function: sqrt(a + b/x)
sabx_p(x, p)	Square root of general 1/x function: sqrt(a + b/x)
cost_sabx(p, x, y)	Sum of absolute deviations of obs and square root of general 1/x function: sqrt(a + b/x)
cost2_sabx(p, x, y)	Sum of squared deviations of obs and square root of general 1/x function: sqrt(a + b/x)
poly(x, *args)	General polynomial: c0 + c1*x + c2*x**2 + ...
poly_p(x, p)	General polynomial: c0 + c1*x + c2*x**2 + ...
cost_poly(p, x, y)	Sum of absolute deviations of obs and general polynomial: c0 + c1*x + c2*x**2 + ...
cost2_poly(p, x, y)	Sum of squared deviations of obs and general polynomial: c0 + c1*x + c2*x**2 + ...
cost_logistic(p, x, y)	Sum of absolute deviations of obs and logistic function L/(1+exp(-k(x-x0)))
cost2_logistic(p, x, y)	Sum of squared deviations of obs and logistic function L/(1+exp(-k(x-x0)))
cost_logistic_offset(p, x, y)	Sum of absolute deviations of obs and logistic function 1/x function: L/(1+exp(-k(x-x0))) + a
cost2_logistic_offset(p, x, y)	Sum of squared deviations of obs and logistic function 1/x function: L/(1+exp(-k(x-x0))) + a
cost_logistic2_offset(p, x, y)	Sum of absolute deviations of obs and double logistic function with offset: L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

Continued on next page

Table 13 – continued from previous page

cost2_logistic2_offset(p, x, y)	Sum of squared deviations of obs and double logistic function with offset: L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a
see(x, a, b, c)	Fit function of Sequential Elementary Effects: a * (x-b)**c
see_p(x, p)	Fit function of Sequential Elementary Effects: a * (x-b)**c
cost_see(p, x, y)	Sum of absolute deviations of obs and fit function of Sequential Elementary Effects: a * (x-b)**c
cost2_see(p, x, y)	Sum of squared deviations of obs and fit function of Sequential Elementary Effects: a * (x-b)**c

cost_abs(*p*, *func*, *x*, *y*)
General cost function for robust optimising *func(x,p)* vs. *y* with sum of absolute deviations.

Parameters

- **p** (*iterable of floats*) – parameters
- **func** (*callable*) – *fun(x,p)* -> *float*
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost_square(*p*, *func*, *x*, *y*)
General cost function for optimising *func(x,p)* vs. *y* with sum of square deviations.

Parameters

- **p** (*iterable of floats*) – parameters
- **func** (*callable*) – *fun(x,p)* -> *float*
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

arrhenius(*T*, *E*)
Arrhenius temperature dependence of rates.

Parameters

- **T** (*float or array_like of floats*) – temperature [degC]
- **E** (*float*) – activation energy [J]

Returns function value(s)

Return type float

arrhenius_p(*T*, *p*)
Arrhenius temperature dependence of rates.

Parameters

- **T** (*float or array_like of floats*) – temperature [degC]
- **p** (*iterable*) – *p[0]* is activation energy [J]

Returns function value(s)

Return type float

cost_arrhenius(*p*, *T*, *rate*)
Sum of absolute deviations of obs and arrhenius function.

Parameters

- **p** (*iterable of floats*) – *p[0]* is activation energy [J]
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_arrhenius(*p*, *T*, *rate*)
Sum of squared deviations of obs and arrhenius.

Parameters

- **p** (*iterable of floats*) – *p[0]* is activation energy [J]
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

f1x(*x*, *a*, *b*)
General 1/x function: a + b/x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter
- **b** (*float*) – second parameter

Returns function value(s)

Return type float

f1x_p(*x*, *p*)
General 1/x function: a + b/x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

Returns function value(s)

Return type float

cost_f1x(*p*, *x*, *y*)
Sum of absolute deviations of obs and general 1/x function: a + b/x

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

- **x** (*float or array_like of floats*) – independent variable

- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_f1x(*p*, *x*, *y*)
Sum of squared deviations of obs and general 1/x function: a + b/x

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

fexp(*x*, *a*, *b*, *c*)
General exponential function: a + b * exp(c*x)

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter
- **b** (*float*) – second parameter
- **c** (*float*) – third parameter

Returns function value(s)

Return type float

fexp_p(*x*, *p*)
General exponential function: a + b * exp(c*x)

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c

Returns function value(s)

Return type float

cost_fexp(*p*, *x*, *y*)
Sum of absolute deviations of obs and general exponential function: a + b * exp(c*x)

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c

- **x** (*float or array_like of floats*) – independent variable

- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_fexp(*p*, *x*, *y*)
Sum of squared deviations of obs and general exponential function: a + b * exp(c*x)

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

gauss(*x*, *mu*, *sig*)
Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **mu** (*float*) – mean
- **sig** (*float*) – width

Returns function value(s)

Return type float

gauss_p(*x*, *p*)
Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] mean

  p[1] width

Returns function value(s)

Return type float

cost_gauss(*p*, *x*, *y*)
Sum of absolute deviations of obs and Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] mean

  p[1] width

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_gauss(*p*, *x*, *y*)

Sum of squared deviations of obs and Gauss function: 1 / (sqrt(2*pi)*sig) * exp(-(x-mu)**2 / (2*sig**2))

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] mean

  p[1] width

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

lasslop(*Rg*, *et*, *VPD*, *alpha*, *beta0*, *k*, *Rref*)

Lasslop et al. (2010) is the rectangular, hyperbolic light-response of NEE as by Falge et al. (2001), where the respiration is calculated with Lloyd & Taylor (1994), and the maximum canopy uptake rate at light saturation decreases exponentially with VPD as in Koerner (1995).

Parameters

- **Rg** (*float or array_like of floats*) – Global radiation [W m-2]
- **et** (*float or array_like of floats*) – Exponential in Lloyd & Taylor: np.exp(E0*(1./(Tref-T0)-1./(T-T0))) []
- **VPD** (*float or array_like of floats*) – Vapour Pressure Deficit [Pa]
- **alpha** (*float*) – Light use efficiency, i.e. initial slope of light response curve [umol(C) J-1]
- **beta0** (*float*) – Maximum CO2 uptake rate at VPD0=10 hPa [umol(C) m-2 s-1]
- **k** (*float*) – e-folding of exponential decrease of maximum CO2 uptake with VPD increase [Pa-1]
- **Rref** (*float*) – Respiration at Tref (10 degC) [umol(C) m-2 s-1]

Returns net ecosystem exchange [umol(CO2) m-2 s-1]

Return type float

lasslop_p(*Rg*, *et*, *VPD*, *p*)

Lasslop et al. (2010) is the rectangular, hyperbolic light-response of NEE as by Falge et al. (2001), where the respiration is calculated with Lloyd & Taylor (1994), and the maximum canopy uptake rate at light saturation decreases exponentially with VPD as in Koerner (1995).

Parameters

- **Rg** (*float or array_like of floats*) – Global radiation [W m-2]
- **et** (*float or array_like of floats*) – Exponential in Lloyd & Taylor: np.exp(E0*(1./(Tref-T0)-1./(T-T0))) []
- **VPD** (*float or array_like of floats*) – Vapour Pressure Deficit [Pa]
- **p** (*iterable of floats*) – parameters (*len(p)=4*)

  p[0] Light use efficiency, i.e. initial slope of light response curve [umol(C) J-1]

  p[1] Maximum CO2 uptake rate at VPD0=10 hPa [umol(C) m-2 s-1]

  p[2] e-folding of exponential decrease of maximum CO2 uptake with VPD increase [Pa-1]

  p[3] Respiration at Tref (10 degC) [umol(C) m-2 s-1]

Returns net ecosystem exchange [umol(CO2) m-2 s-1]

Return type float

cost_lasslop(*p*, *Rg*, *et*, *VPD*, *NEE*)
Sum of absolute deviations of obs and Lasslop.

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=4*)

  p[0] Light use efficiency, i.e. initial slope of light response curve [umol(C) J-1]

  p[1] Maximum CO2 uptake rate at VPD0=10 hPa [umol(C) m-2 s-1]

  p[2] e-folding of exponential decrease of maximum CO2 uptake with VPD increase [Pa-1]

  p[3] Respiration at Tref (10 degC) [umol(C) m-2 s-1]

- **Rg** (*float or array_like of floats*) – Global radiation [W m-2]
- **et** (*float or array_like of floats*) – Exponential in Lloyd & Taylor: np.exp(E0*(1./(Tref-T0)-1./(T-T0))) []
- **VPD** (*float or array_like of floats*) – Vapour Pressure Deficit [Pa]
- **NEE** (*float or array_like of floats*) – Observed net ecosystem exchange [umol(CO2) m-2 s-1]

Returns sum of absolute deviations

Return type float

cost2_lasslop(*p*, *Rg*, *et*, *VPD*, *NEE*)
Sum of squared deviations of obs and Lasslop.

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=4*)

  p[0] Light use efficiency, i.e. initial slope of light response curve [umol(C) J-1]

  p[1] Maximum CO2 uptake rate at VPD0=10 hPa [umol(C) m-2 s-1]

  p[2] e-folding of exponential decrease of maximum CO2 uptake with VPD increase [Pa-1]

  p[3] Respiration at Tref (10 degC) [umol(C) m-2 s-1]

- **Rg** (*float or array_like of floats*) – Global radiation [W m-2]
- **et** (*float or array_like of floats*) – Exponential in Lloyd & Taylor: np.exp(E0*(1./(Tref-T0)-1./(T-T0))) []
- **VPD** (*float or array_like of floats*) – Vapour Pressure Deficit [Pa]
- **NEE** (*float or array_like of floats*) – Observed net ecosystem exchange [umol(CO2) m-2 s-1]

Returns sum of squared deviations

Return type float

line(*x*, *a*, *b*)
Straight line: a + b*x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter
- **b** (*float*) – second parameter

Returns function value(s)

Return type float

line_p(*x*, *p*)
Straight line: a + b*x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

Returns function value(s)

Return type float

cost_line(*p*, *x*, *y*)
Sum of absolute deviations of obs and straight line: a + b*x

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_line(*p*, *x*, *y*)
Sum of squared deviations of obs and straight line: a + b*x

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

line0(*x*, *a*)
Straight line through origin: a*x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter

Returns function value(s)

Return type float

line0_p(*x*, *p*)
Straight line through origin: a*x

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – *p[0]* is a

Returns function value(s)

Return type float

cost_line0(*p*, *x*, *y*)
Sum of absolute deviations of obs and straight line through origin: a*x

Parameters

- **p** (*iterable of floats*) – *p[0]* is a
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_line0(*p*, *x*, *y*)
Sum of squared deviations of obs and straight line through origin: a*x

Parameters

- **p** (*iterable of floats*) – *p[0]* is a
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

lloyd_fix(*T*, *Rref*, *E0*)
Lloyd & Taylor (1994) Arrhenius type with T0=-46.02 degC and Tref=10 degC

Parameters

- **T** (*float or array_like of floats*) – Temperature [K]
- **Rref** (*float*) – Respiration at Tref=10 degC [umol(C) m-2 s-1]
- **E0** (*float*) – Activation energy [K]

Returns Respiration [umol(C) m-2 s-1]

Return type float

lloyd_fix_p(*T*, *p*)
Lloyd & Taylor (1994) Arrhenius type with T0=-46.02 degC and Tref=10 degC

Parameters

- **T** (*float or array_like of floats*) – Temperature [K]
- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] Respiration at Tref=10 degC [umol(C) m-2 s-1]

  p[1] Activation energy [K]

Returns Respiration [umol(C) m-2 s-1]

Return type float

cost_lloyd_fix(*p*, *T*, *resp*)
Sum of absolute deviations of obs and Lloyd & Taylor (1994) Arrhenius type.

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] Respiration at Tref=10 degC [umol(C) m-2 s-1]

  p[1] Activation energy [K]

- **T** (*float or array_like of floats*) – Temperature [K]
- **resp** (*float or array_like of floats*) – Observed respiration [umol(C) m-2 s-1]

Returns sum of absolute deviations

Return type float

cost2_lloyd_fix(*p*, *T*, *resp*)
Sum of squared deviations of obs and Lloyd & Taylor (1994) Arrhenius type.

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] Respiration at Tref=10 degC [umol(C) m-2 s-1]

  p[1] Activation energy [K]

- **T** (*float or array_like of floats*) – Temperature [K]
- **resp** (*float or array_like of floats*) – Observed respiration [umol(C) m-2 s-1]

Returns sum of squared deviations

Return type float

lloyd_only_rref(*et*, *Rref*)
If E0 is know in Lloyd & Taylor (1994) then one can calc the exponential term outside the routine and the fitting becomes linear. One could also use functions.line0.

Parameters

- **et** (*float or array_like of floats*) – exp-term in Lloyd & Taylor
- **Rref** (*float*) – Respiration at Tref=10 degC [umol(C) m-2 s-1]

Returns Respiration [umol(C) m-2 s-1]

Return type float

lloyd_only_rref_p(*et*, *p*)
If E0 is know in Lloyd & Taylor (1994) then one can calc the exponential term outside the routine and the fitting becomes linear. One could also use functions.line0.

Parameters

- **et** (*float or array_like of floats*) – exp-term in Lloyd & Taylor
- **p** (*iterable of floats*) – *p[0]* is respiration at Tref=10 degC [umol(C) m-2 s-1]

Returns Respiration [umol(C) m-2 s-1]

Return type float

cost_lloyd_only_rref(*p*, *et*, *resp*)
Sum of absolute deviations of obs and Lloyd & Taylor with known exponential term.

Parameters

- **p** (*iterable of floats*) – *p[0]* is respiration at Tref=10 degC [umol(C) m-2 s-1]
- **et** (*float or array_like of floats*) – exp-term in Lloyd & Taylor
- **resp** (*float or array_like of floats*) – Observed respiration [umol(C) m-2 s-1]

Returns sum of absolute deviations

Return type float

cost2_lloyd_only_rref(*p*, *et*, *resp*)
Sum of squared deviations of obs and Lloyd & Taylor with known exponential term.

