# DOLfYN Documentation

***Release 1.3.0***

**DOLfYN Developers**

**Apr 20, 2023**

# CONTENTS


**1 Table of Contents** **3**
- 1.1 About . . . 3
- 1.2 Installation . . . 4
- 1.3 The Basics . . . 5
- 1.4 Metadata . . . 10
- 1.5 Rotations & Coordinate Systems . . . 12
- 1.6 Motion Correction . . . 16
- 1.7 DOLfYN API . . . 26

**2 Examples** **75**
- 2.1 ADCP Example . . . 75
- 2.2 ADV Example . . . 82

**3 Indices** **89**

**Bibliography** **91**

**Python Module Index** **93**

**Index** **95**

DOLfYN is the Doppler Oceanography Library for pYthoN.

It is designed to read and work with Acoustic Doppler Velocimeter (ADV) and Acoustic Doppler Profiler (ADP/ADCP) data. DOLfYN includes libraries for reading binary Nortek(tm) and Teledyne-RDI(tm) data files.

Please document any issues and submit feature requests via the DOLfYN issues page.

CHAPTER

ONE

# TABLE OF CONTENTS

## 1.1 About

DOLfYN is a library of tools for reading, processing, and analyzing data from oceanographic velocity measurement instruments such as acoustic Doppler velocimeters (ADVs) and acoustic Doppler current profilers (ADCPs). It includes tools to

- Read in raw ADCP/ADV datafiles
- QC velocity data
- Rotate vector data through coordinate systems (i.e. beam to instrument to Earth to principal frames of reference)
- Motion correction for ADV velocity measurements (via onboard IMU data)
- Bin/ensemble averaging
- Turbulence statistics for ADV data

### 1.1.1 Instrument Support

- Nortek:
  - AWAC ADCP (current data only, waves in development)
  - Signature AD2CP (current and waves)
  - Vector ADV
- TRDI:
  - Workhorse ADCPs (Monitor and Sentinel)
  - WinRiver output files
  - VMDAS output files

### 1.1.2 History

DOLfYN was originally created to provide open-source software for motion correction and turbulence analysis of velocity data collected from ADVs mounted on compliant moorings. It has since been expanded to include reading and analyzing ADCP data.

### 1.1.3 License

DOLfYN is released Apache License 2.0 (see the LICENSE.txt file in the repository).

## 1.2 Installation

DOLfYN can be installed using pip:

```
$ pip install dolfyn
```

Or, if you would like download the source code locally so that you can modify it, you can clone the repository and then use pip to install it as an 'editable' package:

```
$ git clone https://github.com/lkilcher/dolfyn.git
$ cd dolfyn
$ pip install -e .
```

Once installed, to create documentation (you may have to pip install sphinx_rtd_theme):

```
$ cd dolfyn/docs
$ make html
```

If you would like to contribute, please follow the guidelines in the *contributing.md* file.

### 1.2.1 Data Files and Test Files

DOLfYN has several moderately large (a few MB each) binary data files included with the repo. These are example data files, and test-data files used to confirm that the repository is functioning correctly. In order to keep the size of the source repository minimal, these data files are actually stored using GitHub's git-lfs tools.

This means that if you want to be able to load these example data files, or run the tests, you will need to install git-lfs. If you cloned the repository prior to installing git-lfs, run the command `git lfs fetch` after installing git-lfs to pull the files.

### 1.2.2 MATLAB Users

For users who want to use DOLfYN's file reading capabilities with minimal Python scripting, the binary2mat.py script can be used. So long as DOLfYN has been *installed properly*, you can use this script from the command line in a directory which contains your data files:

```
$ python binary2mat.py vector_data_imu01.vec
```

And DOLfYN will save the converted .mat file to your working directory, where *raw data* is stored into a 2-layer MATLAB structure.

### 1.2.3 Testing

Currently all testing is housed in the `tests/` folder (including the data files). To run the tests, you'll need to install pytest, then open a command prompt and run:

```
$ python -m pytest
```

If any of the tests do not pass, first confirm that you have installed all of the dependencies correctly, including *git-lfs*, then check to see if others are having a similar issue before creating a new one.

### 1.2.4 Dependencies

DOLfYN was originally built upon the h5py package and has since been refactored to build off xarray to make use of the netCDF data format. Support is upheld for python 3.6 onward.

- NumPy >=1.17.0
- SciPy. >=1.5.0
- xarray >= 1.17
- netCDF4 >= 1.5.7

## 1.3 The Basics

DOLfYN data objects are built on xarray DataArrays combined into a single Dataset with *attributes*, or info about the data. Xarray can be thought of as a multidimensional extension of pandas, though it is not built on top of pandas. Datasets and DataArrays support all of the same basic functionality of dictionaries (e.g., indexing, iterating, etc.), with additional functionality that is designed to streamline the process of analyzing and working with data.

### 1.3.1 Reading Source Datafiles

To begin, we load the DOLfYN module and read a data file:

```
>> import dolfyn
>> dat = dolfyn.read(<path/to/my_data_file>)
```

DOLfYN's read function supports reading Nortek and TRDI binary data files straight from the ADCP or from the manufacturer's processing software (e.g. TRDI's WinADCP, VMDAS, or WinRiver).

In an interactive shell, typing the variable name followed by enter/return will display information about the dataset, e.g.:

```
>> dat = dolfyn.read_example('AWAC_test01.wpr')
>> dat
<xarray.Dataset>
Dimensions:              (range: 20, time: 9997, beam: 3, dir: 3, x*: 3, earth: 3, inst:␣
↪3)
Coordinates:
  * range                (range) float32 1.41 2.41 3.41 ... 18.41 19.41 20.41
  * time                 (time) datetime64[ns] 2012-06-12T12:00:00 ... 2012-0...
  * beam                 (beam) int32 1 2 3
  * dir                  (dir) <U1 'E' 'N' 'U'
```

(continues on next page)

(continued from previous page)

```
  * x*                    (x*) int32 1 2 3
  * earth                 (earth) <U1 'E' 'N' 'U'
  * inst                  (inst) <U1 'X' 'Y' 'Z'
Data variables:
  beam2inst_orientmat  (beam, x*) float64 1.577 -0.7891 ... 0.3677 0.3677
  error                (time) uint16 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  batt                 (time) float32 13.6 13.6 13.6 13.6 ... 13.4 13.4 13.4
  c_sound              (time) float32 1.489e+03 1.489e+03 ... 1.512e+03
  heading              (time) float32 119.3 119.4 119.7 ... 128.7 128.9 129.0
  pitch                (time) float32 6.55e+03 6.55e+03 ... 6.552e+03
        ...
  pressure             (time) float32 16.03 16.01 16.02 ... 0.042 0.032 0.038
  status               (time) float32 48.0 48.0 48.0 48.0 ... 48.0 48.0 48.0
  temp                 (time) float32 11.49 11.49 11.48 ... 18.72 18.72 18.72
  vel                  (dir, range, time) float64 -0.6648 -0.655 ... 0.645
  amp                  (beam, range, time) uint8 146 147 144 150 ... 25 25 25
  orientmat            (earth, inst, time) float64 0.3026 0.2995 ... 0.3123
Attributes:
  config:                    {'ProLogID': 156, 'ProLogFWver': '4.06', 'conf...
  inst_make:                 Nortek
  inst_model:                AWAC
  inst_type:                 ADCP
  SerialNum:                 WPR 1549
  Comments:                  AWAC on APL-UW Tidal Turbulence Mooring at Adm...
          ...
  has_imu:                   0
  cell_size:                 1.0
  blank_dist:                0.41
  latlon:                    [39.9402, -105.2283]
  declination:               8.28
  declination_in_orientmat:  1
```

This view reveals all the data stored within the xarray Dataset. There are four types of data displyed here: data variables, coordinates, dimensions and attributes.

- Data variables contain the main information stored as xarray DataArrays:

```
>> dat.vel
<xarray.DataArray 'vel' (dir: 3, range: 20, time: 9997)>
array([[[-0.66479734, -0.65496222, -0.69909159, ...,  1.90351055,
     1.94648366,  1.91131579],
  [-0.53663862, -0.56178903, -0.76993938, ...,  0.67961291,
     6.46099706, -0.3679769 ],
  [-0.63192198, -0.63142786, -0.52826604, ..., -0.0844491 ,
     2.69917045, -0.69253631],
  ...,
  [-0.90170625, -0.85587418, -0.48779671, ...,  3.71806074,
     0.63299628,  1.34105901],
  [-0.73322984, -0.66709612, -0.46033165, ..., -1.68639582,
     0.31451557,  2.93691549],
  [-0.90169828, -0.68338529, -0.57451738, ..., -2.77793829,
     2.43313374, -0.98629605]],
       ...
```

(continues on next page)

(continued from previous page)

```
   [[-0.11800002, -0.06400001,  0.13500002, ...,  1.483     ,
    1.49100016,  1.50800014],
  [-0.17100002, -0.18900001,  0.029     , ...,  0.86300005,
   -0.95699996,  0.14500003],
  [-0.08900001, -0.21400001,  0.016     , ..., -1.54799992,
    1.6550002 , -0.82600006],
  ...,
  [ 0.05099999, -0.12099999,  0.22100003, ..., -0.78300005,
    0.7650001 ,  0.164     ],
  [-0.05500002, -0.17199998,  0.152     , ..., -0.93699997,
    1.22200003, -0.87500013],
  [ 0.05099998, -0.146     ,  0.16600001, ..., -1.09600008,
    0.49300008,  0.64500009]]])
Coordinates:
  * range    (range) float32 1.41 2.41 3.41 4.41 ... 17.41 18.41 19.41 20.41
  * time     (time) datetime64[ns] 2012-06-12T12:00:00 ... 2012-06-12T14:46:36
  * dir      (dir) <U1 'E' 'N' 'U'
Attributes:
  units:    m/s
```

- Coordinates are arrays that contain the indices/labels/values of the data variables' dimensions, e.g. time, latitude, or longitude:

```
>> dat.time
<xarray.DataArray 'time' (time: 9997)>
array(['2012-06-12T12:00:00.000000000', '2012-06-12T12:00:01.000000000',
  '2012-06-12T12:00:02.000000000', ..., '2012-06-12T14:46:34.000000000',
  '2012-06-12T14:46:35.000000000', '2012-06-12T14:46:36.000000000'],
  dtype='datetime64[ns]')
Coordinates:
  * time     (time) datetime64[ns] 2012-06-12T12:00:00 ... 2012-06-12T14:46:36
```

- Dimensions are simply the names of the coordinate arrays
- Attributes can be thought of as comments, or information that provides insight into the data variables, and must be floats, strings or arrays. DOLfYN uses attributes to store information on coordinate rotations.

Data variables and coordinates can be accessed using dict-style syntax, *or* attribute-style syntax. For example:

```
>> dat['range']
<xarray.DataArray 'range' (range: 20)>
array([ 1.41,  2.41,  3.41,  4.41,  5.41,  6.41,  7.41,  8.41,  9.41, 10.41,
  11.41, 12.41, 13.41, 14.41, 15.41, 16.41, 17.41, 18.41, 19.41, 20.41],
  dtype=float32)
Coordinates:
  * range    (range) float32 1.41 2.41 3.41 4.41 ... 17.41 18.41 19.41 20.41
Attributes:
  units:    m

>> dat.amp[0]
<xarray.DataArray 'amp' (range: 20, time: 9997)>
array([[146, 147, 144, ...,  38,  38,  38],
  [136, 135, 136, ...,  25,  25,  25],
```

(continues on next page)

(continued from previous page)

```
  [130, 129, 132, ...,  25,  24,  25],
  ...,
  [ 89,  96,  88, ...,  23,  22,  23],
  [ 77,  82,  84, ...,  23,  23,  23],
  [ 61,  49,  58, ...,  23,  22,  23]], dtype=uint8)
Coordinates:
* range    (range) float32 1.41 2.41 3.41 4.41 ... 17.41 18.41 19.41 20.41
* time     (time) datetime64[ns] 2012-06-12T12:00:00 ... 2012-06-12T14:46:36
  beam     int32 1
Attributes:
  units:    counts
```

Dataset/DataArray attributes can be accessed as follows:

```
>> dat.blank_dist
0.41

>> dat.attrs['fs']
1.0
```

Note here that the display information includes the size of each array, it's coordinates and attributes. Active DataArray coordinates are signified with a '*'. The units of most variables are in the *MKS* system (e.g., velocity is in m/s), and angles are in degrees. Units are saved in relevant DataArrays as attributes; see the *Metadata* section for a complete list of the units of DOLfYN variables.

### 1.3.2 Subsetting Data

Xarray has its own built-in methods for selecting data.

A section of data can be extracted to a new Dataset or DataArray using `.isel`, `.sel` and/or with python's built-in `slice` function, for example:

```
# Returns a new DataArray containing data from 0 to 5 m.
>> datsub = dat.vel.sel(range=slice(0,5))

# Returns velocity in 'streamwise' direction
>> datsub = dat.vel.sel(orient='streamwise')

# Returns a new DataArray with the first 1000 indices (timesteps) from the original␣
↪DataArray
>> datsub = dat.vel.isel(time=slice(0,1000))
```

### 1.3.3 Data Analysis Tools

Analysis in DOLfYN is primarily set up to work through two API's (Advanced Programming Interfaces): the *ADCP Module* and the *ADV Module*, each of which contain functions that pertain to ADCP and ADV instruments, respectively. Functions and classes that pertain to both can be accessed from the main package import. See the *DOLfYN API* for further detail.

### 1.3.4 The DOLfYN view

In addition to working with xarray datasets directly, as described above DOLfYN also provides an alternate *DOLfYN view* into the data. This is accessed by:

```
>> dat_dolfyn = dat.velds
```

This view has several convenience methods, shortcuts, and functions built-in. It includes an alternate – and somewhat more informative/compact – description of the data object when in interactive mode:

```
>> dat_dolfyn
<ADCP data object>:  Nortek AWAC
  . 2.78 hours (started: Jun 12, 2012 12:00)
  . earth-frame
  . (9997 pings @ 1.0Hz)
  Variables:
  - time ('time',)
  - vel ('dir', 'range', 'time')
  - range ('range',)
  - orientmat ('earth', 'inst', 'time')
  - heading ('time',)
  - pitch ('time',)
  - roll ('time',)
  - temp ('time',)
  - pressure ('time',)
  - amp ('beam', 'range', 'time')
  ... and others (see `<obj>.variables`)
```

The variables in the dataset can be accessed using standard dictionary (key/item) syntax:

```
>> dat_dolfyn['time']
<xarray.DataArray 'time' (time: 9997)>
array(['2012-06-12T12:00:00.000000000', '2012-06-12T12:00:01.000000000',
       '2012-06-12T12:00:02.000000000', ..., '2012-06-12T14:46:34.000000000',
       '2012-06-12T14:46:35.000000000', '2012-06-12T14:46:36.000000000'],
      dtype='datetime64[ns]')
Coordinates:
  * time     (time) datetime64[ns] 2012-06-12T12:00:00 ... 2012-06-12T14:46:36
```

But trying to accessing variables using attribute syntax (`dat_dolfyn.time`) is not supported (returns `AttributeError`). However, we do include several shortcuts that utilize attribute syntax. The full list of *dolfyn-view* convenience methods and properties/shortcuts can be found in *`dolfyn.velocity.Velocity`*.

## 1.4 Metadata

DOLfYN generally uses the *MKS* system. Common variables and units are listed in Table 1:

Table 1.1: : The units of common variables found in DOLfYN data objects.

| Name | Units | Description/Notes |
|---|---|---|
| time | sec since 1970/01/01 00:00:00 | Time data (unaware of timezone) |
| vel | m/s | Velocity vector data |
| range | m | Distance from the instrument's transducer head(s) |
| depth | m | Instrument depth |
| c_sound | m/s | Speed of sound used in velocity calculations |
| press / pressure | dbar | Pressure measured by the instrument |
| temp | deg C | Temperature measured by the instrument |
| accel | $m/s^2$ | Vector acceleraton of the instrument |
| angrt | rad/s | Angular rotation rate of the instrument |
| mag | mG or nT | Magnetometer data |
| velraw | m/s | Velocity without motion-correction |
| velrot | m/s | Rotational motion velocity (computed from angrt) |
| velacc | m/s | Translational motion velocity (computed from accel) |
| acclow | $m/s^2$ | Low-pass filtered acceleration signal |
| heading | deg | Instrument heading* (clockwise from North) |
| pitch | deg | Instrument pitch* |
| roll | deg | Instrument roll* |
| orientmat | — | earth to instrument orientation matrix |
| amp | dB or counts | Measured sound level amplitude |
| corr | % or counts | Correlation between the sent and received pings. |
| ambig_vel | m/s | Ambiguity velocity |
| ensemble | — | Ensemble counter |
| error | — | Instrument error code |
| status | — | Instrument status code |
| tke_vec | $m^2/s^2$ | Variance of velocity components (0: u'u', 1: v'v', 2: w'w') |
| stress_vec | $m^2/s^2$ | Reynolds stress array (0: u'v', 1: u'w', 2: v'w') |
| U_mag | m/s | Horizontal velocity magnitude |
| U_std | m/s | Standard deviation of horizontal velocity magnitude |
| epsilon | $m^2/s^3$ | Turbulence dissipation rate |
| psd | <var units>$^2$/<freq units> | Spectra, calculated as power spectral densities |
| f | Hz | Spectral frequency |
| omega | rad/s | Spectral radial frequency |

### 1.4.1 DOLfYN Attributes

The `attrs` data-group of xarray Datasets is a place for user-specified meta-data and DOLfYN-specific implementation data. The most common variables found here are described in Table 2.

Table 1.2: The entries in `dat.attrs` that are used in DOLfYN.

| Name | Description/Notes |
|---|---|
| fs* | This is the sample rate of the instrument [Hz] |
| coord_sys* | The coordinate system of the data object. When a data object is rotated to a new coordinate system using the `dat.velds.rotate2()` method, the value of `dat.attrs['coord_sys']` is updated to reflect the final coordinate system. Valid values are: **beam**, **inst**, **earth**, and **principal**. For further details on these coordinate systems see the *Rotations & Coordinate Systems* section. |
| rotate_vars* | The variables in the data object that should be rotated when rotating the data object. |
| declination** | The magnetic declination where the measurements were made (in degrees that magnetic North is right of True North). Set this value using `dat.velds.set_declination(<value>)`. |
| declination_in_orier | A boolean specifying whether the `dat.orientmat` includes the declination. If this is True, then the earth coordinate system is True (i.e., v is velocity toward True North). |
| principal_heading | The heading of the +u direction for the 'principal' coordinate system [degrees clockwise from north]. |
| has_imu* | A boolean indicating whether the instrument has an IMU (inertial measurement unit) or AHRS (attitude heading reference system). |
| inst_make* | The manufacturer name. |
| inst_model* | The instrument model. |
| DutyCycle_NBurst* | The number of pings in a burst. |
| DutyCycle_NCycle* | The time – in number of pings – before the next burst starts. (this may be incorrect for some instrument types, please report issues) |
| inst2head_ve | The vector from the center of the ADV inst reference frame to the center of the head's reference frame, *in coordinates of the inst reference frame*. It must be specified in order to perform motion correction. For ADPs this is always zero because the two coordinate systems are centered at the same place. |
| inst2head_ro | The rotation matrix that rotates vectors from the instrument reference-frame to the head reference-frame. For ADCPs this is always the identity matrix. Valid values are: 3-by-3 valid rotation matrices, or: `'eye'`, `'identity'`, or `1` all of which specify that it is the identity matrix. This is typically used for cable-head ADVs where the ADV head is not oriented the same as the ADV pressure case. It must be specified in order to perform motion correction. |
| motion accel_filtfreq Hz | The filter-frequency for the computing translational motion from the acceleration signal of an IMU. This is only used for motion correction. This high-pass filter is applied prior to integrating acceleration. This value is only used if when `accel_filtfreq` is not explicitly specified when motion-correcting |
| motion vel_filtfreq Hz | The filter-frequency for the computing translational motion from the acceleration signal of an IMU. This is only used for motion correction. This high-pass filter is applied after integrating acceleration. This value is only used if when `vel_filtfreq` is not explicitly specified when motion-correcting. |
| latlon† | The location of the measurements in decimal degrees. Latitude is positive North of the equator, longitude is positive west of the prime-meridian. |

*: These entries are set and controlled by DOLfYN, and are not meant to be modified directly by the user.

**: These entries are set and controlled via `dat.set_<property name>` methods.

†: These entries are not used or set by DOLfYN, but they are useful measurement meta-data and are listed here to assist in standardizing the location and format of this information.

## 1.5 Rotations & Coordinate Systems

One of DOLfYN's primary advantages is that it contains tools for managing the coordinate system (a.k.a. the reference frame) of tensor data. The coordinate-system/rotation tools provided in DOLfYN have been tested to varying degrees on different types of instruments and configurations. See *the table* at the bottom of this page for details on the degree of testing of DOLfYN's rotations and coordinate-system tools that has occurred for several instrument types. With your help, we hope to improve our confidence in these tools for the wide-array of instruments and configurations that exist.

The values in the list `dat.attrs['rotate_vars']` specifies the vectors that are rotated when changing between different coordinate systems. The first dimension of these vectors are their coordinate directions, which are defined by the following coordinate systems:

- **Beam**: this is the coordinate system of the 'along-beam' velocities. When the data object is in 'beam' coordinates, the first dimension of the velocity vectors are: [beam1, beam2, … beamN]. This coordinate system is *not* orthonormal, which means that the inverse rotation (inst to beam) cannot be computed using the transpose of the beam-to-inst rotation matrix. Instead, the inverse of the matrix must be computed explicitly, which is done internally in DOLfYN (in `beam2inst()`).

  When a data object is in this coordinate system, only the velocity data (i.e., the variables in `dat.attrs['rotate_vars']` starting with `'vel'`) is in beam coordinates. Other vector variables listed in `'rotate_vars'` are in the 'inst' frame (e.g., `dat.angrt`). This is true for data read from binary files that is in beam coordinates, and also when rotating from other coordinate systems to beam coordinates.

- **Inst**: this is the 'instrument' coordinate system defined by the manufacturer. This coordinate system is orthonormal, but is not necessarily fixed. That is, if the instrument is rotating, then this coordinate system changes relative to the earth. When the data object is in 'inst' coordinates, the first dimension of the vectors are: [X, Y, Z, …].

- **Earth**: When the data object is in 'earth' coordinates, the first dimension of vectors are: [East, North, Up, …]. This coordinate system is also sometimes denoted as "ENU". If the declination is set the earth coordinate system is "True-East, True-North, Up" otherwise, East and North are magnetic. See the *Declination Handling* section for further details on setting declination.

  Note that the ENU definition used here is different from the 'North, East, Down' local coordinate system typically used by aircraft. Also note that the earth coordinate system is a 'rotationally-fixed' coordinate system: it does not rotate, but it is not necessarily *inertial* or *stationary* if the instrument slides around translationally (see the *Motion Correction* section for details on how to correct for translational motion).

- **Principal**: the principal coordinate system is a fixed coordinate system that has been rotated in the horizontal plane (around the Up axis) to align with the flow. In this coordinate system the first dimension of a vector is meant to be: [Stream-wise, Cross-stream, Up]. This coordinate system is defined by the variable `dat.attrs['principal_heading']`, which specifies the principal coordinate system's $+u$ direction. The $v$ direction is then defined by the right-hand-rule (with $w$ up). See the *Principal Heading* section for further details.

To rotate a data object into one of these coordinate systems, simply use the `rotate2` method:

```
>> dat = dolfyn.read_example('vector_data_imu01.VEC')
>> dolfyn.rotate2(dat, 'earth')
>> dat
<xarray.Dataset>
Dimensions:              (time: 27043, dir: 3, beam: 3, x*: 3, earth: 3, inst: 3, dirIMU:
↪ 3)
Coordinates:
  * time                 (time) datetime64[ns] 2012-06-12T12:00:02.681046 ......
  * dir                  (dir) <U1 'E' 'N' 'U'
  * beam                 (beam) int32 1 2 3
  * x*                   (x*) int32 1 2 3
```

(continues on next page)

(continued from previous page)

```
  * earth                (earth) <U1 'E' 'N' 'U'
  * inst                 (inst) <U1 'X' 'Y' 'Z'
  * dirIMU               (dirIMU) <U1 'E' 'N' 'U'
Data variables: (12/18)
  beam2inst_orientmat  (beam, x*) float64 2.74 -1.384 -1.354 ... 0.3489 0.3413
  batt                 (time) float32 11.3 11.3 11.3 11.3 ... 10.8 10.8 10.8
  c_sound              (time) float32 1.491e+03 1.491e+03 ... 1.486e+03
  heading              (time) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  pitch                (time) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  roll                 (time) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
                ...
  accel                (dir, time) float32 -0.03771 0.01074 ... 9.796 9.788
  angrt                (dir, time) float32 -0.006857 -0.004057 ... 0.1279
  mag                  (dir, time) float32 0.001869 0.001455 ... -0.5543
  orientmat            (earth, inst, time) float32 0.7867 0.7819 ... -0.9979
  orientation_down     (time) bool False False False ... False False False
  pressure             (time) float64 198.6 198.6 198.6 ... 623.2 623.2 623.2
Attributes:
  config:       {'ProLogID': 187, 'ProLogFWver': '4.12', 'config': 15412, '...
  inst_make:    Nortek
  inst_model:   Vector
  inst_type:    ADV
  rotate_vars:  ['vel', 'accel', 'angrt', 'mag']
  freq:         6000
  SerialNum:    VEC 9625
  Comments:     NREL Vector with INS on APL-UW Tidal Turbulence Mooring in ...
  fs:           32.0
  coord_sys:    earth
  has_imu:      1
```

### 1.5.1 Orientation Data

The instrument orientation data in DOLfYN data objects is contained in `orientmat` and `beam2inst_orientmat`. The `orientmat` data item is the earth2inst orientation matrix, $R$, of the instrument in the earth reference frame. It is a 3x3xNt array, where each 3x3 array is the rotation matrix that rotates vectors in the earth frame, $v_e$, into the instrument coordinate system, $v_i$, at each timestep:

$$v_i = R \cdot v_e$$

The ENU definitions of coordinate systems means that the rows of $R$ are the unit-vectors of the XYZ coordinate system in the ENU reference frame, and the columns are the unit vectors of the ENU coordinate system in the XYZ reference frame. That is, for this kind of simple rotation matrix between two orthogonal coordinate systems, the inverse rotation matrix is simply the transpose:

$$v_e = R^T \cdot v_i$$

### 1.5.2 Heading, Pitch, Roll

Most instruments do not calculate or output the orientation matrix by default. Instead, these instruments typically provide *heading*, *pitch*, and *roll* data (hereafter, *h,p,r*). Instruments that provide an `orientmat` directly will contain `dat.attrs['has_imu'] = 1`. Otherwise, the `orientmat` was calculated from *h,p,r*.

Note that an orientation matrix calculated from *h,p,r* can have larger error associated with it, partly because of the gimbal lock problem, and also because the accuracy of some *h,p,r* sensors decreases for large pitch or roll angles (e.g., >40 degrees).

Because the definitions of *h,p,r* are not consistent between instrument makes/models, and because DOLfYN-developers have chosen to utilize consistent definitions of orientation data (`orientmat`, and *h,p,r*), the following things are true:

- DOLfYN uses instrument-specific functions to calculate a consistent `orientmat` from the inconsistent definitions of *h,p,r*
- DOLfYN's consistent definitions *h,p,r* are generally different from the definitions provided by an instrument manufacturer (i.e., there is no consensus on these definitions, so DOLfYN developers have chosen one)

Varying degrees of validation have been performed to confirm that the `orientmat` is being calculated correctly for each instrument's definitions of *h,p,r*. See the *the table* at the bottom of this page for details on this. If your instrument has low confidence, or you suspect an error in rotating data into the earth coordinate system, and you have interest in doing the work to fix this, please reach out to us by filing an issue.

#### DOLfYN-Defined Heading, Pitch, Roll

The DOLfYN-defined *h,p,r* variables can be calculated using the `dolfyn.orient2euler()` function (`dolfyn.euler2orient()` provides the reverse functionality). This function computes these variables according to the following conventions:

- a "ZYX" rotation order. That is, these variables are computed assuming that rotation from the earth -> instrument frame happens by rotating around the z-axis first (heading), then rotating around the y-axis (pitch), then rotating around the x-axis (roll).
- heading is defined as the direction the x-axis points, positive clockwise from North (this is *opposite* the right-hand-rule around the Z-axis)
- pitch is positive when the x-axis pitches up (this is *opposite* the right-hand-rule around the Y-axis)
- roll is positive according to the right-hand-rule around the instrument's x-axis

#### Instrument heading, pitch, roll

The raw *h,p,r* data as defined by the instrument manufacturer is available in `dat.data_vars`. Note that this data does not obey the above definitions, and instead obeys the instrument manufacturer's definitions of these variables (i.e., it is exactly the data contained in the binary file). Also note that `dat['heading']` is unaffected by setting declination as described in the next section.

### 1.5.3 Declination Handling

DOLfYN includes functionality for handling declination, but the value of the declination must be specified by the user. There are two ways to set a data-object's declination:

1. Set declination explicitly using the `set_declination` method, for example:

```
dolfyn.set_declination(dat, 16.53)
```

2. Set declination in the `<data_filename>.userdata.json` file (more details ), then read the binary data file (i.e., using `dat = dolfyn.read(<data_filename>)`).

Both of these approaches produce modify the `dat` as described in the documentation for `set_declination()` .

### 1.5.4 Principal Heading

As described above, the principal coordinate system is meant to be the flow-aligned coordinate system (Streamwise, Cross-stream, Up). DOLfYN includes the `calc_principal_heading()` function to aide in identifying/calculating the principal heading. Using this function to identify the principal heading, an ADV data object that is in the earth-frame can be rotated into the principal coordinate system like this:

```
dat.attrs['principal_heading'] = dolfyn.calc_principal_heading(dat.vel)
dat.rotate2('principal')
```

Note here that if `dat` is in a coordinate system other than EARTH, you will get unexpected results, because you will calculate a *principal_heading* in the coordinate system that the data is in.

It should also be noted that by setting `dat.attrs['principal_heading']` the user can choose any horizontal coordinate system, and this might not be consistent with the *streamwise, cross-stream, up* definition described here. In those cases, the user should take care to clarify this point with collaborators to avoid confusion.

### 1.5.5 Degree of testing by instrument type

The table below details the degree of testing of the rotation, *p,r,h*, and coordinate-system tools contained in DOLfYN. The *confidence* column provides a general indication of the level of confidence that we have in these tools for each instrument.

If you encounter unexpected results that seem to be related to coordinate systems (especially for instruments and configurations that are listed as "low" or "medium" confidence), the best thing to do is file an issue.

Table 1.3: Table 1: Instruments tested to be consistent with DOLfYN's coordinate systems and rotation tools.

| Make | Series | Config | Confidence | Notes |
|---|---|---|---|---|
| Nortek | Vector | modern (~2019) firmware | Medium | "Some direct "instrument on the desk" confirmation of orientation-matrix and p,r,h calcs." |
| Nortek | Vector | with IMU, modern (2019) firmware | High | "Lots of direct 'instrument on the desk' confirmation of orientation-matrix and p,r,h calcs." |
| Nortek | AWAC | modern (~2019) firmware | Low | "This works, but there has been almost no testing to validate results" |
| Nortek | Signature | modern (~2019) firmware | High | "Lots of direct 'instrument on the desk' confirmation of orientation-matrix and p,r,h calcs." |
| Nortek | Signature | with IMU, modern (2019) firmware | Medium | "Some validation by reasonable results when working with data." |
| Teledyne RDI | Workhorse | modern (~2019-ish) firmware | Medium | "Some cross-validation with other sensors in post-processing, but minimal 'instrument on the desk' testing." |
| ALL | ALL | External input orientation data | NONE | "There has been no testing of external heading, pitch, or roll inputs" |

## 1.6 Motion Correction

The Nortek Vector ADV can be purchased with an Inertial Motion Unit (IMU) that measures the ADV motion. These measurements can be used to remove motion from ADV velocity measurements when the ADV is mounted on a moving platform (e.g. a mooring). This approach has been found to be effective for removing high-frequency motion from ADV measurements, but cannot remove low-frequency ($\lesssim$ 0.03Hz) motion because of bias-drift inherent in IMU accelerometer sensors that contaminates motion estimates at those frequencies.

This documentation is designed to document the methods for performing motion correction of ADV-IMU measurements. The accuracy and applicability of these measurements is beyond the scope of this documentation (see [Harding_etal_2017], [Kilcher_etal_2017]).

Nortek's Signature ADCP's are now also available with an Altitude and Heading Reference System (AHRS), but DOLfYN does not yet support motion correction of ADCP data.

### 1.6.1 Pre-Deployment Requirements

In order to perform motion correction the ADV-IMU must be assembled and configured correctly:

1. The ADV *head* must be rigidly connected to the ADV *pressure case*.
2. The ADV software must be configured properly. In the 'Deployment Planning' frame of the Vector Nortek Software, be sure that:
    a. The IMU sensor is enabled (checkbox) and set to record *'dAng dVel Orient'*.
    b. The 'Coordinate system' must be set to *'XYZ'*.
    c. It is recommended to set the ADV velocity range to $\pm$ *4 m/s*, or larger.

3. For cable-head ADVs be sure to record the position and orientation of the ADV head relative to the ADV pressure case 'inst' coordinate system (Figure 1). This information is specified in terms of the following variables:

   **inst2head_rotmat**
   The rotation matrix (a 3-by-3 array) that rotates vectors in the 'inst' coordinate system, to the ADV 'head' coordinate system. For fixed-head ADVs this is the identity matrix, but for cable-head ADVs it is an arbitrary unimodular (determinant of 1) matrix. This property must be in `dat.data_vars` in order to do motion correction.

   **inst2head_vec**
   The 3-element vector that specifies the position of the ADV head in the inst coordinate system (IMU coordinate system, Figure 1). This property must be in `dat.attrs` in order to do motion correction.

   **These variables are set in either the userdata.json file (prior to calling**
   `dolfyn.read`), or by setting them explicitly after the data file has been read:

```
dat.velds.set_inst2head_rotmat(<3x3 rotation matrix>)
dat.attrs['inst2head_vec'] = np.array([3-element vector])
```

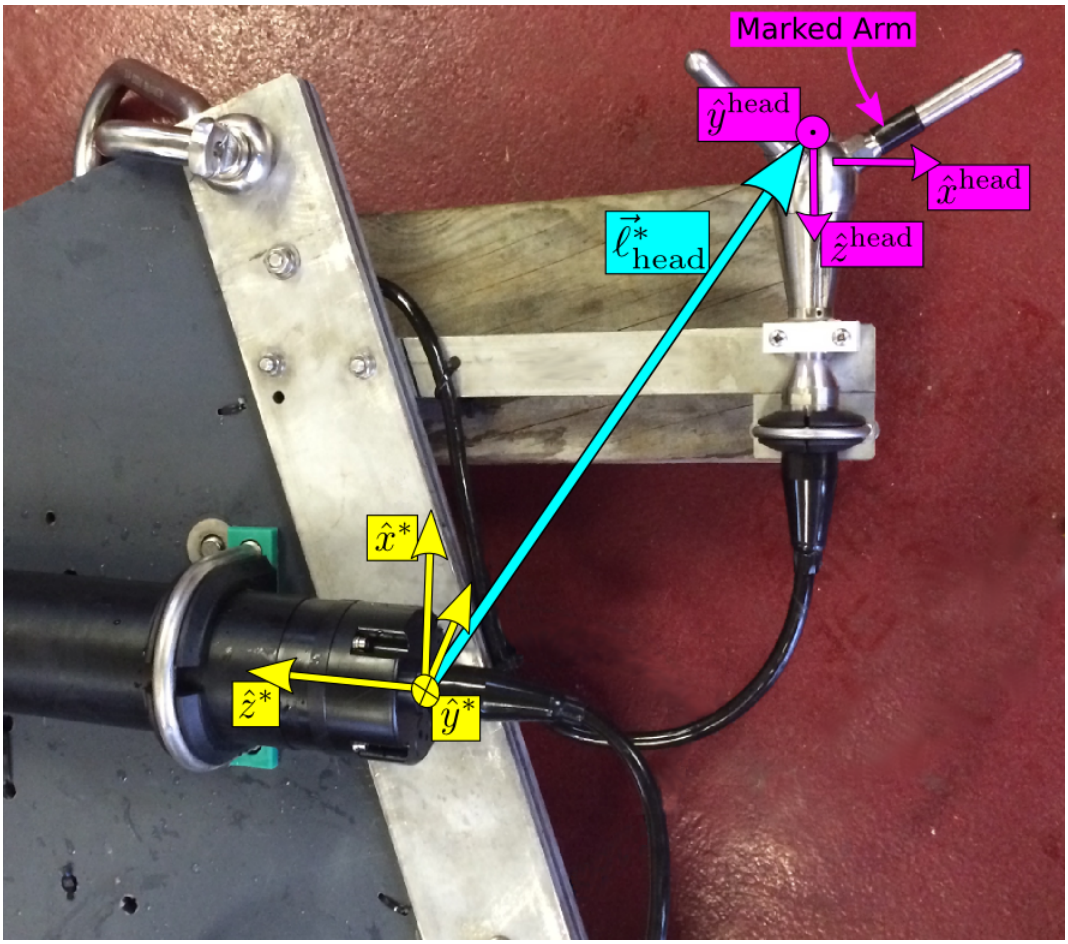



Figure 1.1: The ADV 'inst' (yellow) and head (magenta) coordinate systems. The $\hat{x}^{\mathrm{head}}$ -direction is known by the black-band around the transducer arm, and the $\hat{x}^{*}$ -direction is marked by a notch on the end-cap (indiscernible in the image). The cyan arrow indicates the `inst2head_vec` vector $\vec{\ell}^{*}_{head}$ . The perspective slightly distorts the fact that $\hat{x}^{\mathrm{head}} \parallel -\hat{z}^{*}$ , $\hat{y}^{\mathrm{head}} \parallel -\hat{y}^{*}$ , and $\hat{z}^{\mathrm{head}} \parallel -\hat{x}^{*}$ .

### 1.6.2 Specify metadata in a JSON file

The values in `dat.attrs` can also be set in a json file, `<data_filename>.userdata.json`, containing a single json-object. For example, the contents of these files should look something like:

```
{"inst2head_rotmat": "identity",
 "inst2head_vec": [-1.0, 0.5, 0.2],
 "motion accel_filtfreq Hz": 0.03,
 "declination": 8.28,
 "latlon": [39.9402, -105.2283]
}
```

Prior to reading a binary data file `my_data.VEC`, you can create a `my_data.userdata.json` file. Then when you do `dolfyn.read('my_data.VEC')`, DOLfYN will read the contents of `my_data.userdata.json` and include that information in the `dat.attrs` attribute of the returned data object. This feature is provided so that meta-data can live alongside your binary data files.

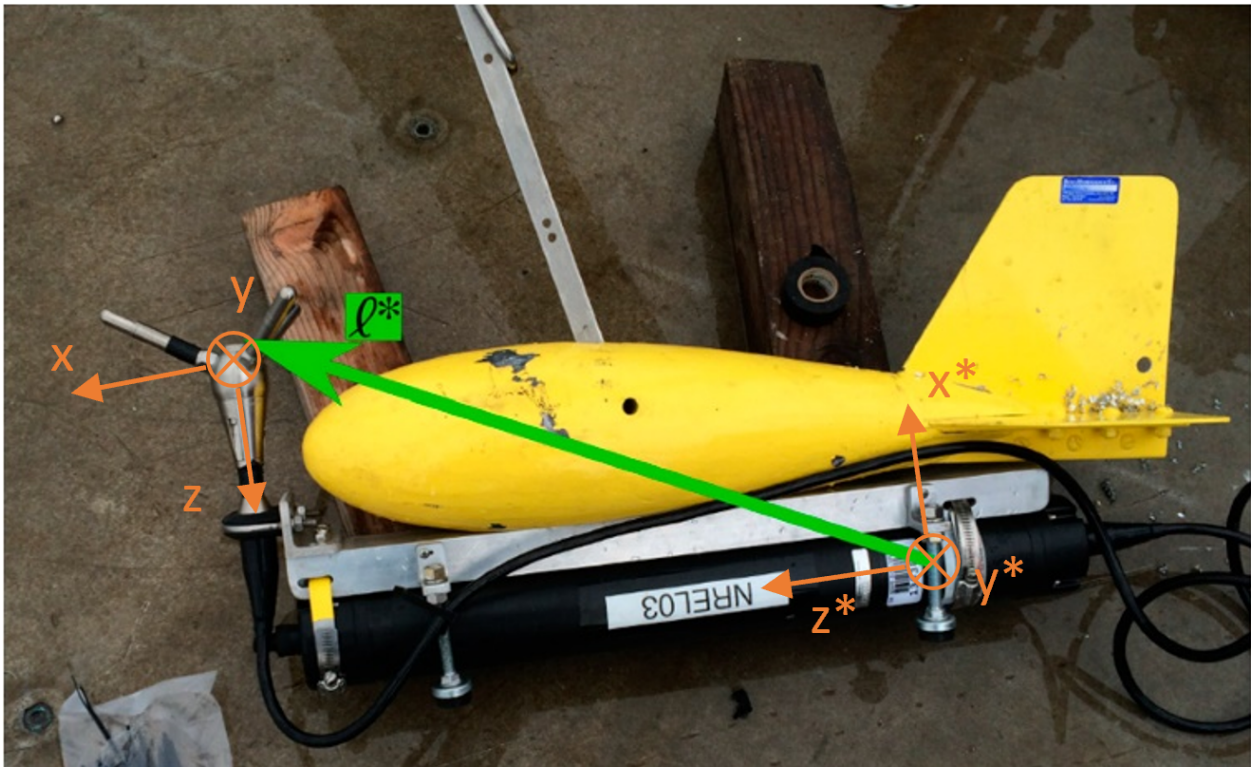



Figure 1.2: ADV mounted on a Columbus-type sounding weight.