Parameters

- **p** (*iterable of floats*) – *p[0]* is respiration at Tref=10 degC [umol(C) m-2 s-1]
- **et** (*float or array_like of floats*) – exp-term in Lloyd & Taylor
- **resp** (*float or array_like of floats*) – Observed respiration [umol(C) m-2 s-1]

Returns sum of squared deviations

Return type float

sabx(*x*, *a*, *b*)
Square root of general 1/x function: sqrt(a + b/x)

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter
- **b** (*float*) – second parameter

Returns function value(s)

Return type float

sabx_p(*x*, *p*)
Square root of general 1/x function: sqrt(a + b/x)

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

Returns function value(s)

Return type float

cost_sabx(*p*, *x*, *y*)
Sum of absolute deviations of obs and square root of general 1/x function: sqrt(a + b/x)

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_sabx(*p*, *x*, *y*)
Sum of squared deviations of obs and square root of general 1/x function: sqrt(a + b/x)

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=2*)

  p[0] a

  p[1] b

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

poly(*x*, **args*)

General polynomial: c0 + c1*x + c2*x**2 + … + cn*x**n

Parameters

- **x** (*float or array_like of floats*) – independent variable
- ***args** (*float*) – parameters *len(args)=n+1*

Returns function value(s)

Return type float

poly_p(*x*, *p*)

General polynomial: c0 + c1*x + c2*x**2 + … + cn*x**n

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=n+1*)

Returns function value(s)

Return type float

cost_poly(*p*, *x*, *y*)

Sum of absolute deviations of obs and general polynomial: c0 + c1*x + c2*x**2 + … + cn*x**n

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=n+1*)
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_poly(*p*, *x*, *y*)

Sum of squared deviations of obs and general polynomial: c0 + c1*x + c2*x**2 + … + cn*x**n

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=n+1*)
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

cost_logistic(*p*, *x*, *y*)

Sum of absolute deviations of obs and logistic function L/(1+exp(-k(x-x0)))

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] L - Maximum of logistic function

  p[1] k - Steepness of logistic function

  p[2] x0 - Inflection point of logistic function

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_logistic(*p*, *x*, *y*)
Sum of squared deviations of obs and logistic function L/(1+exp(-k(x-x0)))

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] L - Maximum of logistic function

  p[1] k - Steepness of logistic function

  p[2] x0 - Inflection point of logistic function

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

cost_logistic_offset(*p*, *x*, *y*)
Sum of absolute deviations of obs and logistic function 1/x function: L/(1+exp(-k(x-x0))) + a

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=4*)

  p[0] L - Maximum of logistic function

  p[1] k - Steepness of logistic function

  p[2] x0 - Inflection point of logistic function

  p[3] a - Offset of logistic function

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_logistic_offset(*p*, *x*, *y*)
Sum of squared deviations of obs and logistic function 1/x function: L/(1+exp(-k(x-x0))) + a

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=4*)

  p[0] L - Maximum of logistic function

  p[1] k - Steepness of logistic function

  p[2] x0 - Inflection point of logistic function

  p[3] a - Offset of logistic function

- **x** (*float or array_like of floats*) – independent variable

- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

cost_logistic2_offset(*p*, *x*, *y*)

Sum of absolute deviations of obs and double logistic function with offset: L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=7*)

  p[0] L1 - Maximum of first logistic function

  p[1] k1 - Steepness of first logistic function

  p[2] x01 - Inflection point of first logistic function

  p[3] L2 - Maximum of second logistic function

  p[4] k2 - Steepness of second logistic function

  p[5] x02 - Inflection point of second logistic function

  p[6] a - Offset of double logistic function

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_logistic2_offset(*p*, *x*, *y*)

Sum of squared deviations of obs and double logistic function with offset: L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=7*)

  p[0] L1 - Maximum of first logistic function

  p[1] k1 - Steepness of first logistic function

  p[2] x01 - Inflection point of first logistic function

  p[3] L2 - Maximum of second logistic function

  p[4] k2 - Steepness of second logistic function

  p[5] x02 - Inflection point of second logistic function

  p[6] a - Offset of double logistic function

- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

see(*x*, *a*, *b*, *c*)

Fit function of Sequential Elementary Effects: a * (x-b)**c

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **a** (*float*) – first parameter
- **b** (*float*) – second parameter

- **c** (*float*) – third parameter

Returns function value(s)

Return type float

see_p(*x*, *p*)

Fit function of Sequential Elementary Effects: a * (x-b)**c

Parameters

- **x** (*float or array_like of floats*) – independent variable
- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c

Returns function value(s)

Return type float

cost_see(*p*, *x*, *y*)

Sum of absolute deviations of obs and fit function of Sequential Elementary Effects: a * (x-b)**c

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of absolute deviations

Return type float

cost2_see(*p*, *x*, *y*)

Sum of squared deviations of obs and fit function of Sequential Elementary Effects: a * (x-b)**c

Parameters

- **p** (*iterable of floats*) – parameters (*len(p)=3*)

  p[0] a

  p[1] b

  p[2] c
- **x** (*float or array_like of floats*) – independent variable
- **y** (*float or array_like of floats*) – dependent variable, observations

Returns sum of squared deviations

Return type float

4.15 hesseflux.functions.general_functions

Module with general functions that are not specialised for fitting, optimisation, sensitivity analysis, etc.

The current functions are: curvature Curvature of function f: f''/(1+f'^2)^3/2

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Mar 2015 by Matthias Cuntz (mc (at) macu (dot) de)
- Changed to Sphinx docstring and numpydoc, Dec 2019, Matthias Cuntz
- Split logistic and curvature into separate files, May 2020, Matthias Cuntz

The following functions are provided:

`curvature`(x, dfunc, d2func, *args, **kwargs)	Curvature of function:

curvature(*x*, *dfunc*, *d2func*, **args*, ***kwargs*)

Curvature of function:

f''/(1+f'^2)^3/2

Parameters

- **x** (*array_like*) – Independent variable to evalute curvature
- **dfunc** (*callable*) – Function giving first derivative of function f: f', to be called *dfunc(x, *args, **kwargs)*
- **d2func** (*callable*) – Function giving second derivative of function f: f'', to be called *d2func(x, *args, **kwargs)*
- **args** (*iterable*) – Arguments passed to *dfunc* and *d2func*
- **kwargs** (*dict*) – Keyword arguments passed to *dfunc* and *d2func*

Returns Curvature of function f at *x*

Return type float or ndarray

4.16 hesseflux.functions.logistic_function

Module with several forms of the logistic function and its first and second derivatives.

The current functions are: logistic Logistic function L/(1+exp(-k(x-x0)))

logistic_p logistic(x,*p)

dlogistic First derivative of logistic function

dlogistic_p dlogistic(x,*p)

d2logistic Second derivative of logistic function

d2logistic_p d2logistic(x,*p)

logistic_offset logistic function with offset L/(1+exp(-k(x-x0))) + a

logistic_offset_p logistic_offset(x,*p)

dlogistic_offset First derivative of logistic function with offset

dlogistic_offset_p dlogistic_offset(x,*p)

d2logistic_offset Second derivative of logistic function with offset

d2logistic_offset_p d2logistic_offset(x,*p)

logistic2_offset Double logistic function with offset L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

logistic2_offset_p logistic2_offset(x,*p)

dlogistic2_offset First derivative of double logistic function with offset

dlogistic2_offset_p dlogistic2_offset(x,*p)

d2logistic2_offset Second derivative of double logistic function with offset

d2logistic2_offset_p d2logistic2_offset(x,*p)

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Mar 2015 by Matthias Cuntz (mc (at) macu (dot) de)
- Added functions logistic_p and logistic_offset_p, Dec 2017, Matthias Cuntz
- Changed to Sphinx docstring and numpydoc, Dec 2019, Matthias Cuntz
- Distinguish iterable and array_like parameter types, Jan 2020, Matthias Cuntz
- Make systematically function_p versions of all logistic functions and its derivatives, Feb 2020, Matthias Cuntz
- Split logistic and curvature into separate files, May 2020, Matthias Cuntz

The following functions are provided:

logistic(x, L, k, x0)	Logistic function:
logistic_p(x, p)	Wrapper function for *logistic()*: *logistic(x, *p)*.
dlogistic(x, L, k, x0)	First derivative of logistic function:
dlogistic_p(x, p)	Wrapper function for *dlogistic()*: *dlogistic(x, *p)*.
d2logistic(x, L, k, x0)	Second derivative of logistic function:

Continued on next page

Table 15 – continued from previous page

d2logistic_p(x, p)	Wrapper function for d2logistic(): *d2logistic(x, *p)*.
logistic_offset(x, L, k, x0, a)	Logistic function with offset:
logistic_offset_p(x, p)	Wrapper function for logistic_offset(): *logistic_offset(x, *p)*.
dlogistic_offset(x, L, k, x0, a)	First derivative of logistic function with offset:
dlogistic_offset_p(x, p)	Wrapper function for dlogistic_offset(): *dlogistic_offset(x, *p)*.
d2logistic_offset(x, L, k, x0, a)	Second derivative of logistic function with offset
d2logistic_offset_p(x, p)	Wrapper function for d2logistic_offset(): *d2logistic_offset(x, *p)*.
logistic2_offset(x, L1, k1, x01, L2, k2, x02, a)	Double logistic function with offset:
logistic2_offset_p(x, p)	Wrapper function for logistic2_offset(): *logistic2_offset(x, *p)*.
dlogistic2_offset(x, L1, k1, x01, L2, k2, x02, a)	First derivative of double logistic function with offset:
dlogistic2_offset_p(x, p)	Wrapper function for dlogistic2_offset(): *dlogistic2_offset(x, *p)*.
d2logistic2_offset(x, L1, k1, x01, L2, k2, ...)	Second derivative of double logistic function with offset:
d2logistic2_offset_p(x, p)	Wrapper function for d2logistic2_offset(): *d2logistic2_offset(x, *p)*.

logistic(*x*, *L*, *k*, *x0*)

Logistic function:

L/(1+exp(-k(x-x0)))

Parameters

- **x** (*array_like*) – Independent variable to evalute logistic function
- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function

Returns Logistic function at *x* with maximum *L*, steepness *k* and inflection point *x0*

Return type float or ndarray

logistic_p(*x*, *p*)

Wrapper function for logistic(): *logistic(x, *p)*.

dlogistic(*x*, *L*, *k*, *x0*)

First derivative of logistic function:

L/(1+exp(-k(x-x0)))

which is

k.L/(2(cosh(k(x-x0))+1))

Parameters

- **x** (*array_like*) – Independent variable to evalute derivative of logistic function
- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function

Returns First derivative of logistic function at *x* with maximum *L*, steepness *k* and inflection point *x0*

Return type float or ndarray

dlogistic_p(*x*, *p*)
Wrapper function for `dlogistic()`: *dlogistic(x, *p)*.

d2logistic(*x*, *L*, *k*, *x0*)
Second derivative of logistic function:

L/(1+exp(-k(x-x0)))

which is

-k^2.L.sinh(k(x-x0))/(2(cosh(k(x-x0))+1)^2)

Parameters

- **x** (*array_like*) – Independent variable to evalute derivative of logistic function
- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function

Returns Second derivative of logistic function at *x* with maximum *L*, steepness *k* and inflection point *x0*

Return type float or ndarray

d2logistic_p(*x*, *p*)
Wrapper function for `d2logistic()`: *d2logistic(x, *p)*.

logistic_offset(*x*, *L*, *k*, *x0*, *a*)
Logistic function with offset:

L/(1+exp(-k(x-x0))) + a

Parameters

- **x** (*array_like*) – Independent variable to evalute logistic function
- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function
- **a** (*float*) – Offset of logistic function

Returns Logistic function at *x* with maximum *L*, steepness *k*, inflection point *x0* and offset *a*

Return type float or ndarray

logistic_offset_p(*x*, *p*)
Wrapper function for `logistic_offset()`: *logistic_offset(x, *p)*.

dlogistic_offset(*x*, *L*, *k*, *x0*, *a*)
First derivative of logistic function with offset:

L/(1+exp(-k(x-x0))) + a

which is

k.L/(2(cosh(k(x-x0))+1))

Parameters

- **x** (*array_like*) – Independent variable to evalute derivative of logistic function

- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function
- **a** (*float*) – Offset of logistic function

Returns First derivative of logistic function with offset at *x* with maximum *L*, steepness *k*, inflection point *x0*, and offset *a*

Return type float or ndarray

dlogistic_offset_p(*x*, *p*)
Wrapper function for *dlogistic_offset()*: *dlogistic_offset(x, *p)*.

d2logistic_offset(*x*, *L*, *k*, *x0*, *a*)
Second derivative of logistic function with offset

L/(1+exp(-k(x-x0))) + a

which is

-k^2.L.sinh(k(x-x0))/(2(cosh(k(x-x0))+1)^2)

Parameters

- **x** (*array_like*) – Independent variable to evalute derivative of logistic function
- **L** (*float*) – Maximum of logistic function
- **k** (*float*) – Steepness of logistic function
- **x0** (*float*) – Inflection point of logistic function
- **a** (*float*) – Offset of logistic function

Returns Second derivative of logistic function at *x* with maximum *L*, steepness *k*, inflection point *x0*, and offset *a*

Return type float or ndarray

d2logistic_offset_p(*x*, *p*)
Wrapper function for *d2logistic_offset()*: *d2logistic_offset(x, *p)*.

logistic2_offset(*x*, *L1*, *k1*, *x01*, *L2*, *k2*, *x02*, *a*)
Double logistic function with offset:

L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

Parameters

- **x** (*array_like*) – Independent variable to evalute logistic function
- **L1** (*float*) – Maximum of first logistic function
- **k1** (*float*) – Steepness of first logistic function
- **x01** (*float*) – Inflection point of first logistic function
- **L2** (*float*) – Maximum of second logistic function
- **k2** (*float*) – Steepness of second logistic function
- **x02** (*float*) – Inflection point of second logistic function
- **a** (*float*) – Offset of double logistic function

Returns Double Logistic function at *x*

Return type float or ndarray

logistic2_offset_p(*x*, *p*)
Wrapper function for `logistic2_offset()`: *logistic2_offset(x, *p)*.

dlogistic2_offset(*x*, *L1*, *k1*, *x01*, *L2*, *k2*, *x02*, *a*)
First derivative of double logistic function with offset:

L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

which is

k1.L1/(2(cosh(k1(x-x01))+1)) - k2.L2/(2(cosh(k2(x-x02))+1))

Parameters

- **x** (*array_like*) – Independent variable to evalute logistic function
- **L1** (*float*) – Maximum of first logistic function
- **k1** (*float*) – Steepness of first logistic function
- **x01** (*float*) – Inflection point of first logistic function
- **L2** (*float*) – Maximum of second logistic function
- **k2** (*float*) – Steepness of second logistic function
- **x02** (*float*) – Inflection point of second logistic function
- **a** (*float*) – Offset of double logistic function

Returns First derivative of double logistic function with offset at *x*

Return type float or ndarray

dlogistic2_offset_p(*x*, *p*)
Wrapper function for `dlogistic2_offset()`: *dlogistic2_offset(x, *p)*.

d2logistic2_offset(*x*, *L1*, *k1*, *x01*, *L2*, *k2*, *x02*, *a*)
Second derivative of double logistic function with offset:

L1/(1+exp(-k1(x-x01))) - L2/(1+exp(-k2(x-x02))) + a

which is

-k1^2.L1.sinh(k1(x-x01))/(2(cosh(k1(x-x01))+1)^2) +k2^2.L2.sinh(k2(x-x02))/(2(cosh(k2(x-x02))+1)^2)

Parameters

- **x** (*array_like*) – Independent variable to evalute logistic function
- **L1** (*float*) – Maximum of first logistic function
- **k1** (*float*) – Steepness of first logistic function
- **x01** (*float*) – Inflection point of first logistic function
- **L2** (*float*) – Maximum of second logistic function
- **k2** (*float*) – Steepness of second logistic function
- **x02** (*float*) – Inflection point of second logistic function
- **a** (*float*) – Offset of double logistic function

Returns Second derivative of double logistic function with offset at *x*

Return type float or ndarray

d2logistic2_offset_p(*x*, *p*)
Wrapper function for `d2logistic2_offset()`: *d2logistic2_offset(x, *p)*.