The 'userdata.json' file corresponding to the ADV sounding weight in Figure 2 looks like:

```
{"inst2head_rotmat": [[ 0, 0,-1],
                      [ 0, 1, 0],
                      [ 1, 0, 0]],
 "inst2head_vec": [0.04, 0, 0.20],
 "motion accel_filtfreq Hz": 0.03,
}
```

### 1.6.3 Data Processing

After making ADV-IMU measurements, the DOLfYN package can perform motion correction processing steps on the ADV data. Assuming you have created a `vector_data_imu.userdata.json` file (to go with your `vector_data_imu.vec` data file), motion correction is fairly simple. You can either:

1. Utilize the DOLfYN API to perform motion-correction explicitly in Python:

```
import dolfyn.adv.api as avm
```

   a. Load your data file, for example:

```
dat = avm.read('vector_data_imu01.vec')
```

   b. Then perform motion correction:

```
avm.correct_motion(dat, accel_filtfreq=0.1) # specify the filter frequency in␣
↪Hz.
```

2. For users who want to perform motion correction with minimal Python scripting, the motcorrect_vector.py script can be used. So long as DOLfYN has been *installed properly*, you can use this script from the command line in a directory which contains your data files:

```
$ python motcorrect_vector.py vector_data_imu01.vec
```

   By default this will write a Matlab file containing your motion-corrected ADV data in ENU coordinates. Note that for fixed-head ADVs (no cable b/t head and battery case), the standard values for `inst2head_rotmat` and `inst2head_vec` can be specified by using the `--fixed-head` command-line parameter:

```
$ python motcorrect_vector.py --fixed-head vector_data_imu01.vec
```

   Otherwise, these parameters should be specified in the `.userdata.json` file, as described above.

   The motcorrect_vector.py script also allows the user to specify the `accel_filtfreq` using the `-f` flag. Therefore, to use a filter frequency of 0.1Hz (as opposed to the default 0.033Hz), you could do:

```
$ python motcorrect_vector.py -f 0.1 vector_data_imu01.vec
```

   It is also possible to do motion correction of multiple data files at once, for example:

```
$ python motcorrect_vector.py vector_data_imu01.vec vector_data_imu02.vec
```

   In all of these cases the script will perform motion correction on the specified file and save the data in ENU coordinates, in Matlab format. Happy motion-correcting!

After following one of these paths, your data will be motion corrected and its `.u`, `.v` and `.w` attributes are in an East, North and Up (ENU) coordinate system, respectively. In fact, all vector quantities in `dat` are now in this ENU coordinate system. See the documentation of the *`correct_motion()`* function for more information.

A key input parameter of motion-correction is the high-pass filter frequency that removes low-frequency bias drift from the IMU accelerometer signal (the default value is 0.03 Hz, a ~30 second period). For more details on choosing the appropriate value for a particular application, please see [Kilcher_etal_2016].

### 1.6.4 Motion Correction Examples

The two following examples depict the standard workflow for analyzing ADV-IMU data using DOLfYN.

#### ADV Motion Correction Ex.1

```
# To get started first import the DOLfYN ADV advanced programming
# interface (API):
import dolfyn.adv.api as api
from dolfyn import time

# Import matplotlib tools for plotting the data:
from matplotlib import pyplot as plt
import matplotlib.dates as dt
from datetime import datetime
import numpy as np

##############################
# User input and customization

# The file to load:
fname = '../../dolfyn/example_data/vector_data_imu01.vec'

# This is the vector from the ADV head to the instrument frame, in meters,
# in the ADV coordinate system.
inst2head_vec = np.array([0.48, -0.07, -0.27])

# This is the orientation matrix of the ADV head relative to the body
# (battery case).
# In this case the head was aligned with the body, so it is the
# identity matrix:
inst2head_rotmat = np.eye(3)

# The time range of interest.
# The instrument was in place on the seafloor starting at 12:08:30 on June 12, 2012.
t_range = [time.date2dt64(datetime(2012, 6, 12, 12, 8, 30)),
           # The data is good to the end of the file.
           time.date2dt64(datetime(2012, 6, 13))]

# This is the filter to use for motion correction:
accel_filter = 0.1

# End user input section.
##############################

# Read a file containing adv data:
dat_raw = api.read(fname, userdata=False)

# Crop the data for t_range
t_range_inds = (t_range[0] < dat_raw.time) & (dat_raw.time < t_range[1])
dat = dat_raw.isel(time=t_range_inds)
```

(continues on next page)

(continued from previous page)

```
# Set the inst2head rotation matrix and vector
api.set_inst2head_rotmat(dat, inst2head_rotmat)
dat.attrs['inst2head_vec'] = inst2head_vec

# Then clean the file using the Goring+Nikora method:
mask = api.clean.GN2002(dat.vel)
dat['vel'] = api.clean.clean_fill(dat.vel, mask, method='cubic')

####
# Create a figure for comparing screened data to the original.
fig = plt.figure(1, figsize=[8, 4])
fig.clf()
ax = fig.add_axes([.14, .14, .8, .74])

# Plot the raw (unscreened) data:
ax.plot(dat_raw.time, dat_raw.velds.u, 'r-', rasterized=True)

# Plot the screened data:
ax.plot(dat.time, dat.velds.u, 'g-', rasterized=True)
bads = np.abs(dat.velds.u - dat_raw.velds.u.isel(time=t_range_inds))
ax.text(0.55, 0.95,
        "%0.2f%% of the data were 'cleaned'\nby the Goring and Nikora method."
        % (float(sum(bads > 0)) / len(bads) * 100),
        transform=ax.transAxes,
        va='top',
        ha='left')

# Add some annotations:
text0 = dt.date2num(datetime(2012, 6, 12, 12, 8, 30))
ax.axvspan(dt.date2num(datetime(2012, 6, 12, 12)),
           text0, zorder=-10, facecolor='0.9',
           edgecolor='none')
ax.text(0.13, 0.9, 'Mooring falling\ntoward seafloor',
        ha='center', va='top', transform=ax.transAxes,
        size='small')
ax.text(text0 + 0.0001, 0.6, 'Mooring on seafloor',
        size='small',
        ha='left')
ax.annotate('', (text0 + 0.006, 0.3),
            (text0, 0.3),
            arrowprops=dict(facecolor='black', shrink=0.0),
            ha='right')

# Finalize the figure
# Format the time axis:
tkr = dt.MinuteLocator(interval=5)
frmt = dt.DateFormatter('%H:%M')
ax.xaxis.set_major_locator(tkr)
ax.xaxis.set_minor_locator(dt.MinuteLocator(interval=1))
ax.xaxis.set_major_formatter(frmt)
ax.set_ylim([-3, 3])
```

(continues on next page)

(continued from previous page)

```
# Label the axes:
ax.set_ylabel('$u\,\mathrm{[m/s]}$', size='large')
ax.set_xlabel('Time [June 12, 2012]')
ax.set_title('Data cropping and cleaning')
ax.set_xlim([dt.date2num(datetime(2012, 6, 12, 12)),
             dt.date2num(datetime(2012, 6, 12, 12, 30))])
####

dat_cln = dat.copy(deep=True)

# Perform motion correction (including rotation into earth frame):
dat = api.correct_motion(dat, accel_filter)

# Rotate the uncorrected data into the earth frame,
# for comparison to motion correction:
api.rotate2(dat_cln, 'earth')

# Then rotate it into a 'principal axes frame':
dat.attrs['principal_heading'] = api.calc_principal_heading(dat.vel)
dat_cln.attrs['principal_heading'] = api.calc_principal_heading(dat_cln.vel)
api.rotate2(dat, 'principal')
api.rotate2(dat_cln, 'principal')

# Average the data and compute turbulence statistics
dat_bin = api.calc_turbulence(dat, n_bin=9600, fs=dat.fs, n_fft=4096)
dat_cln_bin = api.calc_turbulence(
    dat_cln, n_bin=9600, fs=dat_cln.fs, n_fft=4096)

####
# Figure to look at spectra
fig2 = plt.figure(2, figsize=[6, 6])
fig2.clf()
ax = fig2.add_axes([.14, .14, .8, .74])

ax.loglog(dat_bin.freq, dat_bin['psd'].sel(S='Sxx').mean(axis=0),
          'b-', label='motion corrected')
ax.loglog(dat_cln_bin.freq, dat_cln_bin['psd'].sel(S='Sxx').mean(axis=0),
          'r-', label='no motion correction')

# Add some annotations
ax.axhline(1.7e-4, color='k', zorder=21)
ax.text(2e-3, 1.7e-4, 'Doppler noise level', va='bottom', ha='left',)

ax.text(1, 2e-2, 'Motion\nCorrection')
ax.annotate('', (3.6e-1, 3e-3), (1, 2e-2),
            arrowprops={'arrowstyle': 'fancy',
                        'connectionstyle': 'arc3,rad=0.2',
                        'facecolor': '0.8',
                        'edgecolor': '0.6'},
            ha='center')

ax.annotate('', (1.6e-1, 7e-3), (1, 2e-2),
```

(continues on next page)

(continued from previous page)

```
            arrowprops={'arrowstyle': 'fancy',
                        'connectionstyle': 'arc3,rad=0.2',
                        'facecolor': '0.8',
                        'edgecolor': '0.6'},
            ha='center')

# Finalize the figure
ax.set_xlim([1e-3, 20])
ax.set_ylim([1e-4, 1])
ax.set_xlabel('frequency [hz]')
ax.set_ylabel('$\mathrm{[m^2s^{-2}/hz]}$', size='large')

f_tmp = np.logspace(-3, 1)
ax.plot(f_tmp, 4e-5 * f_tmp ** (-5. / 3), 'k--')

ax.set_title('Velocity Spectra')
ax.legend()
ax.axvspan(1, 16, 0, .2, facecolor='0.8', zorder=-10, edgecolor='none')
ax.text(4, 4e-4, 'Doppler noise', va='bottom', ha='center',
        #bbox=dict(facecolor='w', alpha=0.9, edgecolor='none'),
        zorder=20)
```

### ADV Motion Correction Ex.2

```
import dolfyn
import dolfyn.adv.api as api

import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import matplotlib.dates as mpldt


#############################################################################
# User-input data
fname = '../../dolfyn/example_data/vector_data_imu01.VEC'
accel_filter = .03  # motion correction filter [Hz]
ensemble_size = 32*300  # sampling frequency * 300 seconds

# Read the data in, use the '.userdata.json' file
data_raw = dolfyn.read(fname, userdata=True)

# Crop the data for the time range of interest:
t_start = dolfyn.time.date2dt64(datetime(2012, 6, 12, 12, 8, 30))
t_end = data_raw.time[-1]
data = data_raw.sel(time=slice(t_start, t_end))


#############################################################################
# Clean the file using the Goring, Nikora 2002 method:
```

(continues on next page)

(continued from previous page)

```
bad = api.clean.GN2002(data.vel)
data['vel'] = api.clean.clean_fill(data.vel, bad, method='cubic')
# To not replace data:
# data.coords['mask'] = (('dir','time'), ~bad)
# data.vel.values = data.vel.where(data.mask)

# plotting raw vs qc'd data
ax = plt.figure(figsize=(20, 10)).add_axes([.14, .14, .8, .74])
ax.plot(data_raw.time, data_raw.velds.u, label='raw data')
ax.plot(data.time, data.velds.u, label='despiked')
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(mpldt.DateFormatter('%D %H:%M'))
ax.set_ylabel('u-dir velocity, (m/s)')
ax.set_title('Raw vs Despiked Data')
plt.legend(loc='upper right')
plt.show()

data_cleaned = data.copy(deep=True)


##############################################################################
# Perform motion correction
data = api.correct_motion(data, accel_filter, to_earth=False)
# For reference, dolfyn defines 'inst' as the IMU frame of reference, not
# the ADV sensor head
# After motion correction, the pre- and post-correction datasets coordinates
# may not align. Since here the ADV sensor head and battery body axes are
# aligned, data.u is the same axis as data_cleaned.u

# Plotting corrected vs uncorrect velocity in instrument coordinates
ax = plt.figure(figsize=(20, 10)).add_axes([.14, .14, .8, .74])
ax.plot(data_cleaned.time, data_cleaned.velds.u, 'g-', label='uncorrected')
ax.plot(data.time, data.velds.u, 'b-', label='motion-corrected')
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(mpldt.DateFormatter('%D %H:%M'))
ax.set_ylabel('u velocity, (m/s)')
ax.set_title('Pre- and Post- Motion Corrected Data in XYZ coordinates')
plt.legend(loc='upper right')
plt.show()


# Rotate the uncorrected data into the earth frame for comparison to motion
# correction:
dolfyn.rotate2(data, 'earth', inplace=True)
data_uncorrected = dolfyn.rotate2(data_cleaned, 'earth', inplace=False)

# Calc principal heading (from earth coordinates) and rotate into the
# principal axes
data.attrs['principal_heading'] = dolfyn.calc_principal_heading(data.vel)
data_uncorrected.attrs['principal_heading'] = dolfyn.calc_principal_heading(
    data_uncorrected.vel)
data.velds.rotate2('principal')
```

(continues on next page)

(continued from previous page)

```
data_uncorrected.velds.rotate2('principal')

# Plotting corrected vs uncorrected velocity in principal coordinates
ax = plt.figure(figsize=(20, 10)).add_axes([.14, .14, .8, .74])
ax.plot(data_uncorrected.time, data_uncorrected.velds.u,
        'g-', label='uncorrected')
ax.plot(data.time, data.velds.u, 'b-', label='motion-corrected')
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(mpldt.DateFormatter('%D %H:%M'))
ax.set_ylabel('streamwise velocity, (m/s)')
ax.set_title('Corrected and Uncorrected Data in Principal Coordinates')
plt.legend(loc='upper right')
plt.show()


##############################################################################
# Create velocity spectra
# Initiate tool to bin data based on the ensemble length. If n_fft is none,
# n_fft is equal to n_bin
ensemble_tool = api.ADVBinner(n_bin=9600, fs=data.fs, n_fft=4800)

# motion corrected data
mc_spec = ensemble_tool.calc_psd(data.vel, freq_units='Hz')
# not-motion corrected data
unm_spec = ensemble_tool.calc_psd(data_uncorrected.vel, freq_units='Hz')
# Find motion spectra from IMU velocity
uh_spec = ensemble_tool.calc_psd(data['velacc'] + data['velrot'],
                                 freq_units='Hz')

# Plot U, V, W spectra
U = ['u', 'v', 'w']
for i in range(len(U)):
    plt.figure(figsize=(15, 13))
    plt.loglog(uh_spec.freq, uh_spec[i].mean(axis=0), 'c',
               label=('motion spectra ' + str(accel_filter) + 'Hz filter'))
    plt.loglog(unm_spec.freq, unm_spec[i].mean(
        axis=0), 'r', label='uncorrected')
    plt.loglog(mc_spec.freq, mc_spec[i].mean(
        axis=0), 'b', label='motion corrected')

    # plot -5/3 slope
    f_tmp = np.logspace(-2, 1)
    plt.plot(f_tmp, 4e-5*f_tmp**(-5/3), 'k--', label='f^-5/3 slope')

    if U[i] == 'u':
        plt.title('Spectra in streamwise dir')
    elif U[i] == 'v':
        plt.title('Spectra in cross-stream dir')
    else:
        plt.title('Spectra in up dir')
    plt.xlabel('Freq [Hz]')
    plt.ylabel('$\mathrm{[m^2s^{-2}/Hz]}$', size='large')
```

(continues on next page)

(continued from previous page)

```
    plt.legend()
plt.show()
```

## 1.7 DOLfYN API

This is the Doppler Oceanography Library for pYthoN (DOLfYN). It is designed to read and work with oceanographyic velocity measurements from Acoustic Doppler Current Profilers (ADPs/ADCPs) and Acoustic Doppler Velocimeters (ADVs). It is a high-level object-oriented library composed of a set of data-object classes (types) that contain data from a particular measurement instrument, and a collection of functions that manipulate those data objects to accomplish data processing and data analysis tasks.

### 1.7.1 ADCP Module

This module contains routines for reading and working with ADP/ADCP data. It contains:

| | |
|---|---|
| *read* | Read a binary Nortek (e.g., .VEC, .wpr, .ad2cp, etc.) or RDI (.000, .PD0, .ENX, etc.) data file. |
| *load* | Load xarray dataset from netCDF (.nc) |
| *rotate2* | Rotate a dataset to a new coordinate system. |
| *calc_principal_heading* | Compute the principal angle of the horizontal velocity. |
| *clean* | Module containing functions to clean data |
| *VelBinner* | This is the base binning (averaging) tool. |
| *ADPBinner* | A class for calculating turbulence statistics from ADCP data |

#### Quick Example

```
# Start by importing DOLfYN:
import dolfyn
import dolfyn.adp.api as api

# Then read a file containing adv data:
ds = dolfyn.read_example('BenchFile01.ad2cp')

# This ADCP was sitting 0.5 m up from the seabed
# in a tripod
api.clean.set_range_offset(ds, h_deploy=0.5)

# Filter the data by low correlation values (< 50% here)
ds = api.clean.correlation_filter(ds, thresh=50)

# Rotate data from the instrument to true ENU (vs magnetic) frame:
# First set the magnetic declination
dolfyn.set_declination(ds, 10)  # 10 degrees East
dolfyn.rotate2(ds, 'earth')

# At any point you can save the data:
```

(continues on next page)

(continued from previous page)

```
#dolfyn.save(dat_cln, 'adcp_data.nc')

# And reload the data:
#dat_copy = dolfyn.load('adcp_data.nc')
```

### Cleaning Data

| | |
|---|---|
| *set_range_offset* | Adds an instrument's height above seafloor (for an up-facing instrument) or depth below water surface (for a down-facing instrument) to the range coordinate. |
| *find_surface* | Find the surface (water level or seafloor) from amplitude data and adds the variable "depth" to the input Dataset. |
| *find_surface_from_P* | Calculates the distance to the water surface. |
| *nan_beyond_surface* | Mask the values of 3D data (vel, amp, corr, echo) that are beyond the surface. |
| *correlation_filter* | Filters out data where correlation is below a threshold in the along-beam correlation data. |
| *medfilt_orient* | Median filters the orientation data (heading-pitch-roll or quaternions) |
| *val_exceeds_thresh* | Find values of a variable that exceed a threshold value, and assign "val" to the velocity data where the threshold is exceeded. |
| *fillgaps_time* | Fill gaps (nan values) in var across time using the specified method |
| *fillgaps_depth* | Fill gaps (nan values) in var along the depth profile using the specified method |

Module containing functions to clean data

dolfyn.adp.clean.**set_range_offset**(*ds*, *h_deploy*)

Adds an instrument's height above seafloor (for an up-facing instrument) or depth below water surface (for a down-facing instrument) to the range coordinate. Also adds an attribute to the Dataset with the current "h_deploy" distance.

**Parameters**

- **ds** (*xarray.Dataset*) – The adcp dataset to ajust 'range' on
- **h_deploy** (*numeric*) – Deployment location in the water column, in [m]

**Returns**

*None, operates "in place"*

### Notes

*Center of bin 1 = h_deploy + blank_dist + cell_size*

Nortek doesn't take *h_deploy* into account, so the range that DOLfYN calculates distance is from the ADCP transducers. TRDI asks for *h_deploy* input in their deployment software and is thereby known by DOLfYN.

If the ADCP is mounted on a tripod on the seafloor, *h_deploy* will be the height of the tripod +/- any extra distance to the transducer faces. If the instrument is vessel-mounted, *h_deploy* is the distance between the surface and downward-facing ADCP's transducers.

dolfyn.adp.clean.**find_surface**(*ds*, *thresh=10*, *nfilt=None*)

Find the surface (water level or seafloor) from amplitude data and adds the variable "depth" to the input Dataset.