4.17 hesseflux.functions.opti_test_functions

Module provides common test functions for parameter estimation and optimisation algorithms such as Rosenbrock and Griewank functions.

Current functions are: ackley >=2 params: Ackley function, global optimum: 0.0 at origin

goldstein_price 2 params: Goldstein-Price function, global optimum: 3.0 (0.0,-1.0)

griewank 2 or 10 params: Griewank function, global optimum: 0 at origin

rastrigin 2 params: Rastrigin function, global optimum: -2 (0,0)

rosenbrock 2 params: Rosenbrock function, global optimum: 0 (1,1)

six_hump_camelback 2 params: Six-hump Camelback function True Optima: -1.031628453489877 (-0.08983,0.7126) and (0.08983,-0.7126)

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Oct 2013 by Matthias Cuntz (mc (at) macu (dot) de)
- Rearrange function library, Mar 2015, Matthias Cuntz
- Changed to Sphinx docstring and numpydoc, May 2020, Matthias Cuntz

The following functions are provided:

ackley(x)	Ackley function (>= 2-D).
griewank(x)	Griewank function (2-D or 10-D).
goldstein_price(x)	Goldstein-Price function (2-D).
rastrigin(x)	Rastrigin function (2-D).
rosenbrock(x)	Rosenbrock function (2-D).
six_hump_camelback(x)	Six-hump Camelback function (2-D).

ackley(*x*)

Ackley function (>= 2-D).

Global Optimum: 0.0, at origin.

Parameters **x** (*array*) – multi-dimensional x-values (len(x) >= 2)

Returns Value of Ackley function.

Return type float

griewank(*x*)

Griewank function (2-D or 10-D).

Global Optimum: 0.0, at origin.

Parameters **x** (*array*) – multi-dimensional x-values.

len(x)=2 or *len(x)=10*.

x[i] bound to [-600,600] for all i.

Returns Value of Griewank function.

Return type float

goldstein_price(*x*)

Goldstein-Price function (2-D).

Global Optimum: 3.0, at (0.0,-1.0).

Parameters **x** (*array*) – 2 x-values. *len(x)=2*.

x[i] bound to [-2,2] for i=1 and 2.

Returns Value of Goldstein-Price function.

Return type float

rastrigin(*x*)

Rastrigin function (2-D).

Global Optimum: -2.0, at origin.

Parameters **x** (*array*) – 2 x-values. *len(x)=2*.

x[i] bound to [-1,1] for i=1 and 2.

Returns Value of Rastrigin function.

Return type float

rosenbrock(*x*)

Rosenbrock function (2-D).

Global Optimum: 0.0, at (1.0,1.0).

Parameters **x** (*array*) – 2 x-values. *len(x)=2*.

x[1] bound to [-5,5].

x[2] bound to [-2,8].

Returns Value of Rosenbrock function.

Return type float

six_hump_camelback(*x*)

Six-hump Camelback function (2-D).

Global Optima: -1.031628453489877, at (-0.08983,0.7126) and (0.08983,-0.7126).

Parameters **x** (*array*) – 2 x-values. *len(x)=2*.

x[i] bound to [-5,5] for i=1 and 2.

Returns Value of Six-hump Camelback function.

Return type float

4.18 hesseflux.functions

Purpose

functions provides a variety of special functions, including common test functions for parameter estimations such as Rosenbrock and Griewank, test functions for parameter sensitivity analysis such as the Ishigami and Homma function, several forms of the logistic function and its first and second derivatives, and a variety of functions together with robust and square cost functions to use with scipy.optimize package.

The module is part of the JAMS Python package https://github.com/mcuntz/jams_python It will be synchronised with the JAMS package irregularily if used in other packages.

Subpackages

general_functions	Module with general functions that are not specialised for fitting, optimisation, sensitivity analysis, etc.
fit_functions	Module defines common functions that are used in curve_fit or fmin parameter estimations.
logistic_function	Module with several forms of the logistic function and its first and second derivatives.
opti_test_functions	Module provides common test functions for parameter estimation and optimisation algorithms such as Rosenbrock and Griewank functions.
sa_test_functions	Module provides test functions for parameter sensitivity analysis from

4.19 hesseflux.functions.sa_test_functions

Module provides test functions for parameter sensitivity analysis from

Ishigami and Homma (1990) An importance qualification technique in uncertainty analysis for computer models, Proceedings of the isuma '90, First International Symposium on Uncertainty Modelling and Analysis, University of Maryland, Dec. 03 - Dec 05 1990, 398-403

Oakley and O'Hagan (2004) Probabilistic sensitivity analysis of complex models: a Bayesian approach

J. R. Statist. Soc. B 66, Part 3, 751-769.

Morris (1991) Factorial sampling plans for preliminary computational experiments, Technometrics 33, 161-174.

Saltelli et al. (2008) Global Sensitivity Analysis. The Primer, John Wiley & Sons, pp. 292

Saltelli et al. (2010) Variance based sensitivity analysis of model output, Design and estimator for the total sensitivity index, Comp. Phys. Comm. 181, 259-270.

Sobol' (1990), Sensitivity estimates for nonlinear mathematical models, Matematicheskoe Modelirovanie 2, 112-118 (in Russian), translated in English in Sobol' (1993).

Sobol' (1993) Sensitivity analysis for non-linear mathematical models, Mathematical Modelling and Computational Experiment 1, 407-414, English translation of Russian original paper Sobol' (1990).

Current functions are: B B of Saltelli et al. (2010)

G / g G-function attributed to Sobol' (1990, 1993), given by Saltelli et al. (2008, 2010)

Gstar G* of Saltelli et al. (2010)

ishigami_homma Ishigami and Homma (1990), given by Saltelli et al. (2008, page 179)

K / bratley K of Saltelli et al. (2010)

fmorris / morris After Morris (1991)

oakley_ohagan Oakley and O'Hagan (2004), parameters given in Saltelli et al. (2008) or on http://www.jeremy-oakley.staff.shef.ac.uk/psa_example.txt

This module was written by Matthias Cuntz & Juliane Mai while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Mar 2015 by Matthias Cuntz (mc (at) macu (dot) de) & Juliane Mai
- Added functions to properly test PAWN method: linear, product, ratio, and ishigami_homma_easy, Dec 2017, Juliane Mai
- Provide morris function under the name fmorris and the K function under the name bratley, Nov 2019, Matthias Cuntz
- Changed to Sphinx docstring and numpydoc, Dec 2019, Matthias Cuntz
- Distinguish iterable and array_like parameter types, Jan 2020, Matthias Cuntz

The following functions are provided:

B(X)	B function, Saltelli et al.
g(X, a)	G-function
G(X, a)	G-function
Gstar(X, alpha, delta, a)	G* example, Saltelli et al.

Continued on next page

Table 18 – continued from previous page

K(X)	K example, Saltelli et al.
bratley(*args)	K example, Saltelli et al.
fmorris(X, beta0, beta1, beta2, beta3, beta4)	Morris-function, Morris (1991) Technometrics 33, 161-174
morris(*args)	Morris-function, Morris (1991) Technometrics 33, 161-174
oakley_ohagan(X)	Oakley and O'Hagan (2004) J.
ishigami_homma(X, a, b)	Ishigami and Homma (1990), given by Saltelli et al.
linear(X, a, b)	Linear test function to test PAWN method:
product(X)	Product test function to test PAWN method:
ratio(X)	Ratio test function:
ishigami_homma_easy(X)	Simplified Ishigami and Homma function to test PAWN method:

B (*X*)

B function, Saltelli et al. (2010) Comp. Phys. Comm. 181, p. 259-270

Parameters **X** (*array_like*) – (nX,) or (nX,npoints) array of floats

Returns **B** – float or (npoints,) floats of B function values at *X*

Return type float or ndarray

g (*X*, *a*)

G-function

Sobol' (1990) Matematicheskoe Modelirovanie 2, 112-118 (in Russian)

Sobol' (1993) Mathematical Modelling and Computational Experiment 1, 407-414 (English translation)

Parameters

- **X** (*array_like*) – (nX,) or (nX,npoints) array of floats
- **a** (*array_like*) – (nX,) array of floats

Returns **G** – float or (npoints,) floats of G function values at *X* with parameters *a*

Return type float or ndarray

G (*X*, *a*)

G-function

Sobol' (1990) Matematicheskoe Modelirovanie 2, 112-118 (in Russian)

Sobol' (1993) Mathematical Modelling and Computational Experiment 1, 407-414 (English translation)

Parameters

- **X** (*array_like*) – (nX,) or (nX,npoints) array of floats
- **a** (*array_like*) – (nX,) array of floats

Returns **g** – float or (npoints,) floats of G function values at *X* with parameters *a*

Return type float or ndarray

Gstar (*X*, *alpha*, *delta*, *a*)

G* example, Saltelli et al. (2010) Comp. Phys. Comm., 181, p. 259-270

Parameters

- **X** (*array_like*) – (nX,) or (nX,npoints) array of floats
- **alpha** (*array_like*) – (nX,) array of floats
- **delta** (*array_like*) – (nX,) array of floats
- **a** (*array_like*) – (nX,) array of floats

Returns **G*** – float or (npoints,) floats of G* function values at *X* with parameters *alpha*, *delta* and *a*

Return type float or ndarray

K(*X*)
K example, Saltelli et al. (2010) Comp. Phys. Comm., 181, p. 259-270

Parameters **X** (*array_like*) – (nX,) or (nX,npoints) array of floats

Returns **K** – float or (npoints,) floats of K function values at *X*

Return type float or ndarray

bratley(**args*)
K example, Saltelli et al. (2010) Comp. Phys. Comm., 181, p. 259-270

Parameters **X** (*array_like*) – (nX,) or (nX,npoints) array of floats

Returns **bratley** – float or (npoints,) floats of K function values at *X*

Return type float or ndarray

fmorris(*X*, *beta0*, *beta1*, *beta2*, *beta3*, *beta4*)
Morris-function, Morris (1991) Technometrics 33, 161-174

Parameters

- **X** (*array_like*) – (20,) or (20,npoints) array of floats
- **beta0** (*float*) – float
- **beta1** (*array_like*) – (20,) array of floats
- **beta2** (*array_like*) – (20,20) array of floats
- **beta3** (*array_like*) – (20,20,20) array of floats
- **beta4** (*array_like*) – (20,20,20,20) array of floats

Returns **fmorris** – float or (npoints,) floats of Morris function values at *X* with parameters *beta0-beta4*

Return type float or ndarray

morris(**args*)
Morris-function, Morris (1991) Technometrics 33, 161-174

Parameters

- **X** (*array_like*) – (20,) or (20,npoints) array of floats
- **beta0** (*float*) – float
- **beta1** (*array_like*) – (20,) array of floats
- **beta2** (*array_like*) – (20,20) array of floats
- **beta3** (*array_like*) – (20,20,20) array of floats
- **beta4** (*array_like*) – (20,20,20,20) array of floats

Returns **morris** – float or (npoints,) floats of Morris function values at *X* with parameters *beta0-beta4*

Return type float or ndarray

oakley_ohagan(*X*)
Oakley and O'Hagan (2004) J. R. Statist. Soc. B 66, Part 3, 751-769

Parameters **X** (*array_like*) – (15,) or (15,npoints) array of floats

Returns **oakley_ohagan** – float or (npoints,) floats of Oakley and O'Hagan function values at *X*

Return type float or ndarray

ishigami_homma(*X*, *a*, *b*)
Ishigami and Homma (1990), given by Saltelli et al. (2008, page 179)

Parameters

- **X** (*array_like*) – (3,) or (3,npoints) array of floats
- **a** (*array_like*) – float or (npoints,) array of floats
- **b** (*array_like*) – float or (npoints,) array of floats

Returns **ishigami_homma** – float or (npoints,) floats of Ishigami and Homma function values at *X* with parameters *a* and *b*

Return type float or ndarray

linear(*X*, *a*, *b*)
Linear test function to test PAWN method:

*Y = a*X + b*

Parameters

- **X** (*array_like*) – (1,) or (1,npoints) array of floats
- **a** (*array_like*) – float or (npoints,) array of floats
- **b** (*array_like*) – float or (npoints,) array of floats

Returns **linear** – float or (npoints,) floats of linear function values at *X* with parameters *a* and *b*

Return type float or ndarray

product(*X*)
Product test function to test PAWN method:

*Y = X[0] * X[1]*

Parameters **X** (*array_like*) – (2,) or (2,npoints) array of floats

Returns **product** – float or (npoints,) floats of product function values at *X*

Return type float or ndarray

ratio(*X*)
Ratio test function:

Y = X[0] / X[1]

Simple nonlinear model proposed by Liu et al. (2006):

Liu, H., Sudjianto, A., Chen, W., 2006. Relative entropy based method for probabilistic sensitivity analysis in engineering design. J. Mech. Des. 128, 326-336.

Used by Pianosi & Wagener, Environmental Modelling & Software (2015)

Pianosi, F. & Wagener T., 2015 A simple and efficient method for global sensitivity analysis based on cumulative distribution functions. Environmental Modelling & Software 67, 1-11.

Parameters **X** (*array_like*) – (2,) or (2,npoints) array of floats

Returns **ratio** – float or (npoints,) floats of ratio function values at *X*

Return type float or ndarray

ishigami_homma_easy(*X*)
Simplified Ishigami and Homma function to test PAWN method:

Y = sin(X[0]) + X[1]

with *X[0],X[1] ~ Uniform[-Pi,Pi]*

Parameters **x** (*array_like*) – (2,) or (2,npoints) array of floats

Returns **ishigami_homma_easy** – float or (npoints,) floats of simplified Ishigami and Homma function values at *X*

Return type float or ndarray

4.20 hesseflux.gapfill

gapfill [Fills gaps of flux data from Eddy covariance measurements according to] Reichstein et al. (Global Change Biology, 2005) or estimate flux uncertainties after Lasslop et al. (Biogeosciences, 2008).

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

Copyright (c) 2012-2020 Matthias Cuntz - mc (at) macu (dot) de Released under the MIT License; see LICENSE file for details.

- Written Mar 2012 by Matthias Cuntz - mc (at) macu (dot) de
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Input data can be ND-array, Apr 2014, Matthias Cuntz
- Bug in longestmarginalgap: was only working at time series edges, rename it to longgap, Apr 2014, Matthias Cuntz
- Keyword fullday, Apr 2014, Matthias Cuntz
- Input can be pandas Dataframe or numpy array(s), Apr 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

gapfill(dfin[, flag, date, timeformat, ...])	Fills gaps in flux data from Eddy covariance measurements with Marginal Distribution Sampling (MDS) according to Reichstein et al.

gapfill(*dfin*, *flag=None*, *date=None*, *timeformat='%Y-%m-%d %H:%M:%S'*, *colhead=None*, *sw_dev=50.0*, *ta_dev=2.5*, *vpd_dev=5.0*, *longgap=60*, *fullday=False*, *undef=-9999*, *ddof=1*, *err=False*, *verbose=0*)

Fills gaps in flux data from Eddy covariance measurements with Marginal Distribution Sampling (MDS) according to Reichstein et al. (Global Change Biology, 2005).