**Parameters**

- **ds** (*xarray.Dataset*) – The full adcp dataset
- **thresh** (*int (default: 10)*) – Specifies the threshold used in detecting the surface. (The amount that amplitude must increase by near the surface for it to be considered a surface hit)
- **nfilt** (*int (default: None)*) – Specifies the width of the median filter applied, must be odd

**Returns**

*None, operates "in place"*

dolfyn.adp.clean.**find_surface_from_P**(*ds*, *salinity=35*)

Calculates the distance to the water surface. Temperature, salinity, and pressure are used to calculate seawater density, which is in turn used to calculate depth.

**Parameters**

- **ds** (*xarray.Dataset*) – The full adcp dataset
- **salinity** (*numeric (default: 35)*) – Water salinity in parts per thousand (ppt) or practical salinity units (psu)

**Returns**

- *None, operates "in place" and adds the variables "water_density" and*
- *"depth" to the input dataset.*

### Notes

Requires that the instrument's pressure sensor was calibrated/zeroed before deployment to remove atmospheric pressure.

Calculates seawater density using a linear approximation of the sea water equation of state:

$$\rho - \rho_0 = -\alpha(T - T_0) + \beta(S - S_0) + \kappa P$$

Where $\rho$ is water density, $T$ is water temperature, $P$ is water pressure, $S$ is practical salinity, $\alpha$ is the thermal expansion coefficient, $\beta$ is the haline contraction coefficient, and $\kappa$ is adiabatic compressibility.

dolfyn.adp.clean.**nan_beyond_surface**(*ds*, *val=nan*, *beam_angle=None*, *inplace=False*)

Mask the values of 3D data (vel, amp, corr, echo) that are beyond the surface.

**Parameters**

- **ds** (*xarray.Dataset*) – The adcp dataset to clean

- **val** (*nan or numeric (default: np.nan)*) – Specifies the value to set the bad values to
- **beam_angle** (*int (default: dataset.attrs['beam_angle'])*) – ADCP beam inclination angle
- **inplace** (*bool (default: False)*) – When True the existing data object is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset*) – Sets the adcp dataset where relevant arrays with values greater than *depth* set to NaN

### Notes

Surface interference expected to happen at *distance > range * cos(beam angle) - cell size*

dolfyn.adp.clean.**correlation_filter**(*ds*, *thresh=50*, *inplace=False*)

Filters out data where correlation is below a threshold in the along-beam correlation data.

**Parameters**

- **ds** (*xarray.Dataset*) – The adcp dataset to clean.
- **thresh** (*numeric (default: 50)*) – The maximum value of correlation to screen, in counts or %
- **inplace** (*bool (default: False)*) – When True the existing data object is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset*) – Elements in velocity, correlation, and amplitude are removed if below the correlation threshold

### Notes

Does not edit correlation or amplitude data.

dolfyn.adp.clean.**medfilt_orient**(*ds*, *nfilt=7*)

Median filters the orientation data (heading-pitch-roll or quaternions)

**Parameters**

- **ds** (*xarray.Dataset*) – The adcp dataset to clean
- **nfilt** (*numeric (default: 7)*) – The length of the median-filtering kernel *nfilt* must be odd.

**Returns**

**ds** (*xarray.Dataset*) – The adcp dataset with the filtered orientation data

**See also:**

`scipy.signal.medfilt`

dolfyn.adp.clean.**val_exceeds_thresh**(*var*, *thresh=5*, *val=nan*)

Find values of a variable that exceed a threshold value, and assign "val" to the velocity data where the threshold is exceeded.

**Parameters**

- **var** (*xarray.DataArray*) – Variable to clean
- **thresh** (*numeric (default: 5)*) – The maximum value of velocity to screen
- **val** (*nan or numeric (default: np.nan)*) – Specifies the value to set the bad values to

**Returns**
**ds** (*xarray.Dataset*) – The adcp dataset with datapoints beyond thresh are set to *val*

dolfyn.adp.clean.**fillgaps_time**(*var*, *method='cubic'*, *maxgap=None*)

Fill gaps (nan values) in var across time using the specified method

**Parameters**

- **var** (*xarray.DataArray*) – The variable to clean
- **method** (*string (default: 'cubic')*) – Interpolation method to use
- **maxgap** (*numeric (default: None)*) – Maximum gap of missing data to interpolate across

**Returns**
**out** (*xarray.DataArray*) – The input DataArray 'var' with gaps in 'var' interpolated across time

**See also:**

xarray.DataArray.interpolate_na

dolfyn.adp.clean.**fillgaps_depth**(*var*, *method='cubic'*, *maxgap=None*)

Fill gaps (nan values) in var along the depth profile using the specified method

**Parameters**

- **var** (*xarray.DataArray*) – The variable to clean
- **method** (*string (default: 'cubic')*) – Interpolation method to use
- **maxgap** (*numeric (default: None)*) – Maximum gap of missing data to interpolate across

**Returns**
**out** (*xarray.DataArray*) – The input DataArray 'var' with gaps in 'var' interpolated across depth

**See also:**

xarray.DataArray.interpolate_na

### 1.7.2 ADV Module

This module contains routines for reading and working with ADV data. It contains:

| | |
|---|---|
| *read* | Read a binary Nortek (e.g., .VEC, .wpr, .ad2cp, etc.) or RDI (.000, .PD0, .ENX, etc.) data file. |
| *load* | Load xarray dataset from netCDF (.nc) |
| *rotate2* | Rotate a dataset to a new coordinate system. |
| *set_inst2head_rotmat* | Set the instrument to head rotation matrix for the Nortek ADV if it hasn't already been set through a '.user-data.json' file. |
| *calc_principal_heading* | Compute the principal angle of the horizontal velocity. |
| *clean* | Module containing functions to clean data |
| *correct_motion* | This function performs motion correction on an IMU-ADV data object. |
| *VelBinner* | This is the base binning (averaging) tool. |
| *ADVBinner* | A class that builds upon *VelBinner* for calculating turbulence statistics and velocity spectra from ADV data |
| *calc_turbulence* | Functional version of *ADVBinner* that computes a suite of turbulence statistics for the input dataset, and returns a *binned* data object. |

### Quick Example

```
# Start by importing DOLfYN:
import dolfyn
import dolfyn.adv.api as api

# Then read a file containing adv data:
dat = dolfyn.read_example('vector_data01.VEC')

# Clean the file using the Goring+Nikora method:
mask = api.clean.GN2002(dat.vel)
dat['vel'] = api.clean.clean_fill(dat.vel, mask, npt=12, method='cubic')

# Rotate that data from the instrument to earth frame:
# First set the magnetic declination
dolfyn.set_declination(dat, 10)  # 10 degrees East
dolfyn.rotate2(dat, 'earth')

# Rotate it into a 'principal axes frame':
# First calculate the principal heading
dat.attrs['principal_heading'] = dolfyn.calc_principal_heading(dat.vel)
dolfyn.rotate2(dat, 'principal')

# Define an averaging object, and create an 'ensembled' data set:
binner = api.ADVBinner(n_bin=9600, fs=dat.fs, n_fft=4096)
dat_binned = binner(dat)

# At any point you can save the data:
#dolfyn.save(dat_binned, 'adv_data.nc')

# And reload the data:
#dat_bin_copy = dolfyn.load('adv_data.nc')
```

### Cleaning Data

| | |
|---|---|
| *clean_fill* | Interpolate over mask values in timeseries data using the specified method |
| *fill_nan_ensemble_mean* | Fill missing values with the ensemble mean. |
| *spike_thresh* | Returns a logical vector where a spike in *u* of magnitude greater than *thresh* occurs. |
| *range_limit* | Returns a logical vector that is True where the values of *u* are outside of *range*. |
| *GN2002* | The Goring & Nikora 2002 'despiking' method, with Wahl2003 correction. |

Module containing functions to clean data

dolfyn.adv.clean.**clean_fill**(*u*, *mask*, *npt=12*, *method='cubic'*, *maxgap=6*)

Interpolate over mask values in timeseries data using the specified method

**Parameters**

- **u** (*xarray.DataArray*) – The dataArray to clean.
- **mask** (*bool*) – Logical tensor of elements to "nan" out (from *spikeThresh*, *rangeLimit*, or *GN2002*) and replace
- **npt** (*int*) – The number of points on either side of the bad values that interpolation occurs over
- **method** (*string (default: 'cubic')*) – Interpolation scheme to use (linear, cubic, pchip, etc)
- **maxgap** (*int (default: 6)*) – Max number of consective nan's to interpolate across

**Returns**

**da** (*xarray.DataArray*) – The dataArray with nan's filled in

**See also:**

xarray.DataArray.interpolate_na

dolfyn.adv.clean.**fill_nan_ensemble_mean**(*u*, *mask*, *fs*, *window*)

Fill missing values with the ensemble mean.

**Parameters**

- **u** (*xarray.DataArray (..., time)*) – The dataArray to clean. Can be 1D or 2D.
- **mask** (*bool*) – Logical tensor of elements to "nan" out (from *spikeThresh*, *rangeLimit*, or *GN2002*) and replace
- **fs** (*int*) – Instrument sampling frequency
- **window** (*int*) – Size of window in seconds used to calculate ensemble means

**Returns**

**da** (*xarray.DataArray*) – The dataArray with nan's filled in

### Notes

Gaps larger than the ensemble size will not get filled in.

dolfyn.adv.clean.**spike_thresh**(*u*, *thresh=10*)

Returns a logical vector where a spike in *u* of magnitude greater than *thresh* occurs. Both 'Negative' and 'positive' spikes are found.

**Parameters**

- **u** (*xarray.DataArray*) – The timeseries data to clean.
- **thresh** (*int (default: 10)*) – Magnitude of velocity spike, must be positive.

**Returns**

**mask** (|*np.ndarray*|) – Logical vector with spikes labeled as 'True'

dolfyn.adv.clean.**range_limit**(*u*, *range=[-5, 5]*)

Returns a logical vector that is True where the values of *u* are outside of *range*.

**Parameters**

- **u** (*xarray.DataArray*) – The timeseries data to clean.
- **range** (*list (default: [-5, 5])*) – Min and max magnitudes beyond which are masked

**Returns**

**mask** (|*np.ndarray*|) – Logical vector with spikes labeled as 'True'

dolfyn.adv.clean.**GN2002**(*u*, *npt=5000*)

The Goring & Nikora 2002 'despiking' method, with Wahl2003 correction. Returns a logical vector that is true where spikes are identified.

**Parameters**

- **u** (*xarray.DataArray*) – The velocity array (1D or 3D) to clean.
- **npt** (*int (default: 5000)*) – The number of points over which to perform the method.

**Returns**

**mask** (|*np.ndarray*|) – Logical vector with spikes labeled as 'True'

**class** dolfyn.adv.motion.**CalcMotion**(*ds*, *accel_filtfreq=None*, *vel_filtfreq=None*, *to_earth=True*)

Bases: `object`

A 'calculator' for computing the velocity of points that are rigidly connected to an ADV-body with an IMU.

**Parameters**

- **ds** (*xarray.Dataset*) – The IMU-adv data that will be used to compute motion.
- **accel_filtfreq** (*float*) – the frequency at which to high-pass filter the acceleration sigal to remove low-frequency drift. (default: 0.03 Hz)
- **vel_filtfreq** (*float (optional)*) – a second frequency to high-pass filter the integrated acceleration. (default: 1/3 of accel_filtfreq)

**reshape**(*dat*, *n_bin*)

**calc_velacc**()

Calculates the translational velocity from the high-pass filtered acceleration signal.

**Returns**

**velacc** (*numpy.ndarray (3 x n_time)*) – The acceleration-induced velocity array (3, n_time).

`calc_velrot`(*vec*, *to_earth=None*)

Calculate the induced velocity due to rotations of the instrument about the IMU center.

**Parameters**

**vec** (*numpy.ndarray (len(3) or 3 x M)*) – The vector in meters (or vectors) from the body-origin (center of head end-cap) to the point of interest (in the body coord-sys).

**Returns**

**velrot** (*numpy.ndarray (3 x M x N_time)*) – The rotation-induced velocity array (3, n_time).

dolfyn.adv.motion.**correct_motion**(*ds*, *accel_filtfreq=None*, *vel_filtfreq=None*, *to_earth=True*, *separate_probes=False*)

This function performs motion correction on an IMU-ADV data object. The IMU and ADV data should be tightly synchronized and contained in a single data object.

**Parameters**

- **ds** (*xarray.Dataset*) – Cleaned ADV dataset in "inst" coordinates
- **accel_filtfreq** (*float*) – the frequency at which to high-pass filter the acceleration sigal to remove low-frequency drift.
- **vel_filtfreq** (*float (optional)*) – a second frequency to high-pass filter the integrated acceleration. (default: 1/3 of accel_filtfreq)
- **to_earth** (*bool (optional, default: True)*) – All variables in the ds.props['rotate_vars'] list will be rotated into either the earth frame (to_earth=True) or the instrument frame (to_earth=False).
- **separate_probes** (*bool (optional, default: False)*) – a flag to perform motion-correction at the probe tips, and perform motion correction in beam-coordinates, then transform back into XYZ/earth coordinates. This correction seems to be lower than the noise levels of the ADV, so the default is to not use it (False).

**Returns**

- *This function returns None, it operates on the input data object,*
- ds. The following attributes are added to *ds* – velraw is the uncorrected velocity

  **velrot is the rotational component of the head motion (from** angrt)

  **velacc is the translational component of the head motion (from** accel, the high-pass filtered accel sigal)

  acclow is the low-pass filtered accel sigal (i.e.,
- The primary velocity vector attribute, vel, is motion corrected
- *such that* – vel = velraw + velrot + velacc
- *The sigs are correct in this equation. The measured velocity*
- *induced by head-motion is *in the opposite direction* of the head**
- *motion. i.e. when the head moves one way in stationary flow, it*
- *measures a velocity in the opposite direction. Therefore, to*
- *remove the motion from the raw sigal we *add* the head motion.**

#### Notes

Acceleration signals from inertial sensors are notorious for having a small bias that can drift slowly in time. When integrating these sigals to estimate velocity the bias is amplified and leads to large errors in the estimated velocity. There are two methods for removing these errors,

1) high-pass filter the acceleration sigal prior and/or after integrating. This implicitly assumes that the low-frequency translational velocity is zero.

2) provide a slowly-varying reference position (often from a GPS) to an IMU that can use the sigal (usually using Kalman filters) to debias the acceleration sigal.

Because method (1) removes *real* low-frequency acceleration, method (2) is more accurate. However, providing reference position estimates to undersea instruments is practically challenging and expensive. Therefore, lacking the ability to use method (2), this function utilizes method (1).

For deployments in which the ADV is mounted on a mooring, or other semi-fixed structure, the assumption of zero low-frequency translational velocity is a reasonable one. However, for deployments on ships, gliders, or other moving objects it is not. The measured velocity, after motion-correction, will still hold some of this contamination and will be a sum of the ADV motion and the measured velocity on long time scales. If low-frequency motion is known separate from the ADV (e.g. from a bottom-tracking ADP, or from a ship's GPS), it may be possible to remove that sigal from the ADV sigal in post-processing.

### 1.7.3 Reading and Loading Data

Contains high level routines for reading in instrument binary data, and saving and loading xarray datasets. DOLfYN will automatically search through and select a binary reader based on the input data's file extension.

| | |
|---|---|
| *read* | Read a binary Nortek (e.g., .VEC, .wpr, .ad2cp, etc.) or RDI (.000, .PD0, .ENX, etc.) data file. |
| *read_example* | Read an ADCP or ADV datafile from the examples directory. |
| *save* | Save xarray dataset as netCDF (.nc). |
| *load* | Load xarray dataset from netCDF (.nc) |
| *save_mat* | Save xarray dataset as a MATLAB (.mat) file |
| *load_mat* | Load xarray dataset from MATLAB (.mat) file, complimentary to *save_mat()* |

I/O functions can be accessed directly from DOLfYN's main import:

```
>> import dolfyn
>> dat = dolfyn.read(<path/to/my_data_file>)
>> dolfyn.save(dat, <path/to/save_file.nc>)
```

dolfyn.io.api.**read**(*fname*, *userdata=True*, *nens=None*, ***kwargs*)

Read a binary Nortek (e.g., .VEC, .wpr, .ad2cp, etc.) or RDI (.000, .PD0, .ENX, etc.) data file.

**Parameters**

- **filename** (*string*) – Filename of instrument file to read.
- **userdata** (bool, or string of userdata.json filename (default True)) – Whether to read the '<base-filename>.userdata.json' file.
- **nens** (*None, int or 2-element tuple (start, stop)*) – Number of pings or ensembles to read from the file. Default is None, read entire file

- ****kwargs** (*dict*) – Passed to instrument-specific parser.

**Returns**
**ds** (*xarray.Dataset*) – An xarray dataset from instrument datafile.

dolfyn.io.api.**read_example**(*name*, ***kwargs*)

Read an ADCP or ADV datafile from the examples directory.

**Parameters**
**name** (*str*) – A few available files:

AWAC_test01.wpr BenchFile01.ad2cp RDI_test01.000 vector_burst_mode01.VEC vector_data01.VEC vector_data_imu01.VEC winriver01.PD0 winriver02.PD0

**Returns**
**ds** (*xarray.Dataset*) – An xarray dataset from the binary instrument data.

dolfyn.io.api.**save**(*ds*, *filename*, *format='NETCDF4'*, *engine='netcdf4'*, *compression=False*, ***kwargs*)

Save xarray dataset as netCDF (.nc).

**Parameters**

- **ds** (*xarray.Dataset*) – Dataset to save
- **filename** (*str*) – Filename and/or path with the '.nc' extension
- **compression** (*bool (default: False)*) – When true, compress all variables with zlib complevel=1.
- ****kwargs** (*dict*) – These are passed directly to `xarray.Dataset.to_netcdf()`

### Notes

Drops 'config' lines.

Rewrites variable encoding dict

More detailed compression options can be specified by specifying 'encoding' in kwargs. The values in encoding will take precedence over whatever is set according to the compression option above. See the xarray.to_netcdf documentation for more details.

dolfyn.io.api.**load**(*filename*)

Load xarray dataset from netCDF (.nc)

**Parameters**
**filename** (*str*) – Filename and/or path with the '.nc' extension

**Returns**
**ds** (*xarray.Dataset*) – An xarray dataset from the binary instrument data.

dolfyn.io.api.**save_mat**(*ds*, *filename*, *datenum=True*)

Save xarray dataset as a MATLAB (.mat) file

**Parameters**

- **ds** (*xarray.Dataset*) – Dataset to save
- **filename** (*str*) – Filename and/or path with the '.mat' extension
- **datenum** (*bool*) – If true, converts time to datenum. If false, time will be saved in "epoch time".

#### Notes

The xarray data format is saved as a MATLAB structure with the fields 'vars, coords, config, units'. Converts time to datenum

**See also:**

`scipy.io.savemat`

`dolfyn.io.api.load_mat`(*filename*, *datenum=True*)

Load xarray dataset from MATLAB (.mat) file, complimentary to *save_mat()*

A .mat file must contain the fields: {vars, coords, config, units}, where 'coords' contain the dimensions of all variables in 'vars'.

**Parameters**

- **filename** (*str*) – Filename and/or path with the '.mat' extension
- **datenum** (*bool*) – If true, converts time from datenum. If false, converts time from "epoch time".

**Returns**

**ds** (*xarray.Dataset*) – An xarray dataset from the binary instrument data.

**See also:**

`scipy.io.loadmat`

### 1.7.4 Rotate Functions

Contains functions for rotating data through frames of reference (FoR):

1. **'beam'**: Follows the acoustic beam FoR, where velocity data is organized by beam number 1-3 or 1-4.
2. **'inst'**: The instrument's *XYZ* Cartesian directions. For ADVs, this orientation is from the mark on the ADV body/battery canister, not the sensor head. For TRDI 4-beam instruments, the fourth velocity term is the error velocity (aka *XYZE*). For Nortek 4-beam instruments, this is *XYZ1 Z2*, where *E=Z2-Z1*.
3. **'earth'**: *East North UP* (*ENU*) FoR. Based on either magnetic or true North, depending on whether or not DOLfYN has a magnetic declination associated with the dataset. Instruments do not internally record magnetic declination, unless it has been supplied via external software like TRDI's VMDAS.
4. **'principal'**: Rotates velocity data into a *streamwise*, *cross-stream*, and *vertical* FoR based on the principal flow direction. One must calculate principal heading first.

| | |
|---|---|
| *rotate2* | Rotate a dataset to a new coordinate system. |
| *set_declination* | Set the magnetic declination |
| *calc_principal_heading* | Compute the principal angle of the horizontal velocity. |
| *set_inst2head_rotmat* | Set the instrument to head rotation matrix for the Nortek ADV if it hasn't already been set through a '.user-data.json' file. |
| *euler2orient* | Calculate the orientation matrix from DOLfYN-defined euler angles. |
| *orient2euler* | Calculate DOLfYN-defined euler angles from the orientation matrix. |
| *quaternion2orient* | Calculate orientation from Nortek AHRS quaternions, where q = [W, X, Y, Z] instead of the standard q = [X, Y, Z, W] = [q1, q2, q3, q4] |

These functions pertain to both ADCPs and ADVs:

```
>> import dolfyn
>> dat = dolfyn.read_example('burst_mode01.VEC')

>> dolfyn.set_declination(dat, 12)
>> dolfyn.rotate2(dat, 'earth')

>> dat.attrs['principal_heading'] = dolfyn.calc_principal_heading(dat['vel'])
>> dolfyn.rotate2(dat, 'principal')
```

dolfyn.rotate.api.**rotate2**(*ds*, *out_frame='earth'*, *inplace=True*)

Rotate a dataset to a new coordinate system.