This means, if there is a gap in the data, look for similar meteorological conditions (defined as maximum possible deviations) in a certain time window and fill with the average of these 'similar' values.

The routine can also do the same search for similar meteorological conditions for every data point and calculate its standard deviation as a measure of uncertainty after Lasslop et al. (Biogeosciences, 2008).

Parameters

- **dfin** (*pandas.Dataframe or numpy.array*) – time series of fluxes to fill as well as meteorological variables incoming short-wave radiation, air temperature, air vapour pressure deficit.

  dfin can be a pandas.Dataframe with the columns 'SW_IN' (or starting with 'SW_IN') for incoming short-wave radiation [W m-2] 'TA' (or starting with 'TA_') for air temperature [deg C] 'VPD' (or starting with 'VPD') for air vapour deficit [hPa] and columns with ecosystem fluxes with possible missing values (gaps). The index is taken as date variable.

  dfin can also me a numpy array with the same columns. In this case *colhead*, *date*, and possibly *dateformat* must be given.

- **flag** (*pandas.Dataframe or numpy.array, optional*) – flag Dataframe or array has the same shape as dfin. Non-zero values in *flag* will be treated as missing values in *dfin*.

  flag must follow the same rules as *dfin* if pandas.Dataframe.

If *flag* is numpy array, *df.columns.values* will be used as column heads and the index of *dfin* will be copied to *flag*.

- **date** (*array_like of string, optional*) – 1D-array_like of calendar dates in format given in *timeformat*.

  date must be given if *dfin* is numpy array.

- **timeformat** (*str, optional*) – Format of dates in *date*, if given (default: '%Y-%m-%d %H:%M:%S'). See strftime documentation of Python's datetime module: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

- **colhed** (*array_like of str, optional*) – column names if *dfin* is numpy array. See *dfin* for mandatory column names.

- **sw_dev** (*float, optional*) – threshold for maximum deviation of global radiation (default: 50)

- **ta_dev** (*float, optional*) – threshold for maximum deviation of air Temperature (default: 2.5)

- **vpd_dev** (*float, optional*) – threshold for maximum deviation of vpd (default: 5.)

- **longgap** (*int, optional*) – avoid extraploation into a gap longer than *longgap* days (default: 60)

- **fullday** (*bool, optional*) –

  True: move beginning of large gap to start of next day and move end of large gap to end of last day (default: False)

- **undef** (*float, optional*) – values having *undef* value are treated as missing values in *dfin* (default: -9999)

  np.nan is not allowed (working).

- **ddof** (*int, optional*) – Delta Degrees of Freedom. The divisor used in calculation of standard deviation for error estimates (*err=True*) is `N-ddof`, where `N` represents the number of elements (default: 1).

- **err** (*bool, optional*) – True: fill every data point, i.e. used for error generation (default: False)

- **shape** (*bool or tuple, optional*) –

  True: output have the same shape as input data if *dfin* is numpy array; if a tuple is given, then this tuple is used to reshape.

  False: outputs are 1D arrays if *dfin* is numpy array (default: False).

- **verbose** (*int, optional*) – Verbosity level 0-3 (default: 0). 0 is no output; 3 is very verbose.

Returns

If *err==False*: filled_data, quality_class

If *err==True*: err_data

pandas.Dataframe(s) will be returned if *dfin* was Dataframe.

numpy array(s) will be returned if *dfin* was numpy array.

Return type pandas.Dataframe(s) or numpy array(s)

Notes

If *err==True*, there is no error estimate if there are no meteorological conditions in the vicinity of the data point.

Routine Does not work with *undef=np.nan*.

Reichstein et al. (2005) On the separation of net ecosystem exchange into assimilation and ecosystem respiration: review and improved algorithm. Global Change Biology 11, 1424-1439

Examples

```
>>> import numpy as np
>>> from fread import fread
>>> from date2dec import date2dec
>>> from dec2date import dec2date
>>> ifile = 'test_gapfill.csv' # Tharandt 1998 = Online tool example file
>>> undef = -9999.
>>> # data
>>> dat   = fread(ifile, skip=2, transpose=True)
>>> ndat  = dat.shape[1]
>>> head  = fread(ifile, skip=2, header=True)
>>> head1 = head[0]
>>> # colhead
>>> idx   = []
>>> for i in head1:
...     if i in ['NEE', 'LE', 'H', 'Rg', 'Tair', 'VPD']: idx.append(head1.
↪index(i))
>>> colhead = ['FC', 'LE', 'H', 'SW_IN', 'TA', 'VPD']
>>> # data
>>> dfin = dat[idx,:]
>>> # flag
>>> flag = np.where(dfin == undef, 2, 0)
>>> flag[0,:] = dat[head1.index('qcNEE'),:].astype(np.int)
>>> flag[1,:] = dat[head1.index('qcLE'),:].astype(np.int)
>>> flag[2,:] = dat[head1.index('qcH'),:].astype(np.int)
>>> flag[np.where(flag==1)] = 0
>>> # date
>>> day_id  = head1.index('Day')
>>> hour_id = head1.index('Hour')
>>> ntime   = dat.shape[1]
>>> year  = np.ones(ntime, dtype=np.int) * 1998
>>> hh    = dat[hour_id,:].astype(np.int)
>>> mn    = np.rint((dat[hour_id,:]-hh)*60.).astype(np.int)
>>> y0    = date2dec(yr=year[0], mo=1, dy=1, hr=hh, mi=mn)
>>> jdate = y0 + dat[day_id,:]
>>> adate = dec2date(jdate, eng=True)
>>> # fill
>>> dat_f, flag_f = gapfill(dfin, flag=flag, date=adate, colhead=colhead,␣
↪undef=undef, verbose=0)
>>> print('{:d} {:d} {:d} {:d} {:d} {:d}'.format(*flag_f[0,11006:11012]))
1 1 1 2 2 2
>>> print('{:.2f} {:.2f} {:.2f} {:.2f} {:.2f} {:.2f}'.format(*dat_f[0,
↪11006:11012]))
-18.68 -15.63 -19.61 -15.54 -12.40 -15.33
```

```
>>> # 1D err
>>> dat_std = gapfill(dfin, flag=flag, date=adate, colhead=colhead,␣
↪undef=undef, verbose=0, err=True)
>>> print('{:.3f} {:.3f} {:.3f} {:.3f} {:.3f} {:.3f}'.format(*dat_std[0,
↪11006:11012]))
5.372 13.118 6.477 -9999.000 -9999.000 -9999.000
```

```
>>> dat_err     = np.ones(ndat, dtype=np.int)*(-1)
>>> kk          = np.where((dat_std[0,:] != undef) & (dat_f[0,:] != 0.))[0]
```

(continues on next page)

(continued from previous page)

```
>>> dat_err[kk] = np.abs(dat_std[0,kk]/dat_f[0,kk]*100.).astype(np.int)
>>> print('{:d} {:d} {:d} {:d} {:d} {:d}'.format(*dat_err[11006:11012]))
28 83 33 -1 -1 -1
```

4.21 hesseflux.logtools.logtools

logtools is a Python port of the Control File Functions of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

Some functions are renamed compared to the original logger tools:

chs -> varchs

add -> varadd

sub -> varsub

mul -> varmul

div -> vardiv

sqr -> varsqr/varsqrt

exp -> varexp

log -> varlog

pot -> varpot

Not all functions are implemented (yet). Missing functions are:

varset

met_torad

met_psy_rh

met_dpt_rh

write

Some functions are slightly enhanced, which is reflected in the documentation of the indidual functions.

All functions have an additional keyword *undef*, which defaults to -9999.: elements are excluded from the calculations if any of the inputs equals *undef*.

Only bit_test and the if-statements *ifeq*, *ifne*, *ifle*, *ifge*, *iflt*, *igt* do not have the *undef* keyword.

The Looger Tools control functions are:

1. Assignment # not implemented
2. **Change sign** x = varchs(a) means x = -a, where a is a variable or a number.

   def varchs(var1, undef=-9999.):
3. **Addition** x = varadd(a, b) means x = a + b, where a and b are ndarray or float.

   def varadd(var1, var2, undef=-9999.):
4. **Subtraction** x = varsub(a, b) means x = a - b, where a and b are ndarray or float.

   def varsub(var1, var2, undef=-9999.):
5. **Multiplication** x = varmul(a, b) means x = a * b, where a and b are ndarray or float.

   def varmul(var1, var2, undef=-9999.):
6. **Division** x = vardiv(a, b) means x = a/b, where a and b are ndarray or float.

   def vardiv(var1, var2, undef=-9999.):
7. **Square root** x = varsqr(a) means x = sqrt(a), where a is a variable or a number. x = varsqrt(a) means x = sqrt(a), where a is a variable or a number.

   def varsqr(var1, undef=-9999.): def varsqrt(var1, undef=-9999.):

8. **Exponentiation of e** x = varexp(a) means x = exp(a), where a is a variable or a number.

   def varexp(var1, undef=-9999.):

9. **Natural logarithm** x = varlog(a) means x = ln(a), where a is a variable or a number.

   def varlog(var1, undef=-9999.):

10. Exponentiation x = varpot(a, b) means x = a**b, where a and b are ndarray or float.

    def varpot(var1, var2, undef=-9999.):

11. Apply linear function x = lin(y, a0, a1) means x = a0 + a1 * y, where a0 and a1 are ndarray or float.

    def lin(var1, a, b, undef=-9999.):

12. Apply 2nd order function x=quad(y,a0,a1,a2) means x = a0 +a1*y + a2*y**2, where a0, a1 and a2 are ndarray or float.

    def quad(var1, a, b, c, undef=-9999.):

13. Apply 3rd order function x=cubic(y,a0,a1,a2,a3) means x = a0 +a1*y+a2*y**2+a3*y**3, where a0, a1, a2 and a3 are ndarray or float.

    def cubic(var1, a, b, c, d, undef=-9999.):

14. Calculate fraction of day from hours, minutes and seconds x = hms(h, m, s) means x = (h + m/60 + s/3600)/24, where h, m and s (hours, minutes and seconds) are ndarray or float.

    def hms(h, m, s, undef=-9999.):

15. Bitwise test x = bit_test(y, b, start=0) means x = 1 if bit b ist set in y otherwise x = 0. Returns a list of b is an array. Counting of b starts at start. For the behaviour of the original logger tools, set start=1. Negative b's is not implemented.

    def bit_test(var1, var2, start=0):

16. Replacement of underflows by new value x = setlow(y,lo,ln=None) means IF (y > lo) THEN x = ln ELSE x = y, where lo and ln are ndarray or float. ln is optional. If not given lo will be used. This function may be used to adjust small negative values of short wave radiation during nighttime to zero values.

    def setlow(dat, low, islow=None, undef=-9999.):

17. Replacement of overflows by new value x = sethigh(y,lo,ln=None) means IF (y < lo) THEN x = ln ELSE x = y, where lo and ln are ndarray or float. ln is optional. If not given lo will be used. This function may be used to adjust relative humidity values of a little bit more than 100 % to 100 %.

    def sethigh(dat, high, ishigh=None, undef=-9999.):

18. Replacement of underflows or overflows by the undef x = limits(y, ll, lh) means IF (y > ll) OR (y < lh) THEN x = undef ELSE x = y, where ll and lh are ndarray or float. This function may be used to check values lying in between certain limits. If one of the limits is exceeded the value is set to undef.

    def limits(dat, mini, maxi, undef=-9999.):

19. Calculation of mean value x = mean(y1, y2, ..., yn) means x = (y1 + y2 + ... + yn)/n, where y1, y2, ..., yn are ndarray or float.

    def mean(var1, axis=None, undef=-9999.):

20. Calculation of minimum value x = mini(y1,y2,...,yn) means x = min(y1,y2,...,yn), where y1, y2, ..., yn are ndarray or float.

    def mini(var1, axis=None, undef=-9999.):

21. Calculation of maximum value x = maxi(y1,y2,...,yn) means x = max(y1,y2,...,yn), where y1, y2, ..., yn are ndarray or float.

    def maxi(var1, axis=None, undef=-9999.):

22. Calculation of total radiation from net radiometer # no implemented

23. Calculation of long wave radiation from net radiometer x = met_lwrad(y, Tp) where y is the output voltage of the net radiometer in mV, Tp is the temperature of the net radiometer body in degC. The total radiation in W m-2 is calculated according to the following formula: x=y*fl +sigma*(Tp +273.16)**4 where sigma = 5.67051 * 10**8 W m-2 K-4 is the Stephan-Boltzmann-Constant and fl is the factor for long wave radiation (reciprocal value of sensitivity) in W m-2 per mV. The function assumes that fl was already applied before. All parameters may be ndarray or float.

    def met_lwrad(dat, tpyr, undef=-9999.): # assumes that dat was already multiplied with calibration factor

24. Calculation of radiation temperature from long wave radiation x = met_trad(Rl, epsilon) where Rl is the long wave radiation in W m-2, epsilon is the long wave emissivity of the surface (between 0 and 1). The radiation temperature in degC is calculated according to the following formula: x= sqrt4(Rl/(sigma*epsilon)) - 273.16 where sigma = 5.67051 * 108 W m-2 K-4 is the Stephan-Boltzmann-Constant. Both parameters may be ndarray or float.

    def met_trad(dat, eps, undef=-9999.):

25. Calculation of albedo from short wave downward and upward radiation x = met_alb(Rsd, Rsu) where Rsd is the short wave downward radiation in Wm-2, Rsu is the short wave upward radiation in Wm-2, The albedo in % is calculated according to the following formula: x = 100 * (Rsu / Rsd) If Rsd > 50 W m-2 or Rsu > 10 W m-2 the result is undef. Both parameters may be ndarray or float.

    def met_alb(swd, swu, swdmin=50., swumin=10., undef=-9999.):

26. Calculation of albedo from short wave downward and upward radiation with limits x=met_albl(Rsd,Rsu,Rsd_limit,Rsu_limit)where Rsd is the short wave downward radiation in Wm-2, Rsu is the short wave upward radiation in Wm-2, Rsd_limit is the short wave downward radiation limit in Wm-2, Rsu_limit is the short wave upward radiation limit in Wm-2, The albedo in % is calculated according to the following formula: x = 100 * (Rsu / Rsd) If Rsd > Rsd_limit or Rsu > Rsu_limit the result is undef. All four parameters may be ndarray or float.

    def met_albl(swd, swu, swdmin, swumin, undef=-9999.):

27. Calculation of saturation water vapour pressure x = met_vpmax(T) where T is the air temperature in degC. The saturation water vapour pressure in mbar (hPa) is calculated according to the following formula: x = 6.1078 * exp(17.08085 * T / (234.175 + T)) The parameter may be a variable or a number.

    def met_vpmax(temp, undef=-9999.):

28. Calculation of actual water vapour pressure x = met_vpact(T,rh) where T is the air temperature in degC, rh is the relative humidity in %. The actual water vapour pressure in mbar (hPa) is calculated according to the following for- mulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T)) x = Es * rh/100 Both parameters may be ndarray or float.

    def met_vpact(temp, rh, undef=-9999.):

29. Calculation of water vapour pressure deficit x = met_vpdef(T, rh) where T is the air temperature in degC, rh is the relative humidity in %. The water vapour pressure deficit in mbar (hPa) is calculated according to the following for- mulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T)) E = Es * rh/100 x = Es - E Both parameters may be ndarray or float.

    def met_vpdef(temp, rh, undef=-9999.):

30. Calculation of specific humidity x = met_sh(T, rh, p) where T is the air temperature in degC, rh is the relative humidity in %, p is the air pressure in mbar (hPa). The specific humidity in g kg-1 is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T)) E = Es * rh/100 x = 622 * E/(p-0.378*E) All parameters may be ndarray or float.

    def met_sh(temp, rh, p, undef=-9999.):

31. Calculation of potential temperature x = met_tpot(T, p) where T is the air temperature in degC, p is the air pressure in mbar (hPa). The potential temperature in K is calculated according to the following formula: x = (T + 273.16) * (1000/p)**0.286 Both parameters may be ndarray or float.

    def met_tpot(temp, p, undef=-9999.):

32. Calculation of air density x = met_rho(T, rh, p) where T is the air temperature in degC, rh is the relative humidity in %, p is the air pressure in mbar (hPa). The air density in kg m-3 is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T)) E = Es * rh/100 sh = 622 * E/(p-0.378*E) Tv = ((T + 273.16) * (1 + 0.000608 * sh)) - 273.16 x = p * 100 / (287.05 * (Tv + 273.16)) All parameters may be ndarray or float.

    def met_rho(temp, rh, p, undef=-9999.):

33. Calculation of dew point temperature x = met_dpt(T, rh) where T is the air temperature in degC, rh is the relative humidity in %. The dew point temperature in degC is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/(234.175 + T)) E = Es * rh/100 x = 234.175 * ln(E/6.1078)/(17.08085 - ln(E/6.1078)) Both parameters may be ndarray or float.

    def met_dpt(temp, rh, undef=-9999.):

34. Calculation of water vapour concentration x = met_h2oc(T, rh, p) where T is the air temperature in degC, rh is the relative humidity in %, p is the air pressure in mbar (hPa). The water vapour concentration in mmol mol-1 is calculated according to the following formu- las: Es = 6.1078*exp(17.08085*T/ (234.175 + T)) E = Es * rh/100 x = 0.1 * E /(0.001*p*100*0.001) All parameters may be ndarray or float.

    def met_h2oc(temp, rh, p, undef=-9999.):

35. Calculation of relative humidity from dry and wet bulb temperature # not implemented

36. Calculation of relative humidity from dew point temperature # not implemented

37. Calculation of relative humidity from water vapour concentration x = met_h2oc_rh(T, [H2O], p) where T is the air temperature in degC, [H2O] is the water vapour concentration in mmolmol-1, p is the air pressure in mbar (hPa). The relative humidity in % is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/(234.175 + T)) E = 10 * [H2O] * 0.001 * p * 100 * 0.001 x = 100 * E / Es All parameters may be ndarray or float.

    def met_h2oc_rh(temp, h, p, undef=-9999.):

38. Rotation of wind direction x = met_wdrot(wd, a) where wd is the wind direction in degree, a is the rotation angle in degree (positive is clockwise). The rotated wind direction is calculated according to the following formulas: x = wd + a IF x > 0 THEN x = x + 360 IF x >= 360 THEN x = x - 360 Both parameters may be ndarray or float.

    def met_wdrot(wd, a, undef=-9999.):

39. Rotation of u-component of wind vector x = met_urot(u, v, a) where u is the u-component of the wind vector, v is the v-component of the wind vector, a is the rotation angle in degree (positive is clockwise). The rotated u-component is calculated according to the following formula: x = u * cos (a) + v * sin (a) All three parameters may be ndarray or float.

    def met_urot(u, v, a, undef=-9999.):

40. Rotation of v-component of wind vector x = met_vrot(u, v, a) where u is the u-component of the wind vector, v is the v-component of the wind vector, a is the rotation angle in degree (positive is clockwise). The rotated v-component is calculated according to the following formula: x = -u * sin (a) + v * cos (a) All three parameters may be ndarray or float.

    def met_vrot(u, v, a, undef=-9999.):

41. Calculation of wind velocity from u- and v-component of wind vector x = met_uv_wv(u, v) where u is the u-component of the wind vector, v is the v-component of the wind vector. The horizontal wind velocity is calculated according to the following formula: x = sqrt(u**2 + v**2) Both parameters may be ndarray or float.

    def met_uv_wv(u, v, undef=-9999.):

42. Calculation of wind direction from u- and v-component of wind vector x = met_uv_wd(u, v) where u is the u-component of the wind vector, v is the v-component of the wind vector. The horizontal wind velocity is calculated according to the following formulas: IF u = 0 AND v = 0 THEN x = 0 IF u = 0 AND v > 0 THEN x = 360 IF u = 0 AND v < 0 THEN x = 180 IF u < 0 THEN x = 270 - arctan(v/u) IF u > 0 THEN x = 90 - arctan(v/u) Both parameters may be ndarray or float.

def met_uv_wd(u, v, undef=-9999.):

43. Calculation of u-component of wind vector from wind velocity and wind direction x = met_wvwd_u(wv, wd) where wv is the horizontal wind velocity, wd is the horizontal wind direction. The u-component of the wind vector is calculated according to the following formula: x = -wv * sin (wd) Both parameters may be ndarray or float.

    def met_wvwd_u(wv, wd, undef=-9999.):

44. Calculation of v-component of wind vector from wind velocity and wind direction x = met_wvwd_v(wv, wd) where wv is the horizontal wind velocity, wd is the horizontal wind direction. The v-component of the wind vector is calculated according to the following formula: x = -wv * cos (wd) Both parameters may be ndarray or float.

    def met_wvwd_v(wv, wd, undef=-9999.):

45. If-statements x = ifeq(y,a0,a1,a2) means IF y == a0 THEN x = a1 ELSE x = a2 x = ifne(y,a0,a1,a2) means IF y != a0 THEN x = a1 ELSE x = a2 x = ifle(y,a0,a1,a2) means IF y <= a0 THEN x = a1 ELSE x = a2 x = ifge(y,a0,a1,a2) means IF y >= a0 THEN x = a1 ELSE x = a2 x = iflt(y,a0,a1,a2) means IF y > a0 THEN x = a1 ELSE x = a2 x = ifgt(y,a0,a1,a2) means IF y < a0 THEN x = a1 ELSE x = a2 All parameters may be ndarray or float.

    def ifeq(var1, iif, ithen, ielse): def ifne(var1, iif, ithen, ielse): def ifle(var1, iif, ithen, ielse): def ifge(var1, iif, ithen, ielse): def iflt(var1, iif, ithen, ielse): def ifgt(var1, iif, ithen, ielse):

46. Write variables to a file # not implemented

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jun-Dec 2014 by Matthias Cuntz (mc (at) macu (dot) de)
- Corrected type in met_tpot, Jun 2014, Corinna Rebmann
- Changed to Sphinx docstring and numpydoc, May 2020, Matthias Cuntz

varchs(a[, undef])	Change sign:
varadd(a, b[, undef])	Addition:
varsub(a, b[, undef])	Subtraction:
varmul(a, b[, undef])	Multiplication:
vardiv(a, b[, undef])	Division:
varsqr(a[, undef])	Square root:
varsqrt(a[, undef])	Square root:
varexp(a[, undef])	Exponentiation of e:
varlog(a[, undef])	Natural logarithm:
varpot(a, b[, undef])	Exponentiation:
lin(y, a0, a1[, undef])	Apply linear function:
quad(y, a0, a1, a2[, undef])	Apply 2nd order function:
cubic(y, a0, a1, a2, a3[, undef])	Apply 3rd order function:
hms(h, m, s[, undef])	Calculate fraction of day from hours, minutes and seconds:
bit_test(y, b[, start])	Bitwise test:
setlow(y, low[, islow, undef])	Replacement of underflows by new value:
sethigh(y, high[, ishigh, undef])	Replacement of overflows by new value:
limits(y, mini, maxi[, undef])	Replacement of underflows or overflows by undef:
mean(y[, axis, undef])	Calculation of mean value:
mini(y[, axis, undef])	Calculation of minimum value:
maxi(y[, axis, undef])	Calculation of maximum value:

Continued on next page

Table 20 – continued from previous page

met_lwrad(y, Tp[, undef])	Calculation of long wave radiation from net radiometer:
met_trad(Rl, epsilon[, undef])	Calculation of radiation temperature from long wave radiation:
met_alb(swd, swu[, swdmin, swumin, undef])	Calculation of albedo from short wave downward and upward radiation:
met_albl(swd, swu, swdmin, swumin[, undef])	Calculation of albedo from short wave downward and upward radiation with limits:
met_vpmax(temp[, undef])	Calculation of saturation water vapour pressure:
met_vpact(temp, rh[, undef])	Calculation of actual water vapour pressure:
met_vpdef(temp, rh[, undef])	Calculation of water vapour pressure deficit:
met_sh(temp, rh, p[, undef])	Calculation of specific humidity:
met_tpot(temp, p[, undef])	Calculation of potential temperature:
met_rho(temp, rh, p[, undef])	Calculation of air density:
met_dpt(temp, rh[, undef])	Calculation of dew point temperature:
met_h2oc(temp, rh, p[, undef])	Calculation of water vapour concentration:
met_h2oc_rh(temp, h, p[, undef])	Calculation of relative humidity from water vapour concentration:
met_wdrot(wd, a[, undef])	Rotation of wind direction:
met_urot(u, v, a[, undef])	Rotation of u-component of wind vector:
met_vrot(u, v, a[, undef])	Rotation of v-component of wind vector:
met_uv_wv(u, v[, undef])	Calculation of wind velocity from u- and v-component of wind vector:
met_uv_wd(u, v[, undef])	Calculation of wind direction from u- and v-component of wind vector:
met_wvwd_u(wv, wd[, undef])	Calculation of u-component of wind vector from wind velocity and wind direction:
met_wvwd_v(wv, wd[, undef])	Calculation of v-component of wind vector from wind velocity and wind direction:
ifeq(y, a0, a1, a2)	If-statements:
ifne(y, a0, a1, a2)	If-statements:
ifle(y, a0, a1, a2)	If-statements:
ifge(y, a0, a1, a2)	If-statements:
iflt(y, a0, a1, a2)	If-statements:
ifgt(y, a0, a1, a2)	If-statements:

varchs(*a*, *undef=-9999.0*)

Change sign: x = varchs(a) means x = -a, where a is ndarray or float.

Parameters

- **a** (*ndarray*) – input variable
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Changed sign
- *History*
- ——-
- *Written, Matthias Cuntz, Dec 2014*

varadd(*a*, *b*, *undef=-9999.0*)

Addition: x = varadd(a, b) means x = a + b, where a and b are ndarray or float.

Parameters

- **a** (*ndarray*) – input variable 1
- **b** (*ndarray*) – input variable 2
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Addition
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varsub(*a*, *b*, *undef=-9999.0*)

Subtraction: x = varsub(a, b) means x = a - b, where a and b are ndarray or float.

Parameters

- **a** (*ndarray*) – input variable 1
- **b** (*ndarray*) – input variable 2
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Subtraction
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varmul(*a*, *b*, *undef=-9999.0*)

Multiplication: x = varmul(a, b) means x = a * b, where a and b are ndarray or float.

Parameters

- **a** (*ndarray*) – input variable 1
- **b** (*ndarray*) – input variable 2
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Multiplication
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

vardiv(*a*, *b*, *undef=-9999.0*)

Division: x = vardiv(a, b) means x = a/b, where a and b are ndarray or float.

Parameters

- **a** (*ndarray*) – dividend

- **b** (*ndarray*) – divisor
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Division
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varsqr(*a*, *undef=-9999.0*)

Square root: x = varsqr(a) means x = sqrt(a), where a is ndarray or float.

Parameters

- **a** (*ndarray*) – input variable
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Square root
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varsqrt(*a*, *undef=-9999.0*)

Square root: x = varsqrt(a) means x = sqrt(a), where a is ndarray or float.

Parameters

- **a** (*ndarray*) – input variable
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Square root
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varexp(*a*, *undef=-9999.0*)

Exponentiation of e: x = varexp(a) means x = exp(a), where a is ndarray or float.

Parameters

- **a** (*ndarray*) – exponent
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Exponentiation

- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varlog(*a*, *undef=-9999.0*)

Natural logarithm: x = varlog(a) means x = ln(a), where a is ndarray or float.

Parameters

- **a** (*ndarray*) – input variable
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Natural logarithm
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

varpot(*a*, *b*, *undef=-9999.0*)

Exponentiation: x = varpot(a, b) means x = a**b, where a and b are ndarray or float.

Parameters

- **a** (*ndarray*) – base
- **b** (*ndarray*) – exponent
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – Exponentiation
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

lin(*y*, *a0*, *a1*, *undef=-9999.0*)

Apply linear function: x = lin(y, a0, a1) means x = a0 + a1 * y

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray or float*) – parameter 1
- **a1** (*ndarray or float*) – parameter 2
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – linear function
- *History*
- ——-

- *Written, Matthias Cuntz, Jun 2014*

quad(*y*, *a0*, *a1*, *a2*, *undef=-9999.0*)

Apply 2nd order function: x=quad(y,a0,a1,a2) means x = a0 + a1*y + a2*y**2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray or float*) – parameter 1
- **a1** (*ndarray or float*) – parameter 1
- **a2** (*ndarray or float*) – parameter 1
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – 2nd order function
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

cubic(*y*, *a0*, *a1*, *a2*, *a3*, *undef=-9999.0*)

Apply 3rd order function: x=cubic(y,a0,a1,a2,a3) means x = a0 + a1*y + a2*y**2 + a3*y**3

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray or float*) – parameter 1
- **a1** (*ndarray or float*) – parameter 2
- **a2** (*ndarray or float*) – parameter 3
- **a3** (*ndarray or float*) – parameter 4
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – 3rd order function
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

hms(*h*, *m*, *s*, *undef=-9999.0*)

Calculate fraction of day from hours, minutes and seconds: x = hms(h, m, s) means x = (h + m/60 + s/3600)/24

Parameters

- **h** (*ndarray*) – hour
- **m** (*ndarray*) – minute
- **s** (*ndarray*) – second

- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – fraction of day
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

bit_test (*y*, *b*, *start=0*)

Bitwise test: x = bit_test(y, b, start=0) means x = 1 if bit b is set in y otherwise x = 0.

Returns a list if b is an array.

Counting of b starts at start.

For the behaviour of the original logger tools, set start=1. Negative b's is not implemented.