**Parameters**

- **ds** (*xr.Dataset*) – The dolfyn dataset (ADV or ADCP) to rotate.
- **out_frame** (*string {'beam', 'inst', 'earth', 'principal'}*) – The coordinate system to rotate the data into.
- **inplace** (*bool (default: True)*) – When True `ds` is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns a new rotated dataset **when ``inplace=False``**, otherwise returns None.

### Notes

- This function rotates all variables in `ds.attrs['rotate_vars']`.
- In order to rotate to the 'principal' frame, a value should exist for `ds.attrs['principal_heading']`. The function `calc_principal_heading` is recommended for this purpose, e.g.:

  ds.attrs['principal_heading'] = dolfyn.calc_principal_heading(ds['vel'].mean(range))

  where here we are using the depth-averaged velocity to calculate the principal direction.

dolfyn.rotate.api.**calc_principal_heading**(*vel*, *tidal_mode=True*)

Compute the principal angle of the horizontal velocity.

**Parameters**

- **vel** (*np.ndarray (2,...,Nt), or (3,...,Nt)*) – The 2D or 3D velocity array (3rd-dim is ignored in this calculation)
- **tidal_mode** (*bool (default: True)*) – If true, range is set from 0 to +/-180 degrees. If false, range is 0 to 360 degrees

**Returns**

**p_heading** (*float or ndarray*) – The principal heading in degrees clockwise from North.

### Notes

When tidal_mode=True, this tool calculates the heading that is aligned with the bidirectional flow. It does so following these steps:

1. rotates vectors with negative velocity by 180 degrees
2. then doubles those angles to make a complete circle again
3. computes a mean direction from this, and halves that angle (to undo the doubled-angles in step 2)
4. The returned angle is forced to be between 0 and 180. So, you may need to add 180 to this if you want your positive direction to be in the western-half of the plane.

Otherwise, this function simply computes the average direction using a vector method.

dolfyn.rotate.api.**set_declination**(*ds*, *declin*, *inplace=True*)

Set the magnetic declination

**Parameters**

- **ds** (xarray.Dataset or *dolfyn.velocity.Velocity*) – The input dataset or velocity class
- **declination** (*float*) – The value of the magnetic declination in degrees (positive values specify that Magnetic North is clockwise from True North)
- **inplace** (*bool (default: True)*) – When True `ds` is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns a new dataset with declination set **when ``inplace=False``**, otherwise returns None.

### Notes

This function modifies the data object in the following ways:

- If the dataset is in the *earth* reference frame at the time of setting declination, it will be rotated into the "*True-East*, *True-North*, Up" (hereafter, ETU) coordinate system
- `dat['orientmat']` is modified to be an ETU to instrument (XYZ) rotation matrix (rather than the magnetic-ENU to XYZ rotation matrix). Therefore, all rotations to/from the 'earth' frame will now be to/from this ETU coordinate system.
- The value of the specified declination will be stored in `dat.attrs['declination']`
- `dat['heading']` is adjusted for declination (i.e., it is relative to True North).
- If `dat.attrs['principal_heading']` is set, it is adjusted to account for the orientation of the new 'True' earth coordinate system (i.e., calling set_declination on a data object in the principal coordinate system, then calling dat.rotate2('earth') will yield a data object in the new 'True' earth coordinate system)

dolfyn.rotate.api.**set_inst2head_rotmat**(*ds*, *rotmat*, *inplace=True*)

Set the instrument to head rotation matrix for the Nortek ADV if it hasn't already been set through a '.userdata.json' file.

**Parameters**

- **ds** (*xarray.Dataset*) – The data set to assign inst2head_rotmat
- **rotmat** (*float*) – 3x3 rotation matrix

- **inplace** (*bool (default: True)*) – When True `ds` is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns a new dataset with inst2head_rotmat set **when ``inplace=False``**, otherwise returns None.

### Notes

If the data object is in earth or principal coords, it is first rotated to 'inst' before assigning inst2head_rotmat, it is then rotated back to the coordinate system in which it was input. This way the inst2head_rotmat gets applied correctly (in inst coordinate system).

`dolfyn.rotate.base.`**euler2orient**(*heading*, *pitch*, *roll*, *units='degrees'*)

Calculate the orientation matrix from DOLfYN-defined euler angles.

This function is not likely to be called during data processing since it requires DOLfYN-defined euler angles. It is intended for testing DOLfYN.

The matrices H, P, R are the transpose of the matrices for rotation about z, y, x as shown here https://en.wikipedia.org/wiki/Rotation_matrix. The transpose is used because in DOLfYN the orientation matrix is organized for rotation from EARTH –> INST, while the wiki's matrices are organized for rotation from INST –> EARTH.

**Parameters**

- **heading** (`numpy.ndarray` (Nt)) – The heading angle.
- **pitch** (`numpy.ndarray` (Nt)) – The pitch angle.
- **roll** (`numpy.ndarray` (Nt)) – The roll angle.
- **units** (*str {'degrees' (default), 'radians'}*) –

**Returns**

**omat** (*|np.ndarray| (3x3xNt)*) – The orientation matrix of the data. The returned orientation matrix obeys the following conventions:

- a "ZYX" rotation order. That is, these variables are computed assuming that rotation from the earth -> instrument frame happens by rotating around the z-axis first (heading), then rotating around the y-axis (pitch), then rotating around the x-axis (roll). Note this requires matrix multiplication in the reverse order.
- heading is defined as the direction the x-axis points, positive clockwise from North (this is *opposite* the right-hand-rule around the Z-axis), range 0-360 degrees.
- pitch is positive when the x-axis pitches up (this is *opposite* the right-hand-rule around the Y-axis)
- roll is positive according to the right-hand-rule around the instrument's x-axis

`dolfyn.rotate.base.`**orient2euler**(*omat*)

Calculate DOLfYN-defined euler angles from the orientation matrix.

**Parameters**

**omat** (`numpy.ndarray`) – The orientation matrix

**Returns**

- **heading** (*|np.ndarray|*) – The heading angle. Heading is defined as the direction the x-axis points, positive clockwise from North (this is *opposite* the right-hand-rule around the Z-axis), range 0-360 degrees.

- **pitch** (*np.ndarray*) – The pitch angle (degrees). Pitch is positive when the x-axis pitches up (this is *opposite* the right-hand-rule around the Y-axis).
- **roll** (*np.ndarray*) – The roll angle (degrees). Roll is positive according to the right-hand-rule around the instrument's x-axis.

dolfyn.rotate.base.**quaternion2orient**(*quaternions*)

Calculate orientation from Nortek AHRS quaternions, where q = [W, X, Y, Z] instead of the standard q = [X, Y, Z, W] = [q1, q2, q3, q4]

**Parameters**
**quaternions** (*xarray.DataArray*) – Quaternion dataArray from the raw dataset

**Returns**
**orientmat** (|*np.ndarray*|) – The earth2inst rotation maxtrix as calculated from the quaternions

**See also:**

scipy.spatial.transform.Rotation

dolfyn.rotate.base.**calc_tilt**(*pitch*, *roll*)

Calculate "tilt", the vertical inclination, from pitch and roll.

**Parameters**

- **roll** (*numpy.ndarray or xarray.DataArray*) – Instrument roll
- **pitch** (*numpy.ndarray or xarray.DataArray*) – Instrument pitch

**Returns**
**tilt** (*numpy.ndarray*) – Vertical inclination of the instrument

### 1.7.5 Binning Tools

#### Velocity Analysis

Analysis in DOLfYN is primarily handled through the *VelBinner* class. Below is a list of functions that can be called from *VelBinner*.

| | |
|---|---|
| *VelBinner* | This is the base binning (averaging) tool. |
| *reshape* | Reshape the array *arr* to shape (...,n,n_bin+n_pad). |
| *detrend* | Reshape the array *arr* to shape (...,n,n_bin+n_pad) and remove the best-fit trend line from each bin. |
| *demean* | Reshape the array *arr* to shape (...,n,n_bin+n_pad) and remove the mean from each bin. |
| *mean* | Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the mean of each bin along the specified *axis*. |
| *var* | Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the variance of each bin along the specified *axis*. |
| *std* | Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the standard deviation of each bin along the specified *axis*. |
| *calc_tke* | Calculate the turbulent kinetic energy (TKE) (variances of u,v,w). |
| *calc_psd* | Calculate the power spectral density of velocity. |
| *calc_freq* | Calculate the ordinary or radial frequency vector for the PSDs |
| *calc_psd_base* | Calculate power spectral density of *dat* |
| *calc_csd_base* | Calculate the cross power spectral density of *dat*. |

### Turbulence Analysis

Functions for analyzing ADV data via the *ADVBinner* class, beyond those described in *VelBinner*.

| | |
|---|---|
| *ADVBinner* | A class that builds upon *VelBinner* for calculating turbulence statistics and velocity spectra from ADV data |
| *calc_turbulence* | Functional version of *ADVBinner* that computes a suite of turbulence statistics for the input dataset, and returns a *binned* data object. |
| *calc_csd* | Calculate the cross-spectral density of velocity components. |
| *calc_stress* | Calculate the stresses (covariances of u,v,w) |
| *calc_doppler_noise* | Calculate bias due to Doppler noise using the noise floor of the velocity spectra. |
| *calc_epsilon_LT83* | Calculate the dissipation rate from the PSD |
| *calc_epsilon_SF* | Calculate dissipation rate using the "structure function" (SF) method |
| *calc_epsilon_TE01* | Calculate the dissipation rate according to TE01. |
| *calc_L_int* | Calculate integral length scales. |

Functions for analyzing ADCP data via the *ADPBinner* class, beyond those described in *VelBinner*.

| | |
|---|---|
| *ADPBinner* | A class for calculating turbulence statistics from ADCP data |
| *calc_dudz* | The shear in the first velocity component. |
| *calc_dvdz* | The shear in the second velocity component. |
| *calc_dwdz* | The shear in the third velocity component. |
| *calc_shear2* | The horizontal shear squared. |
| *calc_doppler_noise* | Calculate bias due to Doppler noise using the noise floor of the velocity spectra. |
| *calc_stress_4beam* | Calculate the stresses from the covariance of along-beam velocity measurements |
| *calc_stress_5beam* | Calculate the stresses from the covariance of along-beam velocity measurements |
| *calc_total_tke* | Calculate magnitude of turbulent kinetic energy from 5-beam ADCP. |
| *calc_dissipation_LT83* | Calculate the TKE dissipation rate from the velocity spectra. |
| *calc_dissipation_SF* | Calculate TKE dissipation rate from ADCP along-beam velocity using the "structure function" (SF) method. |
| *calc_ustar_fit* | Approximate friction velocity from shear stress using a logarithmic profile. |

**class** dolfyn.binned.**TimeBinner**(*n_bin*, *fs*, *n_fft=None*, *n_fft_coh=None*, *noise=[0, 0, 0]*)

Bases: `object`

Initialize an averaging object

**Parameters**

- **n_bin** (*int*) – Number of data points to include in a 'bin' (ensemble), not the number of bins
- **fs** (*int*) – Instrument sampling frequency in Hz
- **n_fft** (*int*) – Number of data points to use for fft (*n_fft `<= `n_bin*). Default: *n_fft `= `n_bin*
- **n_fft_coh** (*int*) – Number of data points to use for coherence and cross-spectra ffts Default: *n_fft_coh `= `n_fft*
- **noise** (*float, list or numpy.ndarray*) – Instrument's doppler noise in same units as velocity

**reshape**(*arr*, *n_pad=0*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad).

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **n_pad** (*int*) – Is used to add *n_pad*/2 points from the end of the previous ensemble to the top of the current, and *n_pad*/2 points from the top of the next ensemble to the bottom of the current. Zeros are padded in the upper-left and lower-right corners of the matrix (beginning/end of timeseries). In this case, the array shape will be (...,`n`,`n_pad`+`n_bin`)
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*) – Data in reshaped format, where the last axis is of length *int(n_bin)*.

### Notes

*n_bin* can be non-integer, in which case the output array size will be *n_pad*`+`*n_bin*, and the decimal will cause skipping of some data points in *arr*. In particular, every mod(*n_bin*,1) bins will have a skipped point. For example: - for n_bin=2048.2 every 1/5 bins will have a skipped point. - for n_bin=4096.9 every 9/10 bins will have a skipped point.

**detrend**(*arr*, *axis=-1*, *n_pad=0*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad) and remove the best-fit trend line from each bin.

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **axis** (*int (default: -1)*) – Axis along which to take mean
- **n_pad** (*int (default: 0)*) – Is used to add *n_pad*/2 points from the end of the previous ensemble to the top of the current, and *n_pad*/2 points from the top of the next ensemble to the bottom of the current. Zeros are padded in the upper-left and lower-right corners of the matrix (beginning/end of timeseries). In this case, the array shape will be (...,`n`,`n_pad`+`n_bin`)
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*) – Detrended data, where the last axis is of length *int(n_bin)*.

**demean**(*arr*, *axis=-1*, *n_pad=0*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad) and remove the mean from each bin.

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **axis** (*int (default: -1)*) – Axis along which to take mean
- **n_pad** (*int (default: 0)*) – Is used to add *n_pad*/2 points from the end of the previous ensemble to the top of the current, and *n_pad*/2 points from the top of the next ensemble to the bottom of the current. Zeros are padded in the upper-left and lower-right corners of the matrix (beginning/end of timeseries). In this case, the array shape will be (...,`n`,`n_pad`+`n_bin`)
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*) – Demeaned data, where the last axis is of length *int(n_bin)*.

**mean**(*arr*, *axis=-1*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the mean of each bin along the specified *axis*.

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **axis** (*int (default: -1)*) – Axis along which to take mean
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*)

**var**(*arr*, *axis=-1*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the variance of each bin along the specified *axis*.

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **axis** (*int (default: -1)*) – Axis along which to take variance
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*)

**std**(*arr*, *axis=-1*, *n_bin=None*)

Reshape the array *arr* to shape (...,n,n_bin+n_pad) and take the standard deviation of each bin along the specified *axis*.

**Parameters**

- **arr** (*numpy.ndarray*) – Input data
- **axis** (*int (default: -1)*) – Axis along which to take std dev
- **n_bin** (*int (default: self.n_bin)*) – Override this binner's n_bin.

**Returns**

**out** (*numpy.ndarray*)

**calc_psd_base**(*dat*, *fs=None*, *window='hann'*, *noise=0*, *n_bin=None*, *n_fft=None*, *n_pad=None*, *step=None*)

Calculate power spectral density of *dat*

**Parameters**

- **dat** (*xarray.DataArray*) – The raw dataArray of which to calculate the psd.
- **fs** (*float (optional)*) – The sample rate (Hz).
- **window** (*str*) – String indicating the window function to use (default: 'hanning').
- **noise** (*float*) – The white-noise level of the measurement (in the same units as *dat*).
- **n_bin** (*int*) – n_bin of veldat2, number of elements per bin if 'None' is taken from VelBinner
- **n_fft** (*int*) – n_fft of veldat2, number of elements per bin if 'None' is taken from VelBinner
- **n_pad** (*int (optional)*) – The number of values to pad with zero (default: 0)
- **step** (*int (optional)*) – Controls amount of overlap in fft (default: the step size is chosen to maximize data use, minimize nens, and have a minimum of 50% overlap.).

**Returns**

**out** (*numpy.ndarray*) – The power spectral density of *dat*

### Notes

PSD's are calculated based on sample rate units

**calc_csd_base**(*dat1*, *dat2*, *fs=None*, *window='hann'*, *n_fft=None*, *n_bin=None*)

Calculate the cross power spectral density of *dat*.

**Parameters**

- **dat1** (*numpy.ndarray*) – The first (shorter, if applicable) raw dataArray of which to calculate the cpsd.
- **dat2** (*numpy.ndarray*) – The second (the shorter, if applicable) raw dataArray of which to calculate the cpsd.
- **fs** (*float (optional)*) – The sample rate (rad/s or Hz).
- **window** (*str*) – String indicating the window function to use (default: 'hanning').
- **n_fft** (*int*) – n_fft of veldat2, number of elements per bin if 'None' is taken from VelBinner
- **n_bin** (*int*) – n_bin of veldat2, number of elements per bin if 'None' is taken from VelBinner

**Returns**

**out** (*numpy.ndarray*) – The cross power spectral density of *dat1* and *dat2*

### Notes

PSD's are calculated based on sample rate units

The two velocity inputs do not have to be perfectly synchronized, but they should have the same start and end timestamps

**calc_freq**(*fs=None*, *units='rad/s'*, *n_fft=None*, *coh=False*)

Calculate the ordinary or radial frequency vector for the PSDs

**Parameters**

- **fs** (*float (optional)*) – The sample rate (Hz).
- **units** (*string*) – Frequency units in either Hz or rad/s (f or omega)
- **coh** (*bool*) – Calculate the frequency vector for coherence/cross-spectra (default: False) i.e. use self.n_fft_coh instead of self.n_fft.
- **n_fft** (*int*) – n_fft of veldat2, number of elements per bin if 'None' is taken from VelBinner

**Returns**

**out** (*numpy.ndarray*) – Spectrum frequency array in units of 'Hz' or 'rad/s'

**class** dolfyn.velocity.**Velocity**(*ds*, **args*, ***kwargs*)

Bases: object

All ADCP and ADV xarray datasets wrap this base class. The turbulence-related attributes defined within this class assume that the 'tke_vec' and 'stress_vec' data entries are included in the dataset. These are typically calculated using a *VelBinner* tool, but the method for calculating these variables can depend on the details of the measurement (instrument, it's configuration, orientation, etc.).

**See also:**

*VelBinner*

**rotate2**(*out_frame='earth'*, *inplace=True*)

Rotate the dataset to a new coordinate system.

**Parameters**

- **out_frame** (*string {'beam', 'inst', 'earth', 'principal'}*) – The coordinate system to rotate the data into.
- **inplace** (*bool (default: True)*) – When True the existing data object is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns the rotated dataset **when ``inplace=False``**, otherwise returns None.

### Notes

- This function rotates all variables in `ds.attrs['rotate_vars']`.
- To rotate to the 'principal' frame, a value of `ds.attrs['principal_heading']` must exist. The function `calc_principal_heading` is recommended for this purpose, e.g.:

```
ds.attrs['principal_heading'] = dolfyn.calc_principal_heading(ds['vel'].
↪mean(range))
```

where here we are using the depth-averaged velocity to calculate the principal direction.

**set_declination**(*declin*, *inplace=True*)

Set the magnetic declination

**Parameters**

- **declination** (*float*) – The value of the magnetic declination in degrees (positive values specify that Magnetic North is clockwise from True North)
- **inplace** (*bool (default: True)*) – When True the existing data object is modified. When False a copy is returned.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns the rotated dataset **when ``inplace=False``**, otherwise returns None.

### Notes

This method modifies the data object in the following ways:

- If the dataset is in the *earth* reference frame at the time of

setting declination, it will be rotated into the "*True-East*, *True-North*, Up" (hereafter, ETU) coordinate system

- `dat['orientmat']` is modified to be an ETU to

instrument (XYZ) rotation matrix (rather than the magnetic-ENU to XYZ rotation matrix). Therefore, all rotations to/from the 'earth' frame will now be to/from this ETU coordinate system.

- The value of the specified declination will be stored in

`dat.attrs['declination']`

- `dat['heading']` is adjusted for declination

(i.e., it is relative to True North).