Parameters

- **y** (*ndarray*) – input variable 1
- **b** (*int or ndarray*) – input variable 2
- **start** (*int, optional*) – Counting of *b* starts at start (default: 0)

Returns

- *int or list* – Bitwise test
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

setlow (*y*, *low*, *islow=None*, *undef=-9999.0*)

Replacement of underflows by new value: x = setlow(y,low,islow) means IF (y < low) THEN x = islow ELSE x = y

islow is optional. If not given low will be used.

This function may be used to adjust small negative values of short wave radiation during nighttime to zero values.

Parameters

- **y** (*ndarray*) – input variable
- **low** (*ndarray*) – lower threshold
- **islow** (*None or ndarray, optional*) – if not None, use islow in case of y < low
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – underflows replaced by new value
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

sethigh (*y*, *high*, *ishigh=None*, *undef=-9999.0*)

Replacement of overflows by new value: x = sethigh(y,high,ishigh) means IF (y > high) THEN x = ishigh ELSE x = y

ishigh is optional. If not given high will be used.

This function may be used to adjust relative humidity values of a little bit more than 100 % to 100 %.

Parameters

- **y** (*ndarray*) – input variable
- **high** (*ndarray*) – upper threshold
- **ishigh** (*None or ndarray, optional*) – if not None, use ishigh in case of y > high
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – overflows replaced by new value
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

limits(*y*, *mini*, *maxi*, *undef=-9999.0*)

Replacement of underflows or overflows by undef: x = limits(y, mini, maxi) means IF (y < mini) OR (y > maxi) THEN x = undef ELSE x = y

This function may be used to check values lying in between certain limits.

If one of the limits is exceeded the value is set to undef.

Parameters

- **y** (*ndarray*) – input variable
- **mini** (*ndarray*) – lower threshold
- **maxi** (*ndarray*) – upper threshold
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – underflows or overflows replaced by *undef*
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

mean(*y*, *axis=None*, *undef=-9999.0*)

Calculation of mean value: x = mean(y) means x = (y[0] + y[1] + . . . + y[n-1])/n

Parameters

- **y** (*ndarray*) – input variable
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

  If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – mean value
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

mini (*y*, *axis=None*, *undef=-9999.0*)

Calculation of minimum value: x = mini(y) means x = min(y[0],y[1],...,y[n-1])

Parameters

- **y** (*ndarray*) – input variable
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the minimum are computed. The default is to compute the minimum of the flattened array.

  If this is a tuple of ints, a minimum is performed over multiple axes, instead of a single axis or all the axes as before.
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – minimum value
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

maxi (*y*, *axis=None*, *undef=-9999.0*)

Calculation of maximum value: x = maxi(y) means x = max(y[0],y[1],...,y[n-1])

Parameters

- **y** (*ndarray*) – input variable
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the maximum are computed. The default is to compute the maximum of the flattened array.

  If this is a tuple of ints, a maximum is performed over multiple axes, instead of a single axis or all the axes as before.
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – maximum value
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_lwrad (*y*, *Tp*, *undef=-9999.0*)

Calculation of long wave radiation from net radiometer: x = met_lwrad(y, Tp)

The total radiation in W m-2 is calculated according to the following formula: x = y*fl + sigma*(Tp+T0)**4

where sigma = 5.67051 * 10**8 W m-2 K-4 is the Stephan-Boltzmann-Constant and fl is the factor for long wave radiation (reciprocal value of sensitivity) in W m-2 per mV.

The function assumes that fl was already applied before.

Parameters

- **y** (*ndarray*) – output voltage of the net radiometer [mV]
- **Tp** (*ndarray*) – pyranometer temperature, i.e. the temperature of the net radiometer body [degC]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – total radiation in W m-2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_trad(*Rl*, *epsilon*, *undef=-9999.0*)

Calculation of radiation temperature from long wave radiation: x = met_trad(Rl, epsilon)

The radiation temperature in degC is calculated according to the following formula: x= sqrt4(Rl/(sigma*epsilon)) - T0

where sigma = 5.67051 * 108 W m-2 K-4 is the Stephan-Boltzmann-Constant.

Parameters

- **Rl** (*ndarray*) – longwave radiation [W m-2]
- **epsilon** (*ndarray*) – long wave emissivity of the surface [0-1]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – radiation temperature in degC
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_alb(*swd*, *swu*, *swdmin=50.0*, *swumin=10.0*, *undef=-9999.0*)

Calculation of albedo from short wave downward and upward radiation: x = met_alb(swd, swu)

The albedo in % is calculated according to the following formula: x = 100 * (swu / swd)

If swd < swdmin (50 W m-2) or swu < swumin (10 W m-2) the result is undef.

Parameters

- **swd** (*ndarray*) – shortwave downward radiation [W m-2]
- **swu** (*ndarray*) – shortwave upward radiation [W m-2]
- **swdmin** (*float, optional*) – If *swd* < *swdmin* the result is undef (default: 50).
- **swumin** (*float, optional*) – If *swu* < *swumin* the result is undef (default: 10).

- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – albedo in %
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_albl (*swd*, *swu*, *swdmin*, *swumin*, *undef=-9999.0*)

Calculation of albedo from short wave downward and upward radiation with limits: x=met_albl(swd,swu,swdmin,swumin)

The albedo in % is calculated according to the following formula: x = 100 * (swu / swd)

If swd < swdmin or swu < swumin the result is *undef*.

Parameters

- **swd** (*ndarray*) – shortwave downward radiation [W m-2]
- **swu** (*ndarray*) – shortwave upward radiation [W m-2]
- **swdmin** (*float*) – If *swd* < *swdmin* the result is undef.
- **swumin** (*float*) – If *swu* < *swumin* the result is undef.
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)
- **undef** – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – albedo in %
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_vpmax (*temp*, *undef=-9999.0*)

Calculation of saturation water vapour pressure: x = met_vpmax(T) where

The saturation water vapour pressure in mbar (hPa) is calculated according to the following formula: x = 6.1078 * exp(17.08085 * T / (234.175 + T))

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – saturation water vapour pressure in mbar (hPa)
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_vpact (*temp*, *rh*, *undef=-9999.0*)

Calculation of actual water vapour pressure: x = met_vpact(T,rh)

The actual water vapour pressure in mbar (hPa) is calculated according to the following formulas:
Es = 6.1078*exp(17.08085*T/ (234.175 + T))

x = Es * rh/100

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – actual water vapour pressure in mbar (hPa)
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_vpdef (*temp*, *rh*, *undef=-9999.0*)

Calculation of water vapour pressure deficit: x = met_vpdef(T, rh)

The water vapour pressure deficit in mbar (hPa) is calculated according to the following formulas:
Es = 6.1078*exp(17.08085*T/ (234.175 + T))

E = Es * rh/100

x = Es - E

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – water vapour pressure deficit in mbar (hPa)
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_sh (*temp*, *rh*, *p*, *undef=-9999.0*)

Calculation of specific humidity: x = met_sh(T, rh, p)

The specific humidity in g kg-1 is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T))

E = Es * rh/100

x = 622 * E/(p-0.378*E)

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]

- **p** (*ndarray*) – air pressure [hPa = mbar]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – specific humidity in g kg-1
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_tpot(*temp*, *p*, *undef=-9999.0*)

Calculation of potential temperature: x = met_tpot(T, p)

The potential temperature in K is calculated according to the following formula: x = (T + T0) * (1000/p)**0.286

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **p** (*ndarray*) – air pressure [hPa = mbar]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – potential temperature in K
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_rho(*temp*, *rh*, *p*, *undef=-9999.0*)

Calculation of air density: x = met_rho(T, rh, p)

The air density in kg m-3 is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T))

E = Es * rh/100

sh = 622 * E/(p-0.378*E)

Tv = ((T + T0) * (1 + 0.000608 * sh)) - T0

x = p * 100 / (287.05 * (Tv + T0))

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]
- **p** (*ndarray*) – air pressure [hPa = mbar]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – air density in kg m-3
- *History*

- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_dpt (*temp*, *rh*, *undef=-9999.0*)

Calculation of dew point temperature: x = met_dpt(T, rh)

The dew point temperature in degC is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/(234.175 + T))

E = Es * rh/100

x = 234.175 * ln(E/6.1078)/(17.08085 - ln(E/6.1078))

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – dew point temperature in degC
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_h2oc (*temp*, *rh*, *p*, *undef=-9999.0*)

Calculation of water vapour concentration: x = met_h2oc(T, rh, p)

The water vapour concentration in mmol mol-1 is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/ (234.175 + T))

E = Es * rh/100

x = 0.1 * E /(0.001*p*100*0.001)

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **rh** (*ndarray*) – relative humidity [%]
- **p** (*ndarray*) – air pressure [hPa = mbar]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – water vapour concentration in mmol mol-1
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_h2oc_rh (*temp*, *h*, *p*, *undef=-9999.0*)

Calculation of relative humidity from water vapour concentration: x = met_h2oc_rh(T, [H2O], p)

The relative humidity in % is calculated according to the following formulas: Es = 6.1078*exp(17.08085*T/(234.175 + T))

E = 10 * [H2O] * 0.001 * p * 100 * 0.001

x = 100 * E / Es

Parameters

- **temp** (*ndarray*) – air temperature [degC]
- **h** (*ndarray*) – water vapour concentration [mmol mol-1]
- **p** (*ndarray*) – air pressure [hPa = mbar]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – relative humidity in %
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_wdrot(*wd*, *a*, *undef=-9999.0*)

Rotation of wind direction: x = met_wdrot(wd, a)

The rotated wind direction is calculated according to the following formulas: x = wd + a

IF x < 0 THEN x = x + 360

IF x <= 360 THEN x = x - 360

Parameters

- **wd** (*ndarray*) – wind direction [degree]
- **a** (*ndarray*) – rotation angle (positive is clockwise) [degree]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – rotated wind direction
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_urot(*u*, *v*, *a*, *undef=-9999.0*)

Rotation of u-component of wind vector: x = met_urot(u, v, a)

The rotated u-component is calculated according to the following formula: x = u * cos (a) + v * sin (a)

Parameters

- **u** (*ndarray*) – u-component of the wind vector
- **v** (*ndarray*) – v-component of the wind vector
- **a** (*ndarray*) – rotation angle (positive is clockwise) [degree]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – rotated u-component
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_vrot (*u*, *v*, *a*, *undef=-9999.0*)

Rotation of v-component of wind vector: x = met_vrot(u, v, a)

The rotated v-component is calculated according to the following formula: x = -u * sin (a) + v * cos (a)

Parameters

- **u** (*ndarray*) – u-component of the wind vector
- **v** (*ndarray*) – v-component of the wind vector
- **a** (*ndarray*) – rotation angle (positive is clockwise) [degree]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – rotated v-component
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_uv_wv (*u*, *v*, *undef=-9999.0*)

Calculation of wind velocity from u- and v-component of wind vector: x = met_uv_wv(u, v)

The horizontal wind velocity is calculated according to the following formula: x = sqrt(u**2 + v**2)

Parameters

- **u** (*ndarray*) – u-component of the wind vector
- **v** (*ndarray*) – v-component of the wind vector
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – horizontal wind velocity
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_uv_wd (*u*, *v*, *undef=-9999.0*)

Calculation of wind direction from u- and v-component of wind vector: x = met_uv_wd(u, v)

The horizontal wind velocity is calculated according to the following formulas: IF u = 0 AND v = 0 THEN x = 0

IF u = 0 AND v < 0 THEN x = 360

IF u = 0 AND v > 0 THEN x = 180

IF u > 0 THEN x = 270 - arctan(v/u)

IF u < 0 THEN x = 90 - arctan(v/u)

Parameters

- **u** (*ndarray*) – u-component of the wind vector
- **v** (*ndarray*) – v-component of the wind vector
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – horizontal wind velocity
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_wvwd_u(*wv*, *wd*, *undef=-9999.0*)

Calculation of u-component of wind vector from wind velocity and wind direction: x = met_wvwd_u(wv, wd)

The u-component of the wind vector is calculated according to the following formula: x = -wv * sin (wd)

Parameters

- **wv** (*ndarray*) – horizontal wind velocity [m s-1]
- **wd** (*ndarray*) – horizontal wind direction [degree]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – u-component of the wind vector
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

met_wvwd_v(*wv*, *wd*, *undef=-9999.0*)

Calculation of v-component of wind vector from wind velocity and wind direction: x = met_wvwd_v(wv, wd)

The v-component of the wind vector is calculated according to the following formula: x = -wv * cos (wd)

Parameters

- **wv** (*ndarray*) – horizontal wind velocity [m s-1]
- **wd** (*ndarray*) – horizontal wind direction [degree]
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – v-component of the wind vector
- *History*

- ——-
- *Written, Matthias Cuntz, Jun 2014*

ifeq(*y*, *a0*, *a1*, *a2*)

If-statements: x = ifeq(y,a0,a1,a2) means IF y == a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray*) – compare to input *y == a0*
- **a1** (*ndarray*) – result if *y == a0*
- **a2** (*ndarray*) – result if *y != a0*
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y == a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

ifne(*y*, *a0*, *a1*, *a2*)

If-statements: x = ifne(y,a0,a1,a2) means IF y != a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray*) – compare to input *y != a0*
- **a1** (*ndarray*) – result if *y != a0*
- **a2** (*ndarray*) – result if *y == a0*
- **y** –
- **a0** –
- **a1** –
- **a2** –
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y != a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

ifle(*y*, *a0*, *a1*, *a2*)

If-statements: x = ifle(y,a0,a1,a2) means IF y >= a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable

- **a0** (*ndarray*) – compare to input *y* <= *a0*
- **a1** (*ndarray*) – result if *y* <= *a0*
- **a2** (*ndarray*) – result if *y* > *a0*
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y >= a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

ifge(*y*, *a0*, *a1*, *a2*)

If-statements: x = ifge(y,a0,a1,a2) means IF y <= a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray*) – compare to input *y* >= *a0*
- **a1** (*ndarray*) – result if *y* >= *a0*
- **a2** (*ndarray*) – result if *y* < *a0*
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y <= a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

iflt(*y*, *a0*, *a1*, *a2*)

If-statements: x = iflt(y,a0,a1,a2) means IF y < a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray*) – compare to input *y* < *a0*
- **a1** (*ndarray*) – result if *y* < *a0*
- **a2** (*ndarray*) – result if *y* >= *a0*
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y < a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

ifgt(*y*, *a0*, *a1*, *a2*)

If-statements: x = ifgt(y,a0,a1,a2) means IF y > a0 THEN x = a1 ELSE x = a2

Parameters

- **y** (*ndarray*) – input variable
- **a0** (*ndarray*) – compare to input *y* > *a0*
- **a1** (*ndarray*) – result if *y* > *a0*
- **a2** (*ndarray*) – result if *y* <= *a0*
- **undef** (*float, optional*) – elements are excluded from the calculations if any of the inputs equals *undef* (default: -9999.)