- If `dat.attrs['principal_heading']` is set, it is

adjusted to account for the orientation of the new 'True' earth coordinate system (i.e., calling set_declination on a data object in the principal coordinate system, then calling dat.rotate2('earth') will yield a data object in the new 'True' earth coordinate system)

**set_inst2head_rotmat**(*rotmat*, *inplace=True*)

Set the instrument to head rotation matrix for the Nortek ADV if it hasn't already been set through a '.userdata.json' file.

**Parameters**

- **rotmat** (*float*) – 3x3 rotation matrix
- **inplace** (*bool (default: True)*) – When True the existing data object is rotated. When False a copy is returned that is rotated.

**Returns**

**ds** (*xarray.Dataset or None*) – Returns the rotated dataset **when ``inplace=False``**, otherwise returns None.

### Notes

If the data object is in earth or principal coords, it is first rotated to 'inst' before assigning inst2head_rotmat, it is then rotated back to the coordinate system in which it was input. This way the inst2head_rotmat gets applied correctly (in inst coordinate system).

**save**(*filename*, ***kwargs*)

Save the data object (underlying xarray dataset) as netCDF (.nc).

**Parameters**

- **filename** (*str*) – Filename and/or path with the '.nc' extension
- ****kwargs** (these are passed directly to `xarray.Dataset.to_netcdf()`.) –

### Notes

See DOLfYN's *save* function for additional details.

**property variables**

A sorted list of the variable names in the dataset.

**property attrs**

The attributes in the dataset.

**property coords**

The coordinates in the dataset.

**property u**

The first velocity component.

This is simply a shortcut to self['vel'][0]. Therefore, depending on the coordinate system of the data object (self.attrs['coord_sys']), it is:

- beam: beam1
- inst: x
- earth: east
- principal: streamwise

**property v**

The second velocity component.

This is simply a shortcut to self['vel'][1]. Therefore, depending on the coordinate system of the data object (self.attrs['coord_sys']), it is:

- beam: beam2
- inst: y
- earth: north
- principal: cross-stream

**property w**

The third velocity component. This is simply a shortcut to self['vel'][2]. Therefore, depending on the coordinate system of the data object (self.attrs['coord_sys']), it is: - beam: beam3 - inst: z - earth: up - principal: up

**property U**

Horizontal velocity as a complex quantity

**property U_mag**

Horizontal velocity magnitude

**property U_dir**

Angle of horizontal velocity vector. Direction is 'to', as opposed to 'from'. This function calculates angle as "degrees CCW from X/East/streamwise" and then converts it to "degrees CW from X/North/streamwise".

**property E_coh**

Coherent turbulence energy

Niel Kelley's 'coherent turbulence energy', which is the RMS of the Reynold's stresses.

See: NREL Technical Report TP-500-52353

**property I_tke**

Turbulent kinetic energy intensity.

Ratio of sqrt(tke) to horizontal velocity magnitude.

**property I**

Turbulence intensity.

Ratio of standard deviation of horizontal velocity to horizontal velocity magnitude.

**property tke**

Turbulent kinetic energy (sum of the three components)

**property upvp_**

u'v'bar Reynolds stress

**property upwp_**

u'w'bar Reynolds stress

**property vpwp_**

v'w'bar Reynolds stress

**property upup_**

u'u'bar component of the tke

**property vpvp_**

v'v'bar component of the tke

**property wpwp_**

w'w'bar component of the tke

**class** dolfyn.velocity.**VelBinner**(*n_bin*, *fs*, *n_fft=None*, *n_fft_coh=None*, *noise=[0, 0, 0]*)

Bases: *TimeBinner*

This is the base binning (averaging) tool. All DOLfYN binning tools derive from this base class.

### Examples

The VelBinner class is used to compute averages and turbulence statistics from 'raw' (not averaged) ADV or ADP measurements, for example:

```
# First read or load some data.
rawdat = dolfyn.read_example('BenchFile01.ad2cp')

# Now initialize the averaging tool:
binner = dolfyn.VelBinner(n_bin=600, fs=rawdat.fs)

# This computes the basic averages
avg = binner.do_avg(rawdat)
```

Initialize an averaging object

**Parameters**

- **n_bin** (*int*) – Number of data points to include in a 'bin' (ensemble), not the number of bins
- **fs** (*int*) – Instrument sampling frequency in Hz
- **n_fft** (*int*) – Number of data points to use for fft (*n_fft* `<=` *n_bin*). Default: *n_fft* `=` *n_bin*
- **n_fft_coh** (*int*) – Number of data points to use for coherence and cross-spectra ffts Default: *n_fft_coh* `=` *n_fft*
- **noise** (*float, list or numpy.ndarray*) – Instrument's doppler noise in same units as velocity

**tke = <xarray.DataArray 'tke' (tke: 3)> array(['upup_', 'vpvp_', 'wpwp_'], dtype='<U5') Dimensions without coordinates: tke Attributes: units: 1 long_name: Turbulent Kinetic Energy Vector Components coverage_content_type: coordinate**

tau = <xarray.DataArray 'tau' (tau: 3)> array(['upvp_', 'upwp_', 'vpwp_'], dtype='<U5') Dimensions without coordinates: tau Attributes: units: 1 long_name: Reynolds Stress Vector Components coverage_content_type: coordinate

S = <xarray.DataArray 'S' (S: 3)> array(['Sxx', 'Syy', 'Szz'], dtype='<U3') Dimensions without coordinates: S Attributes: units: 1 long_name: Power Spectral Density Vector Components coverage_content_type: coordinate

C = <xarray.DataArray 'C' (C: 3)> array(['Cxy', 'Cxz', 'Cyz'], dtype='<U3') Dimensions without coordinates: C Attributes: units: 1 long_name: Cross-Spectral Density Vector Components coverage_content_type: coordinate

**do_avg**(*raw_ds*, *out_ds=None*, *names=None*)

Bin the dataset and calculate the ensemble averages of each variable.

**Parameters**

- **raw_ds** (*xarray.Dataset*) – The raw data structure to be binned
- **out_ds** (*xarray.Dataset*) – The bin'd (output) data object to which averaged data is added.
- **names** (*list of strings*) – The names of variables to be averaged. If *names* is None, all data in *raw_ds* will be binned.

**Returns**

**out_ds** (*xarray.Dataset*) – The new (or updated when out_ds is not None) dataset with the averages of all the variables in raw_ds.

:raises AttributeError : when out_ds is supplied as input (not None): :raises and the values in out_ds.attrs are inconsistent with: :raises raw_ds.attrs or the properties of this VelBinner (n_bin,: :raises n_fft, fs, etc.):

### Notes

raw_ds.attrs are copied to out_ds.attrs. Inconsistencies between the two (when out_ds is specified as input) raise an AttributeError.

**do_var**(*raw_ds*, *out_ds=None*, *names=None*, *suffix='_var'*)

Bin the dataset and calculate the ensemble variances of each variable. Complementary to *do_avg()*.

**Parameters**

- **raw_ds** (*xarray.Dataset*) – The raw data structure to be binned.
- **out_ds** (*xarray.Dataset*) – The binned (output) dataset to which variance data is added, nominally dataset output from *do_avg()*
- **names** (*list of strings*) – The names of variables of which to calculate variance. If *names* is None, all data in *raw_ds* will be binned.

**Returns**

**out_ds** (*xarray.Dataset*) – The new (or updated when out_ds is not None) dataset with the variance of all the variables in raw_ds.

:raises AttributeError : when out_ds is supplied as input (not None): :raises and the values in out_ds.attrs are inconsistent with: :raises raw_ds.attrs or the properties of this VelBinner (n_bin,: :raises n_fft, fs, etc.):

### Notes

raw_ds.attrs are copied to out_ds.attrs. Inconsistencies between the two (when out_ds is specified as input) raise an AttributeError.

**calc_coh**(*veldat1*, *veldat2*, *window='hann'*, *debias=True*, *noise=(0, 0)*, *n_fft_coh=None*, *n_bin=None*)

Calculate coherence between *veldat1* and *veldat2*.

**Parameters**

- **veldat1** (*xarray.DataArray*) – The first (the longer, if applicable) raw dataArray of which to calculate coherence
- **veldat2** (*xarray.DataArray*) – The second (the shorter, if applicable) raw dataArray of which to calculate coherence
- **window** (*str*) – String indicating the window function to use (default: 'hanning')
- **noise** (*float*) – The white-noise level of the measurement (in the same units as *veldat*).
- **n_fft_coh** (*int*) – n_fft of veldat2, number of elements per bin if 'None' is taken from VelBinner
- **n_bin** (*int*) – n_bin of veldat2, number of elements per bin if 'None' is taken from VelBinner

**Returns**

**da** (*xarray.DataArray*) – The coherence between signal veldat1 and veldat2.

### Notes

The two velocity inputs do not have to be perfectly synchronized, but they should have the same start and end timestamps.

**calc_phase_angle**(*veldat1*, *veldat2*, *window='hann'*, *n_fft_coh=None*, *n_bin=None*)

Calculate the phase difference between two signals as a function of frequency (complimentary to coherence).

**Parameters**

- **veldat1** (*xarray.DataArray*) – The first (the longer, if applicable) raw dataArray of which to calculate phase angle
- **veldat2** (*xarray.DataArray*) – The second (the shorter, if applicable) raw dataArray of which to calculate phase angle
- **window** (*str*) – String indicating the window function to use (default: 'hanning').
- **n_fft** (*int*) – Number of elements per bin if 'None' is taken from VelBinner
- **n_bin** (*int*) – Number of elements per bin from veldat2 if 'None' is taken from VelBinner

**Returns**

**da** (*xarray.DataArray*) – The phase difference between signal veldat1 and veldat2.

### Notes

The two velocity inputs do not have to be perfectly synchronized, but they should have the same start and end timestamps.

**calc_acov**(*veldat*, *n_bin=None*)

Calculate the auto-covariance of the raw-signal *veldat*

**Parameters**

- **veldat** (*xarray.DataArray*) – The raw dataArray of which to calculate auto-covariance
- **n_bin** (*float*) – Number of data elements to use

**Returns**
**da** (*xarray.DataArray*) – The auto-covariance of veldat

### Notes

As opposed to calc_xcov, which returns the full cross-covariance between two arrays, this function only returns a quarter of the full auto-covariance. It computes the auto-covariance over half of the range, then averages the two sides (to return a 'quartered' covariance).

This has the advantage that the 0 index is actually zero-lag.

**calc_xcov**(*veldat1*, *veldat2*, *npt=1*, *n_bin=None*, *normed=False*)

Calculate the cross-covariance between arrays veldat1 and veldat2

**Parameters**

- **veldat1** (*xarray.DataArray*) – The first raw dataArray of which to calculate cross-covariance
- **veldat2** (*xarray.DataArray*) – The second raw dataArray of which to calculate cross-covariance
- **npt** (*int*) – Number of timesteps (lag) to calculate covariance
- **n_fft** (*int*) – n_fft of veldat2, number of elements per bin if 'None' is taken from VelBinner
- **n_bin** (*int*) – n_bin of veldat2, number of elements per bin if 'None' is taken from VelBinner

**Returns**
**da** (*xarray.DataArray*) – The cross-covariance between signal veldat1 and veldat2.

### Notes

The two velocity inputs must be the same length

**calc_tke**(*veldat*, *noise=None*, *detrend=True*)

Calculate the turbulent kinetic energy (TKE) (variances of u,v,w).

**Parameters**

- **veldat** (*xarray.DataArray*) – Velocity data array from ADV or single beam from ADCP. The last dimension is assumed to be time.
- **noise** (*float or array-like*) – A vector of the noise levels of the velocity data with the same first dimension as the velocity vector.

- **detrend** (*bool (default: False)*) – Detrend the velocity data (True), or simply demean it (False), prior to computing tke. Note: the psd routines use detrend, so if you want to have the same amount of variance here as there use `detrend=True`.

**Returns**
**tke_vec** (*xarray.DataArray*) – dataArray containing u'u'_, v'v'_ and w'w'_

`calc_psd`(*veldat*, *freq_units='rad/s'*, *fs=None*, *window='hann'*, *noise=None*, *n_bin=None*, *n_fft=None*, *n_pad=None*, *step=None*)

Calculate the power spectral density of velocity.

**Parameters**

- **veldat** (*xr.DataArray*) – The raw velocity data (of dims 'dir' and 'time').
- **freq_units** (*string*) – Frequency units of the returned spectra in either Hz or rad/s ($f$ or $\omega$)
- **fs** (*float (optional)*) – The sample rate (default: from the binner).
- **window** (*string or array*) – Specify the window function. Options: 1, None, 'hann', 'hamm'
- **noise** (*float or array-like*) – A vector of the noise levels of the velocity data with the same first dimension as the velocity vector.
- **n_bin** (*int (optional)*) – The bin-size (default: from the binner).
- **n_fft** (*int (optional)*) – The fft size (default: from the binner).
- **n_pad** (*int (optional)*) – The number of values to pad with zero (default: 0)
- **step** (*int (optional)*) – Controls amount of overlap in fft (default: the step size is chosen to maximize data use, minimize nens, and have a minimum of 50% overlap.).

**Returns**
**psd** (*xarray.DataArray (3, M, N_FFT)*) – The spectra in the 'u', 'v', and 'w' directions.

**class** dolfyn.adv.turbulence.**ADVBinner**(*n_bin*, *fs*, *n_fft=None*, *n_fft_coh=None*, *noise=[0, 0, 0]*)

Bases: *VelBinner*

A class that builds upon *VelBinner* for calculating turbulence statistics and velocity spectra from ADV data

**Parameters**

- **n_bin** (*int*) – The length of each *bin*, in number of points, for this averaging operator.
- **fs** (*int*) – Instrument sampling frequency in Hz
- **n_fft** (*int (optional, default: n_fft = n_bin)*) – The length of the FFT for computing spectra (must be <= n_bin)
- **n_fft_coh** (int (optional, default: *n_fft_coh* = *n_fft*)) – Number of data points to use for coherence and cross-spectra ffts
- **noise** (*float, list or numpy.ndarray*) – Instrument's doppler noise in same units as velocity

Initialize an averaging object

**Parameters**

- **n_bin** (*int*) – Number of data points to include in a 'bin' (ensemble), not the number of bins
- **fs** (*int*) – Instrument sampling frequency in Hz

- **n_fft** (*int*) – Number of data points to use for fft (*n_fft <= n_bin*). Default: *n_fft = n_bin*
- **n_fft_coh** (*int*) – Number of data points to use for coherence and cross-spectra ffts Default: *n_fft_coh = n_fft*
- **noise** (*float, list or numpy.ndarray*) – Instrument's doppler noise in same units as velocity

**__call__**(*ds*, *freq_units='rad/s'*, *window='hann'*)

Call self as a function.

**calc_stress**(*veldat*, *detrend=True*)

Calculate the stresses (covariances of u,v,w)

**Parameters**

- **veldat** (*xr.DataArray*) – Velocity data array from ADV data. The last dimension is assumed to be time.
- **detrend** (*bool (default: True)*) – detrend the velocity data (True), or simply demean it (False), prior to computing stress. Note: the psd routines use detrend, so if you want to have the same amount of variance here as there use `detrend=True`.

**Returns**

**out** (*xarray.DataArray*)

**calc_csd**(*veldat*, *freq_units='rad/s'*, *fs=None*, *window='hann'*, *n_bin=None*, *n_fft_coh=None*)

Calculate the cross-spectral density of velocity components.

**Parameters**

- **veldat** (*xarray.DataArray*) – The raw 3D velocity data.
- **freq_units** (*string*) – Frequency units of the returned spectra in either Hz or rad/s ($f$ or $\omega$)
- **fs** (*float (optional)*) – The sample rate (default: from the binner).
- **window** (*string or array*) – Specify the window function. Options: 1, None, 'hann', 'hamm'
- **n_bin** (*int (optional)*) – The bin-size (default: from the binner).
- **n_fft_coh** (*int (optional)*) – The fft size (default: n_fft_coh from the binner).

**Returns**

**csd** (*xarray.DataArray (3, M, N_FFT)*) – The first-dimension of the cross-spectrum is the three different cross-spectra: 'uv', 'uw', 'vw'.

**calc_doppler_noise**(*psd*, *pct_fN=0.8*)

Calculate bias due to Doppler noise using the noise floor of the velocity spectra.

**Parameters**

- **psd** (*xarray.DataArray (dir, time, f)*) – The ADV velocity spectra
- **pct_fN** (*float*) – Percent of Nyquist frequency to calculate characeristic frequency

**Returns**

*doppler_noise (xarray.DataArray)* – Doppler noise level in units of m/s

### Notes

Approximates bias from

where :math: *sigma_{noise}* is the bias due to Doppler noise, *N* is the constant variance or spectral density, and *f_{c}* is the characteristic frequency.

The characteristic frequency is then found as

where *f_{s}/2* is the Nyquist frequency.

Richard, Jean-Baptiste, et al. "Method for identification of Doppler noise levels in turbulent flow measurements dedicated to tidal energy." International Journal of Marine Energy 3 (2013): 52-64.

Thiébaut, Maxime, et al. "Investigating the flow dynamics and turbulence at a tidal-stream energy site in a highly energetic estuary." Renewable Energy 195 (2022): 252-262.

**check_turbulence_cascade_slope**(*psd*, *freq_range=[6.28, 12.57]*)

This function calculates the slope of the PSD, the power spectra of velocity, within the given frequency range. The purpose of this function is to check that the region of the PSD containing the isotropic turbulence cascade decreases at a rate of $f^{-5/3}$.

**Parameters**

- **psd** (*xarray.DataArray ([time,] freq)*) – The power spectral density (1D or 2D)
- **freq_range** (*iterable(2) (default: [6.28, 12.57])*) – The range over which the isotropic turbulence cascade occurs, in units of the psd frequency vector (Hz or rad/s)

**Returns**

**(m, b)** (*tuple (slope, y-intercept)*) – A tuple containing the coefficients of the log-adjusted linear regression between PSD and frequency

### Notes

Calculates slope based on the *standard* formula for dissipation:

$$S(k) = \alpha\epsilon^{2/3}k^{-5/3} + N$$

The slope of the isotropic turbulence cascade, which should be equal to $k^{-5/3}$ or $f^{-5/3}$, where k and f are the wavenumber and frequency vectors, is estimated using linear regression with a log transformation:

$$log10(y) = m * log10(x) + b$$

Which is equivalent to

$$y = 10^b x^m$$

Where $y$ is S(k) or S(f), $x$ is k or f, $m$ is the slope (ideally -5/3), and $10^b$ is the intercept of y at x^m=1.

**calc_epsilon_LT83**(*psd*, *U_mag*, *freq_range=[6.28, 12.57]*)

Calculate the dissipation rate from the PSD

**Parameters**

- **psd** (*xarray.DataArray (...,time,freq)*) – The power spectral density
- **U_mag** (*xarray.DataArray (...,time)*) – The bin-averaged horizontal velocity [m s-1] (from dataset shortcut)

- **freq_range** (*iterable(2) (default: [6.28, 12.57])*) – The range over which to integrate/average the spectrum, in units of the psd frequency vector (Hz or rad/s)

**Returns**
**epsilon** (*xarray.DataArray (…,n_time)*) – dataArray of the dissipation rate

### Notes

This uses the *standard* formula for dissipation:

$$S(k) = \alpha \epsilon^{2/3} k^{-5/3} + N$$

where $\alpha = 0.5$ (1.5 for all three velocity components), *k* is wavenumber, *S(k)* is the turbulent kinetic energy spectrum, and `N' is the doppler noise level associated with the TKE spectrum.

With $k \rightarrow \omega/U$, then – to preserve variance – $S(k) = US(\omega)$, and so this becomes:

$$S(\omega) = \alpha \epsilon^{2/3} \omega^{-5/3} U^{2/3} + N$$

With $k \rightarrow (2\pi f)/U$, then

$$S(\omega) = \alpha \epsilon^{2/3} f^{-5/3} (U/(2\pi))^{2/3} + N$$

LT83 : Lumley and Terray, "Kinematics of turbulence convected by a random wave field". JPO, 1983, vol13, pp2000-2007.

**calc_epsilon_SF**(*vel_raw*, *U_mag*, *fs=None*, *freq_range=[2.0, 4.0]*)

Calculate dissipation rate using the "structure function" (SF) method

**Parameters**

- **vel_raw** (*xarray.DataArray*) – The raw velocity data (1D dimension time) upon which to perform the SF technique.
- **U_mag** (*xarray.DataArray*) – The bin-averaged horizontal velocity (from dataset shortcut)
- **fs** (*float*) – The sample rate of *vel_raw* [Hz]
- **freq_range** (*iterable(2) (default: [2., 4.])*) – The frequency range over which to compute the SF [Hz] (i.e. the frequency range within which the isotropic turbulence cascade falls)

**Returns**
**epsilon** (*xarray.DataArray*) – dataArray of the dissipation rate

**calc_epsilon_TE01**(*dat_raw*, *dat_avg*, *freq_range=[6.28, 12.57]*)

Calculate the dissipation rate according to TE01.