Returns

- *ndarray* – IF y > a0 THEN x = a1 ELSE x = a2
- *History*
- ——-
- *Written, Matthias Cuntz, Jun 2014*

4.22 hesseflux.logtools

Purpose

logtools is a Python port of the Control File Functions of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

From the Logtools manual: "The functions range from simple mathematic operations to more complex and special procedures including functions for checking data. Most of the functions have the following appearance: *a = f(b,p1,p2,...,pn)* where *a* is the variable in which the result of the function *f* is stored, *b* is the input variable of the function and *p1* to *pn* are parameters (numbers) of the function. An output variable (result of a function) may be the same as an input variable. Some functions need more than one input variable, some functions do not need any parameter and some functions (*mean*, *mini*, *maxi*) may have a variable number of input variables."

The module is part of the JAMS Python package https://github.com/mcuntz/jams_python It will be synchronised with the JAMS package irregularily if used in other packages.

Subpackages

logtools	logtools is a Python port of the Control File Functions of Logtools, the Logger Tools Software of Olaf Kolle, MPI-BGC Jena, (c) 2012.

4.23 hesseflux.mad

mad : Median absolute deviation test, either on raw values or on 1st or 2nd derivatives.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Nov 2011 by Matthias Cuntz - mc (at) macu (dot) de
- ND-array, act on axis=0, May 2012, Matthias Cuntz
- Removed bug in broadcasting, Jun 2012, Matthias Cuntz
- Better usage of numpy possibilities, e.g. using np.diff, Jun 2012, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Use bottleneck for medians, otherwise loop over axis=1, Jul 2013, Matthias Cuntz and Juliane Mai
- Re-allow masked arrays and arrays with NaNs, Jul 2013, Matthias Cuntz
- Removed bug in NaN treatment, Oct 2013, Matthias Cuntz
- Keyword nonzero, Oct 2013, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

mad(datin[, z, deriv, nozero])	Median absolute deviation test, either on raw values, or on 1st or 2nd derivatives.

mad (*datin*, *z=7*, *deriv=0*, *nozero=False*)

Median absolute deviation test, either on raw values, or on 1st or 2nd derivatives.

Returns mask with False everywhere except where *<(median-MAD*z/0.6745)* or *>(md+MAD*z/0.6745)*.

Parameters

- **datin** (*array or masked array*) – *mad* acts on *axis=0*.
- **z** (*float, optional*) – Input is allowed to deviate maximum *z* standard deviations from the median (default: 7)
- **deriv** (*int, optional*) – 0: Act on raw input (default).

  1: Use first derivatives.

  2: Use 2nd derivatives.
- **nozero** (*bool, optional*) – True: exclude zeros (0.) from input *datin*.

Returns False everywhere except where input deviates more than *z* standard deviations from median

Return type array of bool

Notes

If input is an array then mad is checked along the zeroth axis for outlier.

1st derivative is calculated as *d = datin[1:n]-datin[0:n-1]* because mean of left and right would give 0 for spikes.

If *all(d.mask==True)* then return *d.mask*, which is all True.

Examples

```
>>> import numpy as np
>>> y = np.array([-0.25,0.68,0.94,1.15,2.26,2.35,2.37,2.40,2.47,2.54,2.62,
...               2.64,2.90,2.92,2.92,2.93,3.21,3.26,3.30,3.59,3.68,4.30,
...               4.64,5.34,5.42,8.01],dtype=np.float)
```

```
>>> # Normal MAD
>>> print(mad(y))
[False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False]
```

```
>>> print(mad(y,z=4))
[False False False False False False False False False False False False
 False False False False False False False False False False False False
 False  True]
```

```
>>> print(mad(y,z=3))
[ True False False False False False False False False False False False
 False False False False False False False False False False False False
  True  True]
```

```
>>> # MAD on 2nd derivatives
>>> print(mad(y,z=4,deriv=2))
[False False False False False False False False False False False False
 False False False False False False False False False False False  True]
```

```
>>> # direct usage
>>> my = np.ma.array(y, mask=mad(y,z=4))
>>> print(my)
[-0.25 0.68 0.94 1.15 2.26 2.35 2.37 2.4 2.47 2.54 2.62 2.64 2.9 2.92 2.92
 2.93 3.21 3.26 3.3 3.59 3.68 4.3 4.64 5.34 5.42 --]
```

```
>>> # MAD on several dimensions
>>> yy = np.transpose(np.array([y,y]))
>>> print(np.transpose(mad(yy,z=4)))
[[False False False False False False False False False False False False
  False False False False False False False False False False False False
  False  True]
 [False False False False False False False False False False False False
  False False False False False False False False False False False False
  False  True]]
```

```
>>> yyy = np.transpose(np.array([y,y,y]))
>>> print(np.transpose(mad(yyy,z=3)))
[[ True False False False False False False False False False False False
  False False False False False False False False False False False False
   True  True]
 [ True False False False False False False False False False False False
  False False False False False False False False False False False False
   True  True]
 [ True False False False False False False False False False False False
  False False False False False False False False False False False False
   True  True]]
```

```
>>> # Masked arrays
>>> my = np.ma.array(y, mask=np.zeros(y.shape))
>>> my.mask[-1] = True
```

(continues on next page)

(continued from previous page)

```
>>> print(mad(my,z=4))
[True False False False False False False False False False False False False
 False False False False False False False False False False False False
 False --]
```

```
>>> print(mad(my,z=3))
[True False False False False False False False False False False False False
 False False False False False False False False False False False True
 True --]
```

```
>>> # Arrays with NaNs
>>> ny = y.copy()
>>> ny[-1] = np.nan
>>> print(mad(ny,z=4))
[ True False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False]
```

```
>>> print(mad(ny,z=3))
[ True False False False False False False False False False False False False
 False False False False False False False False False False False  True
  True False]
```

```
>>> # Exclude zeros
>>> zy = y.copy()
>>> zy[1] = 0.
>>> print(mad(zy,z=3))
[ True  True False False False False False False False False False False False
 False False False False False False False False False False False False
  True  True]
```

```
>>> print(mad(zy,z=3,nozero=True))
[ True False False False False False False False False False False False False
 False False False False False False False False False False False False
  True  True]
```

4.24 hesseflux.madspikes

madspikes : Spike detection for using a moving median absolute difference filter.

This module was original written by Tino Rau and Matthias Cuntz, and maintained by Arndt Piayda while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written 2008 by Tino Rau and Matthias Cuntz - mc (at) macu (dot) de
- Maintained by Arndt Piayda since Aug 2014.
- Input can be pandas Dataframe or numpy array(s), Apr 2020, Matthias Cuntz
- Removed iteration, Apr 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

madspikes(dfin[, flag, isday, colhead, ...])	Spike detection for using a moving median absolute difference filter.

madspikes(*dfin*, *flag=None*, *isday=None*, *colhead=None*, *undef=-9999*, *nscan=720*, *nfill=48*, *z=7*, *deriv=2*, *swthr=10.0*, *plot=False*)

Spike detection for using a moving median absolute difference filter. Used with Eddy covariance data in Papale et al. (Biogeosciences, 2006).

Parameters

- **dfin** (*pandas.Dataframe or numpy.array*) – time series of data where spike detection with MAD should be applied.

  dfin can be a pandas.Dataframe.

  dfin can also me a numpy array. In this case *colhead* must be given. MAD will be applied along axis=0, i.e. on each column of axis=1.

- **flag** (*pandas.Dataframe or numpy.array, optional*) – flag Dataframe or array has the same shape as dfin. Non-zero values in *flag* will be treated as missing values in *dfin*.

  If *flag* is numpy array, *df.columns.values* will be used as column heads.

- **isday** (*array_like of bool, optional*) – True when it is day, False when night. Must have the same length as dfin.shape[0].

  If *isday* is not given, *dfin* must have a column with head 'SW_IN' or starting with 'SW_IN'. *isday* will then be *dfin['SW_IN'] > swthr*.

- **colhed** (*array_like of str, optional*) – column names if *dfin* is numpy array.

- **undef** (*float, optional*) – values having *undef* value are treated as missing values in *dfin* (default: -9999)

  np.nan is not allowed (working).

- **nscan** (*int, optional*) – size of moving window to calculate mad in time steps (default: 15*48)

- **nfill** (*int, optional*) – step size of moving window to calculate mad in time steps (default: 1*48)

mad will be calculated in *nscan* time window. Resulting mask will be applied only in *nfill* window in the middle of the *nscan* window. Then *nscan* window will be moved by *nfill* time steps.

- **z** (*float, optional*) – Input is allowed to deviate maximum *z* standard deviations from the median (default: 7)
- **deriv** (*int, optional*) – 0: Act on raw input.

  1: Use first derivatives.

  2: Use 2nd derivatives (default).
- **swthr** (*float, optional*) – Threshold to determine daytime from incoming shortwave radiation if *isday* not given (default: 10).
- **plot** (*bool, optional*) – True: data and spikes are plotted into madspikes.pdf (default: False).

Returns

- *pandas.Dataframe or numpy array* – flags, 0 everywhere except detected spikes set to 2.
- *History*
- ——-
- *Written, Matthias Cuntz & Tino Rau, 2008*
- *Maintained, Arndt Piayda, Aug 2014*
- *Modified, Matthias Cuntz, Apr 2020 - input can be pandas Dataframe or numpy array(s)* – - removed iteration Matthias Cuntz, May 2020 - numpy docstring format

4.25 hesseflux.nee2gpp

nee2gpp [Estimates photosynthesis (GPP) and ecosystem respiration (RECO)] from Eddy covariance CO2 flux data.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written 2012 by Matthias Cuntz - mc (at) macu (dot) de
- Set default undef to NaN, Mar 2012, Arndt Piayda
- Add wrapper nee2gpp for individual routines, Nov 2012, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Use generel cost function cost_abs from functions module, May 2013, Matthias Cuntz
- Use fmin_tnc to allow params < 0, Aug 2014, Arndt Piayda
- Keyword nogppnight, Aug 2014, Arndt Piayda
- Add wrapper nee2gpp for individual routines, Nov 2012, Matthias Cuntz
- Add wrapper nee2gpp for individual routines, Nov 2012, Matthias Cuntz
- Input can be pandas Dataframe or numpy array(s), Apr 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

nee2gpp(dfin[, flag, isday, date, ...])	Calculate photosynthesis (GPP) and ecosystem respiration (RECO) from Eddy covariance CO2 flux data.

nee2gpp(*dfin*, *flag=None*, *isday=None*, *date=None*, *timeformat='%Y-%m-%d %H:%M:%S'*, *colhead=None*, *undef=-9999*, *method='reichstein'*, *nogppnight=False*, *swthr=10.0*)

Calculate photosynthesis (GPP) and ecosystem respiration (RECO) from Eddy covariance CO2 flux data.

It uses either

1. a fit of Reco vs. temperature to all nighttime data (*method='falge'*), or
2. several fits over the season of Reco vs. temperature as in Reichstein et al. (2005) (*method='reichstein'*), or
3. the daytime method of Lasslop et al. (2010) (*method='lasslop'*),

in order to calculate Reco and then GPP = Reco - NEE.

Parameters

- **dfin** (*pandas.Dataframe or numpy.array*) – time series of CO2 fluxes and air temperature, and possibly incoming shortwave radiation and air vapour pressure deficit.

  dfin can be a pandas.Dataframe with the columns 'FC' or 'NEE' (or starting with 'FC_' or 'NEE_') for observed CO2 flux [umol(CO2) m-2 s-1] 'TA' (or starting with 'TA_') for air temperature [K]

  method='lasslop' or *method='day'* needs also 'SW_IN' (or starting with 'SW_IN') for incoming short-wave radiation [W m-2] 'VPD' (or starting with 'VPD') for air vapour deficit [Pa] The index is taken as date variable.

dfin can also me a numpy array with the same columns. In this case *colhead*, *date*, and possibly *dateformat* must be given.

- **flag** (*pandas.Dataframe or numpy.array, optional*) – flag Dataframe or array has the same shape as *dfin*. Non-zero values in *flag* will be treated as missing values in *dfin*.

  flag must follow the same rules as *dfin* if pandas.Dataframe.

  If *flag* is numpy array, *df.columns.values* will be used as column heads and the index of *dfin* will be copied to *flag*.

- **isday** (*array_like of bool, optional*) – True when it is day, False when night. Must have the same length as *dfin.shape[0]*.

  If *isday* is not given, *dfin* must have a column with head 'SW_IN' or starting with 'SW_IN'. *isday* will then be *dfin['SW_IN'] > swthr*.

- **date** (*array_like of string, optional*) – 1D-array_like of calendar dates in format given in *timeformat*.

  date must be given if *dfin* is numpy array.

- **timeformat** (*str, optional*) – Format of dates in *date*, if given (default: '%Y-%m-%d %H:%M:%S'). See strftime documentation of Python's datetime module: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

- **colhed** (*array_like of str, optional*) – column names if *dfin* is numpy array. See *dfin* for mandatory column names.

- **undef** (*float, optional*) – values having *undef* value are treated as missing values in *dfin* (default: -9999)

- **method** (*str, optional*) – method to use for partitioning. Possible values are:

  'global' or 'falge': fit of Reco vs. temperature to all nighttime data

  'local' of 'reichstein': several fits over the season of Reco vs. temperature as in Reichstein et al. (2005) (default)

  'day' or 'lasslop': method of Lasslop et al. (2010) fitting a light-response curve

- **nogppnight** (*float, optional*) – GPP will be set to zero at night. RECO will then equal NEE at night (default: False)

- **swthr** (*float, optional*) – Threshold to determine daytime from incoming shortwave radiation if *isday* not given (default: 10).

Returns pandas.Dataframe with two columns 'GPP' and 'RECO' with estimated photosynthesis and ecosystem respiration, or two numpy arrays [GPP, RECO].

Return type pandas.Dataframe or numpy arrays

Notes

Negative respiration possible at night if GPP is forced to 0 with *nogppnight=True*.

4.26 hesseflux.sread

sread : Read strings into array from a file.

This module was written by Matthias Cuntz while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written Jul 2009 by Matthias Cuntz (mc (at) macu (dot) de)
- Keyword transpose, Feb 2012, Matthias Cuntz
- Ported to Python 3, Feb 2013, Matthias Cuntz
- Removed bug when nc is list and contains 0, Nov 2014, Matthias Cuntz
- Keyword hskip, Nov 2014, Matthias Cuntz
- Do not use function lif, Feb 2015, Matthias Cuntz
- nc can be tuple, Feb 2015, Matthias Cuntz
- Large rewrite of code to improve speed, Feb 2015, Matthias Cuntz
- range instead of np.arange, Nov 2017, Matthias Cuntz
- Keywords cname, sname, hstrip, rename file to infile, Nov 2017, Matthias Cuntz
- Ignore unicode characters on read, Jun 2019, Matthias Cuntz
- Make ignoring unicode characters campatible with Python 2 and Python 3, Jul 2019, Matthias Cuntz
- Keywords encoding, errors with codecs module, Aug 2019, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

sread(infile[, nc, cname, skip, cskip, ...])	Read strings into string array from a file.

sread(*infile*, *nc=0*, *cname=None*, *skip=0*, *cskip=0*, *hskip=0*, *hstrip=True*, *separator=None*, *squeeze=False*, *reform=False*, *skip_blank=False*, *comment=None*, *fill=False*, *fill_value=''*, *strip=None*, *encoding='ascii'*, *errors='ignore'*, *header=False*, *full_header=False*, *transpose=False*, *strarr=False*)

Read strings into string array from a file.