**Parameters**

- **dat_raw** (*xarray.Dataset*) – The raw (off the instrument) adv dataset
- **dat_avg** (*xarray.Dataset*) – The bin-averaged adv dataset (calc'd from 'calc_turbulence' or 'do_avg'). The spectra (psd) and basic turbulence statistics ('tke_vec' and 'stress_vec') must already be computed.
- **freq_range** (*iterable(2) (default: [6.28, 12.57])*) – The range over which to integrate/average the spectrum, in units of the psd frequency vector (Hz or rad/s)

### Notes

TE01 : Trowbridge, J and Elgar, S, "Turbulence measurements in the Surf Zone". JPO, 2001, vol31, pp2403-2417.

**calc_L_int**(*a_cov*, *U_mag*, *fs=None*)

Calculate integral length scales.

**Parameters**

- **a_cov** (*xarray.DataArray*) – The auto-covariance array (i.e. computed using *calc_acov*).
- **U_mag** (*xarray.DataArray*) – The bin-averaged horizontal velocity (from dataset shortcut)
- **fs** (*float*) – The raw sample rate

**Returns**

**L_int** (*|np.ndarray| (..., n_time)*) – The integral length scale (T_int*U_mag).

### Notes

The integral time scale (T_int) is the lag-time at which the auto-covariance falls to 1/e.

If T_int is not reached, L_int will default to '0'.

dolfyn.adv.turbulence.**calc_turbulence**(*ds_raw*, *n_bin*, *fs*, *n_fft=None*, *freq_units='rad/s'*, *window='hann'*)

Functional version of *ADVBinner* that computes a suite of turbulence statistics for the input dataset, and returns a *binned* data object.

**Parameters**

- **ds_raw** (*xarray.Dataset*) – The raw adv datset to *bin*, average and compute turbulence statistics of.
- **freq_units** (*string (default: rad/s)*) – Frequency units of the returned spectra in either Hz or rad/s ($f$ or $\omega$)
- **window** (*1, None, 'hann', 'hamm'*) – The window to use for calculating power spectral densities

**Returns**

**ds** (*xarray.Dataset*) – Returns an 'binned' (i.e. 'averaged') data object. All fields (variables) of the input data object are averaged in n_bin chunks. This object also computes the following items over those chunks:

- tke_vec : The energy in each component, each components is alternatively accessible as: *upup_*, *vpvp_*, *wpwp_*)
- stress_vec : The Reynolds stresses, each component is alternatively accessible as: upwp_, vpwp_, upvp_)
- U_std : The standard deviation of the horizontal velocity *U_mag*.
- psd : DataArray containing the spectra of the velocity in radial frequency units. The dataarray contains: - vel : the velocity spectra array (m^2/s/rad)) - omega : the radial frequncy (rad/s)

class dolfyn.adp.turbulence.**ADPBinner**(*n_bin*, *fs*, *n_fft=None*, *n_fft_coh=None*, *noise=None*, *orientation='up'*, *diff_style='centered_extended'*)

Bases: *VelBinner*

A class for calculating turbulence statistics from ADCP data

**Parameters**

- **n_bin** (*int*) – Number of data points to include in a 'bin' (ensemble), not the number of bins
- **fs** (*int*) – Instrument sampling frequency in Hz
- **n_fft** (*int*) – Number of data points to use for fft (*n_fft* `<= `*n_bin*). Default: *n_fft* `= `*n_bin*
- **n_fft_coh** (*int*) – Number of data points to use for coherence and cross-spectra ffts Default: *n_fft_coh* `= `*n_fft*
- **noise** (*float, list or numpy.ndarray*) – Instrument's doppler noise in same units as velocity
- **orientation** (*str, default='up'*) – Instrument's orientation, either 'up' or 'down'
- **diff_style** (*str, default='centered_extended'*) – Style of numerical differentiation using Newton's Method. Either 'first' (first difference), 'centered' (centered difference), or 'centered_extended' (centered difference with first and last points

  extended using a first difference).

**calc_dudz**(*vel*, *orientation=None*)

The shear in the first velocity component.

**Parameters**

- **vel** (*xarray.DataArray*) – ADCP raw velocity
- **orientation** (*str, default=ADPBinner.orientation*) – Direction ADCP is facing ('up' or 'down')

### Notes

The derivative direction is along the profiler's 'z' coordinate ('dz' is actually diff(self['range'])), not necessarily the 'true vertical' direction.

**calc_dvdz**(*vel*)

The shear in the second velocity component.

**Parameters**
**vel** (*xarray.DataArray*) – ADCP raw velocity

### Notes

The derivative direction is along the profiler's 'z' coordinate ('dz' is actually diff(self['range'])), not necessarily the 'true vertical' direction.

**calc_dwdz**(*vel*)

The shear in the third velocity component.

**Parameters**
**vel** (*xarray.DataArray*) – ADCP raw velocity

### Notes

The derivative direction is along the profiler's 'z' coordinate ('dz' is actually diff(self['range'])), not necessarily the 'true vertical' direction.

`calc_shear2`(*vel*)

The horizontal shear squared.

**Parameters**
**vel** (*xarray.DataArray*) – ADCP raw velocity

### Notes

This is actually (dudz)^2 + (dvdz)^2. So, if those variables are not actually vertical derivatives of the horizontal velocity, then this is not the 'horizontal shear squared'.

**See also:**

$dudz$, $dvdz$

`calc_doppler_noise`(*psd*, *pct_fN=0.8*)

Calculate bias due to Doppler noise using the noise floor of the velocity spectra.

**Parameters**

- **psd** (*xarray.DataArray (time, f)*) – The velocity spectra from a single depth bin (range), typically in the mid-water range
- **pct_fN** (*float*) – Percent of Nyquist frequency to calculate characeristic frequency

**Returns**
*doppler_noise (xarray.DataArray)* – Doppler noise level in units of m/s

### Notes

Approximates bias from

where :math: *sigma_{noise}* is the bias due to Doppler noise, *N* is the constant variance or spectral density, and *f_{c}* is the characteristic frequency.

The characteristic frequency is then found as

where *f_{s}/2* is the Nyquist frequency.

Richard, Jean-Baptiste, et al. "Method for identification of Doppler noise levels in turbulent flow measurements dedicated to tidal energy." International Journal of Marine Energy 3 (2013): 52-64.

Thiébaut, Maxime, et al. "Investigating the flow dynamics and turbulence at a tidal-stream energy site in a highly energetic estuary." Renewable Energy 195 (2022): 252-262.

`calc_stress_4beam`(*ds*, *noise=None*, *orientation=None*, *beam_angle=None*)

Calculate the stresses from the covariance of along-beam velocity measurements

**Parameters**

- **ds** (*xarray.Dataset*) – Raw dataset in beam coordinates
- **noise** (*int or xarray.DataArray (time)*) – Doppler noise level in units of m/s
- **orientation** (*str, default=ds.attrs['orientation']*) – Direction ADCP is facing ('up' or 'down')

- **beam_angle** (*int, default=ds.attrs['beam_angle']*) – ADCP beam angle in units of degrees

**Returns**

**stress_vec** (*xarray.DataArray(s)*) – Stress vector with u'w'_ and v'w'_ components

### Notes

Assumes zero mean pitch and roll.

Assumes ADCP instrument coordinate system is aligned with principal flow directions.

Stacey, Mark T., Stephen G. Monismith, and Jon R. Burau. "Measurements of Reynolds stress profiles in unstratified tidal flow." Journal of Geophysical Research: Oceans 104.C5 (1999): 10933-10949.

**calc_stress_5beam**(*ds*, *noise=None*, *orientation=None*, *beam_angle=None*, *tke_only=False*)

Calculate the stresses from the covariance of along-beam velocity measurements

**Parameters**

- **ds** (*xarray.Dataset*) – Raw dataset in beam coordinates
- **noise** (*int or xarray.DataArray, default=0 (time)*) – Doppler noise level in units of m/s
- **orientation** (*str, default=ds.attrs['orientation']*) – Direction ADCP is facing ('up' or 'down')
- **beam_angle** (*int, default=ds.attrs['beam_angle']*) – ADCP beam angle in units of degrees
- **tke_only** (*bool, default=False*) – If true, only calculates tke components

**Returns**

**tke_vec(, stress_vec)** (*xarray.DataArray or tuple[xarray.DataArray]*) – If tke_only is set to False, function returns *tke_vec* and *stress_vec*. Otherwise only *tke_vec* is returned

### Notes

Assumes small-angle approximation is applicable.

Assumes ADCP instrument coordinate system is aligned with principal flow directions, i.e. u', v' and w' are aligned to the instrument's (XYZ) frame of reference.

The stress equations here utilize u'v'_ to account for small variations in pitch and roll. u'v'_ cannot be directly calculated by a 5-beam ADCP, so it is approximated by the covariance of *u* and *v*. The uncertainty introduced by using this approximation is small if deviations from pitch and roll are small (< 10 degrees).

Dewey, R., and S. Stringer. "Reynolds stresses and turbulent kinetic energy estimates from various ADCP beam configurations: Theory." J. of Phys. Ocean (2007): 1-35.

Guerra, Maricarmen, and Jim Thomson. "Turbulence measurements from five-beam acoustic Doppler current profilers." Journal of Atmospheric and Oceanic Technology 34.6 (2017): 1267-1284.

**calc_total_tke**(*ds*, *noise=None*, *orientation=None*, *beam_angle=None*)

Calculate magnitude of turbulent kinetic energy from 5-beam ADCP.

**Parameters**

- **ds** (*xarray.Dataset*) – Raw dataset in beam coordinates
- **ds_avg** (*xarray.Dataset*) – Binned dataset in final coordinate reference frame

- **noise** (*int or xarray.DataArray, default=0 (time)*) – Doppler noise level in units of m/s
- **orientation** (*str, default=ds.attrs['orientation']*) – Direction ADCP is facing ('up' or 'down')
- **beam_angle** (*int, default=ds.attrs['beam_angle']*) – ADCP beam angle in units of degrees

**Returns**

**tke** (*xarray.DataArray*) – Turbulent kinetic energy magnitude

### Notes

This function is a wrapper around 'calc_stress_5beam' that then combines the TKE components.

Warning: the integral length scale of turbulence captured by the ADCP measurements (i.e. the size of turbulent structures) increases with increasing range from the instrument.

**check_turbulence_cascade_slope**(*psd*, *freq_range=[0.2, 0.4]*)

This function calculates the slope of the PSD, the power spectra of velocity, within the given frequency range. The purpose of this function is to check that the region of the PSD containing the isotropic turbulence cascade decreases at a rate of $f^{-5/3}$.

**Parameters**

- **psd** (*xarray.DataArray ([[range,] time,] freq)*) – The power spectral density (1D, 2D or 3D)
- **freq_range** (*iterable(2) (default: [6.28, 12.57])*) – The range over which the isotropic turbulence cascade occurs, in units of the psd frequency vector (Hz or rad/s)

**Returns**

**(m, b)** (*tuple (slope, y-intercept)*) – A tuple containing the coefficients of the log-adjusted linear regression between PSD and frequency

### Notes

Calculates slope based on the *standard* formula for dissipation:

$$S(k) = \alpha\epsilon^{2/3}k^{-5/3} + N$$

The slope of the isotropic turbulence cascade, which should be equal to $k^{-5/3}$ or $f^{-5/3}$, where k and f are the wavenumber and frequency vectors, is estimated using linear regression with a log transformation:

$$log10(y) = m * log10(x) + b$$

Which is equivalent to

$$y = 10^b x^m$$

Where $y$ is S(k) or S(f), $x$ is k or f, $m$ is the slope (ideally -5/3), and $10^b$ is the intercept of y at x^m=1.

**calc_dissipation_LT83**(*psd*, *U_mag*, *freq_range=[0.2, 0.4]*)

Calculate the TKE dissipation rate from the velocity spectra.

**Parameters**

- **psd** (*xarray.DataArray (time,f)*) – The power spectral density from a single depth bin (range)
- **U_mag** (*xarray.DataArray (time)*) – The bin-averaged horizontal velocity (a.k.a. speed) from a single depth bin (range)
- **noise** (*int or xarray.DataArray, default=0 (time)*) – Doppler noise level in units of m/s
- **f_range** (*iterable(2)*) – The range over which to integrate/average the spectrum, in units of the psd frequency vector (Hz or rad/s)

**Returns**

**dissipation_rate** (*xarray.DataArray (…,n_time)*) – Turbulent kinetic energy dissipation rate

### Notes

This uses the *standard* formula for dissipation:

$$S(k) = \alpha\epsilon^{2/3}k^{-5/3} + N$$

where $\alpha = 0.5$ (1.5 for all three velocity components), *k* is wavenumber, *S(k)* is the turbulent kinetic energy spectrum, and `N' is the doppler noise level associated with the TKE spectrum.

With $k \to \omega/U$, then – to preserve variance – $S(k) = US(\omega)$, and so this becomes:

$$S(\omega) = \alpha\epsilon^{2/3}\omega^{-5/3}U^{2/3} + N$$

With $k \to (2\pi f)/U$, then

$$S(\omega) = \alpha\epsilon^{2/3}f^{-5/3}(U/(2*\pi))^{2/3} + N$$

LT83 : Lumley and Terray, "Kinematics of turbulence convected by a random wave field". JPO, 1983, vol13, pp2000-2007.

**calc_dissipation_SF**(*vel_raw*, *r_range=[1, 5]*)

Calculate TKE dissipation rate from ADCP along-beam velocity using the "structure function" (SF) method.

**Parameters**

- **vel_raw** (*xarray.DataArray*) – The raw beam velocity data (one beam, last dimension time) upon which to perform the SF technique.
- **r_range** (*numeric, default=[1,5]*) – Range of r in [m] to calc dissipation across. Low end of range should be bin size, upper end of range is limited to the length of largest eddies in the inertial subrange.

**Returns**

- **dissipation_rate** (*xarray.DataArray (range, time)*) – Dissipation rate estimated from the structure function
- **noise** (*xarray.DataArray (range, time)*) – Noise offset estimated from the structure function at r = 0
- **structure_function** (*xarray.DataArray (range, r, time)*) – Structure function D(z,r)

#### Notes

Dissipation rate outputted by this function is only valid if the isotropic turbulence cascade can be seen in the TKE spectra.

Velocity data must be in beam coordinates and should be cleaned of surface interference.

This method calculates the 2nd order structure function:

$$D(z,r) = [(u'(z) - u`(z+r))^2]$$

where *u'* is the velocity fluctuation *z* is the depth bin, *r* is the separation between depth bins, and [] denotes a time average (size 'ADPBinner.n_bin').

The stucture function can then be used to estimate the dissipation rate:

$$D(z,r) = C^2 \epsilon^{2/3} r^{2/3} + N$$

where *C* is a constant (set to 2.1), *epsilon* is the dissipation rate, and *N* is the offset due to noise. Noise is then calculated by

$$\sigma = (N/2)^{1/2}$$

Wiles, et al, "A novel technique for measuring the rate of turbulent dissipation in the marine environment" GRL, 2006, 33, L21608.

**calc_ustar_fit**(*ds_avg*, *upwp_*, *z_inds=slice(1, 5, None)*, *H=None*)

Approximate friction velocity from shear stress using a logarithmic profile.

**Parameters**

- **ds_avg** (*xarray.Dataset*) – Bin-averaged dataset containing *stress_vec*
- **upwp** (*xarray.DataArray*) – First component of Reynolds shear stress vector, "u-prime v-prime bar" Ex *ds_avg['stress_vec'].sel(tau='upwp_')*
- **z_inds** (*slice(int,int)*) – Depth indices to use for profile. Default = slice(1, 5)
- **H** (*int (default=`ds_avg.depth`)*) – Total water depth

**Returns**

**u_star** (*xarray.DataArray*) – Friction velocity

### 1.7.6 Data Shortcuts (Properties)

DOLfYN datasets also contain shortcuts to other variables that can be obtained from simple operations of its data items. Certain shortcuts require variables calculated using the *DOLfYN API*.

Table 1.4: Notes on common properties found in DOLfYN data objects.

| Name | units | Description/Notes |
|---|---|---|
| u | m/s | `dat['vel'][0]` |
| v | m/s | `dat['vel'][1]` |
| w | m/s | `dat['vel'][2]` |
| U | m/s | Horizontal velocity as a complex quantity (u + 1j * v) |
| U_mag | m/s | Magnitude of the horizontal velocity |
| U_dir | deg | Direction of the horizontal velocity (CCW from X, East, or streamwise direction, depending on coordinate system) |
| I | — | Turbulence Intensity: ratio of horizontal velocity standard deviation (*U_std*) to mean ('U_mag') |
| I_tke | — | TKE Intensity: Ratio of sqrt(2*tke) to horizontal velocity magnitude |
| tke | $m^2/s^2$ | Turbulent kinetic energy (half the sum of the data in *'tke_vec'*) |
| E_coh | $m^2/s^2$ | Coherent TKE (root-sum-square of Reynold's stresses) |
| upup_ | $m^2/s^2$ | `dat['tke_vec'].sel(tke="upup_")` |
| vpvp_ | $m^2/s^2$ | `dat['tke_vec'].sel(tke="vpvp_")` |
| wpwp_ | $m^2/s^2$ | `dat['tke_vec'].sel(tke="wpwp_")` |
| upvp_ | $m^2/s^2$ | `dat['stress_vec'].sel(tau="upvp_")` |
| upwp_ | $m^2/s^2$ | `dat['stress_vec'].sel(tau="upwp_")` |
| vpwp_ | $m^2/s^2$ | `dat['stress_vec'].sel(tau="vpwp_")` |

**Important Note:** The items listed in Table 4 are not stored in the dataset but are provided as attributes (shortcuts) to the dataset itself. They are accessed through the xarray accessor *velds*.