Lines or columns can be skipped. Columns can be picked specifically.

Blank (only whitespace) and comment lines can be excluded.

The header of the file can be read separately.

This routines is exactly the same as fread but reads everything as strings instead of floats.

Parameters

- **infile** (*str*) – source file name
- **nc** (*int or iterable, optional*) – number of columns to be read [default: all (*nc<=0*)].

  nc can be an int or a vector of column indexes, starting with 0; *cskip* will be ignored in the latter case.
- **cname** (*iterable of str, optional*) – columns can be chosen by the values in the first header line; must be iterable with strings.

- **skip** (*int, optional*) – number of lines to skip at the beginning of file (default: 0)
- **cskip** (*int, optional*) – number of columns to skip at the beginning of each line (default: 0)
- **hskip** (*int, optional*) – number of lines in skip that do not belong to header (default: 0)
- **hstrip** (*bool, optional*) – True: strip header cells to match with cname (default: True)
- **separator** (*str, optional*) – column separator. If not given, columns separators are (in order): comma (‘,’), semicolon (‘;’), whitespace.
- **comment** (*iterable, optional*) – line gets excluded if first character of line is in comment sequence. Sequence must be iterable such as string, list and tuple.
- **fill_value** (*float, optional*) – value to fill in array in empty cells or if not enough columns in line and *fill==True* (default: ‘’).
- **strip** (*str, optional*) – Strip strings with str.strip(strip).

  None: strip quotes ” and ‘ (default).

  False: no strip (~30% faster).

  str: strip character given by *strip*.
- **encoding** (*str, optional*) – Specifies the encoding which is to be used for the file (default: ‘ascii’). Any encoding that encodes to and decodes from bytes is allowed.
- **errors** (*str, optional*) – Errors may be given to define the error handling during encoding of the file (default: ‘ignore’).

  Possible values: ‘strict’, ‘replace’, ‘ignore’.
- **squeeze** (*bool, optional*) – True: 2-dim array will be cleaned of degenerated dimension, i.e. results in a vector.

  False: array will be two-dimensional as read (default)
- **reform** (*bool, optional*) – Same as squeeze.
- **skip_blank** (*bool, optional*) – True: continues reading after blank line.

  False: stops reading at first blank line (default).
- **fill** (*bool, optional*) – True: fills in *fill_value* if not enough columns in line.

  False: stops execution and returns None if not enough columns in line (default).
- **header** (*bool, optional*) – True: header strings will be returned.

  False: numbers in file will be returned (default).
- **full_header** (*bool, optional*) – True: header is a string vector of the skipped rows.

  False: header will be split in columns, exactly as the data, and will hold only the selected columns (default).
- **transpose** (*bool, optional*) – True: column-major format *output(0:ncolumns,0:nlines)*.

  False: row-major format *output(0:nlines,0:ncolumns)* (default).
- **strarr** (*bool, optional*) – True: return as numpy array of strings.

  False: return as list (default).

Returns

Depending on options:

List of strings if *header==False* and *strarr==False*.

Array of strings if *header==False* and *strarr==True*.

List with file header strings if *header==True* and *strarr==False*.

String array of file header if *header==True* and *strarr==True*.

List of lines of strings if *header=True* and *full_header=True*.

Return type array of str

Notes

If *header=True* then skip is counterintuitive because it is actually the number of header rows to be read. This is to be able to have the exact same call of the function, once with *header=False* and once with *header=True*.

If *fill==True*, blank lines are not filled but are takes as end of file.

transpose=True has no effect on 1D output such as 1 header line.

Examples

```
>>> # Create some data
>>> filename = 'test.dat'
>>> ff = open(filename, 'w')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 2.2 2.3 2.4\n')
>>> ff.close()
```

```
>>> # Read sample file in different ways
>>> # header
>>> print(sread(filename, nc=2, skip=1, header=True))
['head1', 'head2']
>>> print(sread(filename, nc=2, skip=1, header=True, full_header=True))
['head1 head2 head3 head4']
>>> print(sread(filename, nc=1, skip=2, header=True))
[['head1'], ['1.1']]
>>> print(sread(filename, nc=1, skip=2, header=True, squeeze=True))
['head1', '1.1']
>>> print(sread(filename, nc=1, skip=2, header=True, squeeze=True,␣
↪strarr=True))
['head1' '1.1']
>>> print(sread(filename, nc=1, skip=2, header=True, squeeze=True,␣
↪transpose=True))
['head1', '1.1']
```

```
>>> # data
>>> print(sread(filename, skip=1))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename, skip=2))
[['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename, skip=1, cskip=1))
[['1.2', '1.3', '1.4'], ['2.2', '2.3', '2.4']]
>>> print(sread(filename, nc=2, skip=1, cskip=1))
[['1.2', '1.3'], ['2.2', '2.3']]
>>> print(sread(filename, nc=[1,3], skip=1))
```

(continues on next page)

(continued from previous page)

```
[['1.2', '1.4'], ['2.2', '2.4']]
>>> print(sread(filename, nc=1, skip=1))
[['1.1'], ['2.1']]
>>> print(sread(filename, nc=1, skip=1, reform=True))
['1.1', '2.1']
```

```
>>> # skip blank lines
>>> ff = open(filename, 'a')
>>> ff.writelines('\n')
>>> ff.writelines('3.1 3.2 3.3 3.4\n')
>>> ff.close()
>>> print(sread(filename, skip=1))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename, skip=1, skip_blank=True))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4'], ['3.1', '3.2', '3.
↪3', '3.4']]
>>> print(sread(filename, skip=1, strarr=True))
[['1.1' '1.2' '1.3' '1.4']
 ['2.1' '2.2' '2.3' '2.4']]
```

```
>>> print(sread(filename, skip=1, strarr=True, transpose=True))
[['1.1' '2.1']
 ['1.2' '2.2']
 ['1.3' '2.3']
 ['1.4' '2.4']]
>>> print(sread(filename, skip=1, transpose=True))
[['1.1', '2.1'], ['1.2', '2.2'], ['1.3', '2.3'], ['1.4', '2.4']]
```

```
>>> # skip comment lines
>>> ff = open(filename, 'a')
>>> ff.writelines('# First\n')
>>> ff.writelines('! Second second comment\n')
>>> ff.writelines('4.1 4.2 4.3 4.4\n')
>>> ff.close()
>>> print(sread(filename, skip=1))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename, skip=1, skip_blank=True, comment='#'))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4'], ['3.1', '3.2', '3.
↪3', '3.4'], ['!', 'Second', 'second', 'comment'], ['4.1', '4.2', '4.3', '4.4
↪']]
>>> print(sread(filename, skip=1, skip_blank=True, comment='#!'))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4'], ['3.1', '3.2', '3.
↪3', '3.4'], ['4.1', '4.2', '4.3', '4.4']]
>>> print(sread(filename, skip=1, skip_blank=True, comment=('#','!')))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4'], ['3.1', '3.2', '3.
↪3', '3.4'], ['4.1', '4.2', '4.3', '4.4']]
>>> print(sread(filename, skip=1, skip_blank=True, comment=['#','!']))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4'], ['3.1', '3.2', '3.
↪3', '3.4'], ['4.1', '4.2', '4.3', '4.4']]
```

```
>>> # Create some more data with escaped numbers
>>> filename2 = 'test2.dat'
>>> ff = open(filename2, 'w')
>>> ff.writelines('"head1" "head2" "head3" "head4"\n')
>>> ff.writelines('"1.1" "1.2" "1.3" "1.4"\n')
>>> ff.writelines('2.1 nan Inf "NaN"\n')
>>> ff.close()
>>> print(sread(filename2, skip=1, strarr=True, transpose=True, strip='"'))
[['1.1' '2.1']
 ['1.2' 'nan']
```

(continues on next page)

(continued from previous page)

```
 ['1.3' 'Inf']
 ['1.4' 'NaN']]
```

```
>>> # Create some more data with an extra (shorter) header line
>>> filename3 = 'test3.dat'
>>> ff = open(filename3, 'w')
>>> ff.writelines('Extra header\n')
>>> ff.writelines('head1 head2 head3 head4\n')
>>> ff.writelines('1.1 1.2 1.3 1.4\n')
>>> ff.writelines('2.1 2.2 2.3 2.4\n')
>>> ff.close()
```

```
>>> print(sread(filename3, skip=2))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename3, skip=2, hskip=1))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '2.2', '2.3', '2.4']]
>>> print(sread(filename3, nc=2, skip=2, hskip=1, header=True))
['head1', 'head2']
```

```
>>> # Create some more data with missing values
>>> filename4 = 'test4.dat'
>>> ff = open(filename4, 'w')
>>> ff.writelines('Extra header\n')
>>> ff.writelines('head1,head2,head3,head4\n')
>>> ff.writelines('1.1,1.2,1.3,1.4\n')
>>> ff.writelines('2.1,,2.3,2.4\n')
>>> ff.close()
```

```
>>> print(sread(filename4, skip=2))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '', '2.3', '2.4']]
>>> print(sread(filename4, skip=2, fill=True, fill_value='-1'))
[['1.1', '1.2', '1.3', '1.4'], ['2.1', '-1', '2.3', '2.4']]
```

```
>>> # cname
>>> print(sread(filename, cname='head2', skip=1, skip_blank=True, comment='#!',
↪ squeeze=True))
['1.2', '2.2', '3.2', '4.2']
>>> print(sread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!'))
[['1.1', '1.2'], ['2.1', '2.2'], ['3.1', '3.2'], ['4.1', '4.2']]
>>> print(sread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', header=True))
['head1', 'head2']
>>> print(sread(filename, cname=['head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', header=True, full_header=True))
['head1 head2 head3 head4']
>>> print(sread(filename, cname=['  head1','head2'], skip=1, skip_blank=True,␣
↪comment='#!', hstrip=False))
[['1.2'], ['2.2'], ['3.2'], ['4.2']]
```

```
>>> # Clean up doctest
>>> import os
>>> os.remove(filename)
>>> os.remove(filename2)
>>> os.remove(filename3)
>>> os.remove(filename4)
```

4.27 hesseflux.ustarfilter

ustarfilter [Filter Eddy Covariance data with friction velocity] following Papale et al. (Biogeosciences, 2006).

This module was original written by Tino Rau and Matthias Cuntz, and maintained by Arndt Piayda while at Department of Computational Hydrosystems, Helmholtz Centre for Environmental Research - UFZ, Leipzig, Germany, and continued by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

- Written 2008 by Tino Rau and Matthias Cuntz - mc (at) macu (dot) de
- Maintained by Arndt Piayda since Aug 2014.
- Input can be pandas Dataframe or numpy array(s), Apr 2020, Matthias Cuntz
- Using numpy docstring format, May 2020, Matthias Cuntz

The following functions are provided

`ustarfilter`(dfin[, flag, isday, date, ...])	Flag Eddy Covariance data using a threshold of friction velocity (u*) below which u* correlates with a reduction in CO2 flux.

ustarfilter(*dfin*, *flag=None*, *isday=None*, *date=None*, *timeformat='%Y-%m-%d %H:%M:%S'*, *colhead=None*, *ustarmin=0.01*, *nboot=100*, *undef=-9999*, *plot=False*, *nmon=3*, *nta-classes=7*, *corrcheck=0.5*, *nustarclasses=20*, *plateaucrit=0.95*, *swthr=10.0*)

Flag Eddy Covariance data using a threshold of friction velocity (u*) below which u* correlates with a reduction in CO2 flux. The algorithm follows the method presented in Papale et al. (Biogeosciences, 2006).

Parameters

- **dfin** (*pandas.Dataframe or numpy.array*) – time series of CO2 fluxes and friction velocity as well as air temperature.

  dfin can be a pandas.Dataframe with the columns 'FC' or 'NEE' (or starting with 'FC_' or 'NEE_') for observed CO2 flux [umol(CO2) m-2 s-1] 'USTAR' (or starting with 'USTAR') for friction velocity [m s-1] 'TA' (or starting with 'TA_') for air temperature [deg C] The index is taken as date variable.

  dfin can also me a numpy array with the same columns. In this case *colhead*, *date*, and possibly *dateformat* must be given.

- **flag** (*pandas.Dataframe or numpy.array, optional*) – flag Dataframe or array has the same shape as dfin. Non-zero values in *flag* will be treated as missing values in *dfin*.

  flag must follow the same rules as *dfin* if pandas.Dataframe.

  If *flag* is numpy array, *df.columns.values* will be used as column heads and the index of *dfin* will be copied to *flag*.

- **isday** (*array_like of bool, optional*) – True when it is day, False when night. Must have the same length as dfin.shape[0].

  If *isday* is not given, *dfin* must have a column with head 'SW_IN' or starting with 'SW_IN'. *isday* will then be *dfin['SW_IN'] > swthr*.

- **date** (*array_like of string, optional*) – 1D-array_like of calendar dates in format given in *timeformat*.

  date must be given if *dfin* is numpy array.

- **timeformat** (*str, optional*) – Format of dates in *date*, if given (default: '%Y-%m-%d %H:%M:%S'). See strftime documentation of Python's datetime module: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior
- **colhed** (*array_like of str, optional*) – column names if *dfin* is numpy array. See *dfin* for mandatory column names.
- **ustarmin** (*float, optional*) – minimum ustar threshold (default: 0.01)

  Papale et al. (Biogeosciences, 2006) take 0.1 for forest ant 0.01 otherwise.
- **nboot** (*int, optional*) – number of boot straps for estimate of confidence interval of u* threshold (default: 100)
- **undef** (*float, optional*) – values having *undef* value are treated as missing values in *dfin* (default: -9999)
- **plot** (*bool, optional*) – True: data and u* thresholds are plotted into ustarfilter.pdf (default: False)
- **nmon** (*int, optional*) – Number of month to combine for a season (default: 3).
- **ntaclasses** (*int, optional*) – Number of temperature classes per *nmon* months (default: 7).
- **corrcheck** (*float, optional*) – Skip temperature class if absolute of correlation coefficient between air temperature and ustar is greater equal *corrcheck* (default: 0.5).
- **nustarclasses** (*int, optional*) – Number of u* classes per temperature class (default: 20).
- **plateaucrit** (*float, optional*) – The u* threshold is the smallest u* class that has an average CO2 flux, which is higher than *plateaucrit* times the mean CO2 flux of all u* classes above this class (default: 0.95).
- **swthr** (*float, optional*) – Threshold to determine daytime from incoming shortwave radiation if *isday* not given (default: 10).

Returns

- *# report 5, 50 and 95 percentile*
- *oustars = np.quantile(bustars, (0.05, 0.5, 0.95), axis=0)*
- *tuple of numpy array, pandas.Dataframe or numpy array* – numpy array with 5, 50 and 95 percentile of u* thresholds, flags: 0 everywhere except set to 2 where u* < u*-threshold.

Notes

Works ONLY for a data set of at least one full year.

PYTHON MODULE INDEX

h

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

U

V