For example, to return the magnitude of the horizontal velocity:

```
>> import dolfyn
>> dat = dolfyn.read_example('AWAC_test01.wpr')

>> dat.velds.U_mag

<xarray.DataArray 'vel' (range: 20, time: 9997)>
array([[1.12594587, 0.82454599, 0.96503734, ..., 3.40359042, 3.34527587,
        3.44412805],
       [0.86688534, 1.05108722, 1.12899632, ..., 0.72053462, 6.47548786,
        0.49120468],
       [0.88066635, 0.97954744, 0.63123135, ..., 4.37153751, 2.77540426,
        1.81550287],
       ...,
       [1.00013206, 1.21381814, 1.14834231, ..., 5.89236205, 1.44082763,
        2.7157082 ],
       [0.7759962 , 0.89600228, 1.02900833, ..., 2.39949021, 2.18758737,
        4.41797285],
       [0.95729835, 1.15594339, 1.15038508, ..., 3.11517746, 3.79158362,
        2.66788512]])
Coordinates:
  * range    (range) float32 1.41 2.41 3.4 4.4 5.4 ... 17.36 18.36 19.35 20.35
  * time     (time) float64 1.34e+09 1.34e+09 1.34e+09 ... 1.34e+09 1.34e+09
Attributes:
        units:        m/s
        description:  horizontal velocity magnitude
```

### 1.7.7 Time Conversion

Time is handled primary in epoch time, or seconds since 1/1/1970, and includes conversion to Unix timestamps, datetime objects, and MATLAB datenum.

| | |
|---|---|
| *dt642epoch*(dt64) | Convert numpy.datetime64 array to epoch time (seconds since 1/1/1970 00:00:00) |
| *epoch2dt64*(ep_time) | Convert from epoch time (seconds since 1/1/1970 00:00:00) to numpy.datetime64 array |
| *dt642date*(dt64) | Convert numpy.datetime64 array to list of datetime objects |
| *date2dt64*(dt) | Convert numpy.datetime64 array to list of datetime objects |
| *epoch2date*(ep_time[, offset_hr, to_str]) | Convert from epoch time (seconds since 1/1/1970 00:00:00) to a list of datetime objects |
| *date2epoch*(dt) | Convert list of datetime objects to epoch time |
| *date2str*(dt[, format_str]) | Convert list of datetime objects to legible strings |
| *date2matlab*(dt) | Convert list of datetime objects to MATLAB datenum |
| *matlab2date*(matlab_dn) | Convert MATLAB datenum to list of datetime objects |

dolfyn.time.**epoch2dt64**(*ep_time*)

Convert from epoch time (seconds since 1/1/1970 00:00:00) to numpy.datetime64 array

**Parameters**
**ep_time** (*xarray.DataArray*) – Time coordinate data-array or single time element

**Returns**
**time** (*numpy.datetime64*) – The converted datetime64 array

dolfyn.time.**dt642epoch**(*dt64*)

Convert numpy.datetime64 array to epoch time (seconds since 1/1/1970 00:00:00)

**Parameters**
**dt64** (*numpy.datetime64*) – Single or array of datetime64 object(s)

**Returns**
**time** (*float*) – Epoch time (seconds since 1/1/1970 00:00:00)

dolfyn.time.**date2dt64**(*dt*)

Convert numpy.datetime64 array to list of datetime objects

**Parameters**
**time** (*datetime.datetime*) – The converted datetime object

**Returns**
**dt64** (*numpy.datetime64*) – Single or array of datetime64 object(s)

dolfyn.time.**dt642date**(*dt64*)

Convert numpy.datetime64 array to list of datetime objects

**Parameters**
**dt64** (*numpy.datetime64*) – Single or array of datetime64 object(s)

**Returns**
**time** (*datetime.datetime*) – The converted datetime object

dolfyn.time.**epoch2date**(*ep_time*, *offset_hr=0*, *to_str=False*)

Convert from epoch time (seconds since 1/1/1970 00:00:00) to a list of datetime objects

**Parameters**

- **ep_time** (*xarray.DataArray*) – Time coordinate data-array or single time element
- **offset_hr** (*int*) – Number of hours to offset time by (e.g. UTC -7 hours = PDT)
- **to_str** (*logical*) – Converts datetime object to a readable string

**Returns**

**time** (*datetime.datetime*) – The converted datetime object or list(strings)

### Notes

The specific time instance is set during deployment, usually sync'd to the deployment computer. The time seen by DOLfYN is in the timezone of the deployment computer, which is unknown to DOLfYN.

dolfyn.time.**date2str**(*dt*, *format_str=None*)

Convert list of datetime objects to legible strings

**Parameters**

- **dt** (*datetime.datetime*) – Single or list of datetime object(s)
- **format_str** (*string*) – Timestamp string formatting, default: '%Y-%m-%d %H:%M:%S.%f'. See datetime.strftime documentation for timestamp string formatting

**Returns**

**time** (*string*) – Converted timestamps

dolfyn.time.**date2epoch**(*dt*)

Convert list of datetime objects to epoch time

**Parameters**

**dt** (*datetime.datetime*) – Single or list of datetime object(s)

**Returns**

**time** (*float*) – Datetime converted to epoch time (seconds since 1/1/1970 00:00:00)

dolfyn.time.**date2matlab**(*dt*)

Convert list of datetime objects to MATLAB datenum

**Parameters**

**dt** (*datetime.datetime*) – List of datetime objects

**Returns**

**time** (*float*) – List of timestamps in MATLAB datnum format

dolfyn.time.**matlab2date**(*matlab_dn*)

Convert MATLAB datenum to list of datetime objects

**Parameters**

**matlab_dn** (*float*) – List of timestamps in MATLAB datnum format

**Returns**

**dt** (*datetime.datetime*) – List of datetime objects

### 1.7.8 Tools

Spectral analysis and miscellaneous DOLfYN functions are stored here. These functions are used throughout DOLfYN's core code and may also be helpful to users in general.

FFT-based Functions:

| | |
|---|---|
| *psd_freq* | Compute the frequency for vector for a *nfft* and *fs*. |
| *stepsize* | Calculates the fft-step size for a length *l* array. |
| *coherence* | Computes the magnitude-squared coherence of *a* and *b*. |
| *phase_angle* | Compute the phase difference between signals *a* and *b*. |
| *psd* | Compute the power spectral density (PSD). |
| *cpsd* | Compute the cross power spectral density (CPSD) of the signals *a* and *b*. |
| *cpsd_quasisync* | Compute the cross power spectral density (CPSD) of the signals *a* and *b*. |

Other Functions:

| | |
|---|---|
| *detrend* | Remove a linear trend from arr. |
| *group* | Find continuous segments in a boolean array. |
| *slice1d_along_axis* | Return an iterator object for looping over 1-D slices, along `axis`, of an array of shape arr_shape. |
| *fillgaps* | Linearly fill NaN value in an array. |
| *interpgaps* | Fill gaps (NaN values) in `a` by linear interpolation along dimension `dim` with the point spacing specified in `t`. |
| *medfiltnan* | Do a running median filter of the data. |
| *convert_degrees* | Converts between the 'cartesian angle' (counter-clockwise from East) and the 'polar angle' in (degrees clockwise from North) |

dolfyn.tools.psd.**psd_freq**(*nfft*, *fs*, *full=False*)

Compute the frequency for vector for a *nfft* and *fs*.

**Parameters**

- **fs** (*float*) – The sampling frequency (e.g. samples/sec)
- **nfft** (*int*) – The number of samples in a window.
- **full** (*bool (default: False)*) – Whether to return half frequencies (positive), or the full frequencies.

**Returns**

**freq** (|*np.ndarray*|) – The frequency vector, in same units as 'fs'

dolfyn.tools.psd.**stepsize**(*l*, *nfft*, *nens=None*, *step=None*)

Calculates the fft-step size for a length *l* array.

If nens is None, the step size is chosen to maximize data use, minimize nens and have a minimum of 50% overlap.

If nens is specified, the step-size is computed directly.

**Parameters**

- **l** (*The length of the array.*) –
- **nfft** (*The number of points in the fft.*) –

- **nens** (*The number of nens to perform (default compute this).*) –

**Returns**

- **step** (*The step size.*)
- **nens** (*The number of ensemble ffts to average together.*)
- **nfft** (*The number of points in the fft (set to l if nfft>l).*)

dolfyn.tools.psd.**coherence**(*a*, *b*, *nfft*, *window='hann'*, *debias=True*, *noise=(0, 0)*)

Computes the magnitude-squared coherence of *a* and *b*.

**Parameters**

- **a** (numpy.ndarray) – The first array over which to compute coherence.
- **b** (numpy.ndarray) – The second array over which to compute coherence.
- **nfft** (*int*) – The number of points to use in the fft.
- **window** (*string, np.ndarray (default 'hann')*) – The window to use for ffts.
- **debias** (*bool (default: True)*) – Specify whether to debias the signal according to Benignus1969.
- **noise** (*tuple(2), or float*) – The *noise* keyword may be used to specify the signals' noise levels (std of noise in a,b). If *noise* is a two element tuple or list, the first and second elements specify the noise levels of a and b, respectively. default: noise=(0,0)

**Returns**

**out** (|*np.ndarray*|) – Coherence between *a* and *b*

### Notes

Coherence is defined as:

$$C_{ab} = \frac{|S_{ab}|^2}{S_{aa} * S_{bb}}$$

Here $S_{ab}$, $S_{aa}$ and $S_{bb}$ are the cross, and auto spectral densities of the signal *a* and *b*.

dolfyn.tools.psd.**cpsd_quasisync**(*a*, *b*, *nfft*, *fs*, *window='hann'*)

Compute the cross power spectral density (CPSD) of the signals *a* and *b*.

**Parameters**

- **a** (numpy.ndarray) – The first signal.
- **b** (numpy.ndarray) – The second signal.
- **nfft** (*int*) – The number of points in the fft.
- **fs** (*float*) – The sample rate (e.g. sample/second).
- **window** ({None, 1, 'hann', numpy.ndarray}) – The window to use (default: 'hann'). Valid entries are: - None,1 : uses a 'boxcar' or ones window. - 'hann' : hanning window. - a length(nfft) array : use this as the window directly.

**Returns**

**cpsd** (|*np.ndarray*|) – The cross-spectral density of *a* and *b*.

**See also:**

*psd()*, *coherence()*, *cpsd()*, numpy.fft

### Notes

*a* and *b* do not need to be 'tightly' synchronized, and can even be different lengths, but the first- and last-index of both series should be synchronized (to whatever degree you want unbiased phases).

This performs:

$$fft(a) * conj(fft(b))$$

Note that this is consistent with `numpy.correlate()`.

It detrends the data and uses a minimum of 50% overlap for the shorter of *a* and *b*. For the longer, the overlap depends on the difference in size. 1-(l_short/l_long) data will be underutilized (where l_short and l_long are the length of the shorter and longer series, respectively).

The units of the spectra is the product of the units of *a* and *b*, divided by the units of fs.

`dolfyn.tools.psd.`**cpsd**(*a*, *b*, *nfft*, *fs*, *window='hann'*, *step=None*)

Compute the cross power spectral density (CPSD) of the signals *a* and *b*.

**Parameters**

- **a** (`numpy.ndarray`) – The first signal.
- **b** (`numpy.ndarray`) – The second signal.
- **nfft** (*int*) – The number of points in the fft.
- **fs** (*float*) – The sample rate (e.g. sample/second).
- **window** ({None, 1, 'hann', `numpy.ndarray`}) – The window to use (default: 'hann'). Valid entries are: - None,1 : uses a 'boxcar' or ones window. - 'hann' : hanning window. - a length(nfft) array : use this as the window directly.
- **step** (*int*) – Use this to specify the overlap. For example: - step : nfft/2 specifies a 50% overlap. - step : nfft specifies no overlap. - step=2*nfft means that half the data will be skipped. By default, *step* is calculated to maximize data use, have at least 50% overlap and minimize the number of ensembles.

**Returns**

**cpsd** (|*np.ndarray*|) – The cross-spectral density of *a* and *b*.

**See also:**

*psd()*, *coherence()*, None

### Notes

cpsd removes a linear trend from the signals.

The two signals should be the same length, and should both be real.

This performs:

$$fft(a) * conj(fft(b))$$

This implementation is consistent with the numpy.correlate definition of correlation. (The conjugate of D.B. Chelton's definition of correlation.)

The units of the spectra is the product of the units of *a* and *b*, divided by the units of fs.

dolfyn.tools.psd.**psd**(*a*, *nfft*, *fs*, *window='hann'*, *step=None*)

Compute the power spectral density (PSD).

This function computes the one-dimensional *n*-point PSD.

The units of the spectra is the product of the units of *a* and *b*, divided by the units of fs.

**Parameters**

- **a** (numpy.ndarray) – The first signal, as a 1D vector
- **nfft** (*int*) – The number of points in the fft.
- **fs** (*float*) – The sample rate (e.g. sample/second).
- **window** ({None, 1, 'hann', numpy.ndarray}) – The window to use (default: 'hann'). Valid entries are: - None,1 : uses a 'boxcar' or ones window. - 'hann' : hanning window. - a length(nfft) array : use this as the window directly.
- **step** (*int*) – Use this to specify the overlap. For example: - step : nfft/2 specifies a 50% overlap. - step : nfft specifies no overlap. - step=2*nfft means that half the data will be skipped. By default, *step* is calculated to maximize data use, have at least 50% overlap and minimize the number of ensembles.

**Returns**

**psd** (|*np.ndarray*|) – The power spectral density of *a* and *b*.

### Notes

Credit: This function's line of code was copied from JN's fast_psd.m routine.

**See also:**

*cpsd()*, *coherence()*, None

dolfyn.tools.psd.**phase_angle**(*a*, *b*, *nfft*, *window='hann'*, *step=None*)

Compute the phase difference between signals *a* and *b*. This is the complimentary function to coherence and cpsd.

Positive angles means that *b* leads *a*, i.e. this does, essentially:

angle(b) - angle(a)

This function computes one-dimensional *n*-point PSD.

The angles are output as magnitude = 1 complex numbers (to simplify averaging). Therefore, use *numpy.angle* to actually output the angle.

**Parameters**

- **a** (*1d-array_like, the signal. Currently only supports vectors.*) –
- **nfft** (*The number of points in the fft.*) –
- **window** (*The window to use (default: 'hann'). Valid entries are:*) – None,1 : uses a 'boxcar' or ones window. 'hann' : hanning window. a length(nfft) array : use this as the window directly.
- **step** (*Use this to specify the overlap. For example:*) –
  - step=nfft/2 specifies a 50% overlap.
  - step=nfft specifies no overlap.
  - step=2*nfft means that half the data will be skipped.

By default, *step* is calculated to maximize data use, have at least 50% overlap and minimize the number of ensembles.

**Returns**

**ang** (*complex |np.ndarray| (unit magnitude values)*)

**See also:**

None, *coherence()*, *cpsd()*

dolfyn.tools.misc.**detrend**(*arr*, *axis=-1*, *in_place=False*)

Remove a linear trend from arr.

**Parameters**

- **arr** (*array_like*) – The array from which to remove a linear trend.
- **axis** (*int*) – The axis along which to operate.

### Notes

This method is copied from the matplotlib.mlab library, but implements the covariance calcs explicitly for added speed.

This works much faster than mpl.mlab.detrend for multi-dimensional arrays, and is also faster than linalg.lstsq methods.

dolfyn.tools.misc.**group**(*bl*, *min_length=0*)

Find continuous segments in a boolean array.

**Parameters**

- **bl** (numpy.ndarray (dtype='bool')) – The input boolean array.
- **min_length** (*int (optional)*) – Specifies the minimum number of continuous points to consider a *group* (i.e. that will be returned).

**Returns**

**out** (*np.ndarray(slices,)*) – a vector of slice objects, which indicate the continuous sections where *bl* is True.

### Notes

This function has funny behavior for single points. It will return the same two indices for the beginning and end.

dolfyn.tools.misc.**slice1d_along_axis**(*arr_shape*, *axis=0*)

Return an iterator object for looping over 1-D slices, along axis, of an array of shape arr_shape.

**Parameters**

- **arr_shape** (*tuple,list*) – Shape of the array over which the slices will be made.
- **axis** (*integer*) – Axis along which *arr* is sliced.

**Returns**

**Iterator** (*object*) – The iterator object returns slice objects which slices arrays of shape arr_shape into 1-D arrays.

### Examples

>> out=np.empty(replace(arr.shape,0,1)) >> for slc in slice1d_along_axis(arr.shape,axis=0): >> out[slc]=my_1d_function(arr[slc])

dolfyn.tools.misc.**fillgaps**(*a*, *maxgap=inf*, *dim=0*, *extrapFlg=False*)

Linearly fill NaN value in an array.

**Parameters**

- **a** (numpy.ndarray) – The array to be filled.
- **maxgap** (numpy.ndarray (optional: inf)) – The maximum gap to fill.
- **dim** (*int (optional: 0)*) – The dimension to operate along.
- **extrapFlg** (*bool (optional: False)*) – Whether to extrapolate if NaNs are found at the ends of the array.

**See also:**

*dolfyn.tools.misc.interpgaps*
Linearly interpolates in time.

### Notes

This function interpolates assuming spacing/timestep between successive points is constant. If the spacing is not constant, use interpgaps.

dolfyn.tools.misc.**interpgaps**(*a*, *t*, *maxgap=inf*, *dim=0*, *extrapFlg=False*)

Fill gaps (NaN values) in a by linear interpolation along dimension dim with the point spacing specified in t.

**Parameters**

- **a** (numpy.ndarray) – The array containing NaN values to be filled.
- **t** (numpy.ndarray (len(t) == a.shape[dim])) – Independent variable of the points in a, e.g. timestep
- **maxgap** (numpy.ndarray (optional: inf)) – The maximum gap to fill.
- **dim** (*int (optional: 0)*) – The dimension to operate along.
- **extrapFlg** (*bool (optional: False)*) – Whether to extrapolate if NaNs are found at the ends of the array.

**See also:**

*dolfyn.tools.misc.fillgaps*
Linearly interpolates in array-index space.

dolfyn.tools.misc.**medfiltnan**(*a*, *kernel*, *thresh=0*)

Do a running median filter of the data. Regions where more than thresh fraction of the points are NaN are set to NaN.

**Parameters**

- **a** (numpy.ndarray) – 2D array containing data to be filtered.
- **kernel_size** (numpy.ndarray or list, optional) – A scalar or a list of length 2, giving the size of the median filter window in each dimension. Elements of kernel_size should be odd. If kernel_size is a scalar, then this scalar is used as the size in each dimension.

- **thresh** (*int*) – Maximum gap in *a* to filter over

**Returns**

**out** (|*np.ndarray*|) – 2D array of same size containing filtered data

**See also:**

`scipy.signal.medfilt2d`

`dolfyn.tools.misc.`**convert_degrees**(*deg*, *tidal_mode=True*)

Converts between the 'cartesian angle' (counter-clockwise from East) and the 'polar angle' in (degrees clockwise from North)

**Parameters**

- **deg** (*float or array-like*) – Number or array in 'degrees CCW from East' or 'degrees CW from North'
- **tidal_mode** (*bool (default: True)*) – If true, range is set from 0 to +/-180 degrees. If false, range is 0 to 360 degrees

**Returns**

**out** (*float or array-like*) – Input data transformed to 'degrees CW from North' or 'degrees CCW from East', respectively (based on *deg*)

### Notes

The same algorithm is used to convert back and forth between 'CCW from E' and 'CW from N'

CHAPTER

TWO

# EXAMPLES

## 2.1 ADCP Example

The following example shows a typical workflow for analyzing ADCP data using DOLfYN's tools.

A typical ADCP data workflow is broken down into 1. Review the raw data - Check timestamps - Calculate/check that the depth bin locations are correct - Look at velocity, beam amplitude and/or beam correlation data quality 2. Remove data located above the water surface or below the seafloor 3. Check for spurious datapoints and remove if necessary 4. If not already done within the instrument, average the data into bins of a set time length (normally 5 to 10 min) 5. Conduct further analysis as required

Start by importing the necessary DOLfYN tools through MHKiT:

[1]:
```
# Import core DOLfYN functions
import dolfyn
# Import ADCP-specific API tools
from dolfyn.adp import api
```

### 2.1.1 Read Raw Instrument Data

The core benefit of DOLfYN is that it can read in raw data directly after transferring it off of the ADCP. The ADCP used here is a Nortek Signature 1000, with the file extension '.ad2cp'. This specific dataset contains several hours worth of velocity data collected at 1 Hz from the ADCP mounted on a bottom lander in a tidal inlet. The instruments that DOLfYN supports are listed in the docs.

Start by reading in the raw datafile downloaded from the instrument. The `read` function reads the raw file and dumps the information into an xarray Dataset, which contains a few groups of variables:

1. Velocity in the instrument-saved coordinate system (beam, XYZ, ENU)
2. Beam amplitude and correlation data
3. Measurements of the instrument's bearing and environment
4. Orientation matrices DOLfYN uses for rotating through coordinate frames.

[2]:
```
ds = dolfyn.read('../dolfyn/example_data/Sig1000_tidal.ad2cp')
```

```
Reading file ../dolfyn/example_data/Sig1000_tidal.ad2cp ...
```

There are two ways to see what's in a DOLfYN Dataset. The first is to simply type the dataset's name to see the standard xarray output. To access a particular variable in a dataset, use dict-style (`ds['vel']`) or attribute-style syntax (`ds.vel`). See the xarray docs for more details on how to use the xarray format.

[3]:
```
# print the dataset
ds
```

[3]:
```
<xarray.Dataset>
Dimensions:               (time: 55000, dirIMU: 3, dir: 4, range: 28, beam: 4,
                           earth: 3, inst: 3, q: 4, time_b5: 55000,
                           range_b5: 28, x: 4, x*: 4)
Coordinates:
  * time                  (time) datetime64[ns] 2020-08-15T00:20:00.500999927 ...
  * dirIMU                (dirIMU) <U1 'E' 'N' 'U'
  * dir                   (dir) <U2 'E' 'N' 'U1' 'U2'
  * range                 (range) float64 0.6 1.1 1.6 2.1 ... 12.6 13.1 13.6 14.1
  * beam                  (beam) int32 1 2 3 4
  * earth                 (earth) <U1 'E' 'N' 'U'
  * inst                  (inst) <U1 'X' 'Y' 'Z'
  * q                     (q) <U1 'w' 'x' 'y' 'z'
  * time_b5               (time_b5) datetime64[ns] 2020-08-15T00:20:00.4384999...
  * range_b5              (range_b5) float64 0.6 1.1 1.6 2.1 ... 13.1 13.6 14.1
  * x                     (x) int32 1 2 3 4
  * x*                    (x*) int32 1 2 3 4
Data variables: (12/38)
    c_sound               (time) float32 1.502e+03 1.502e+03 ... 1.498e+03
    temp                  (time) float32 14.55 14.55 14.55 ... 13.47 13.47 13.47
    pressure              (time) float32 9.713 9.718 9.718 ... 9.596 9.594 9.596
    mag                   (dirIMU, time) float32 72.5 72.7 72.6 ... -197.2 -195.7
    accel                 (dirIMU, time) float32 -0.00479 -0.01437 ... 9.729
    batt                  (time) float32 16.6 16.6 16.6 16.6 ... 16.4 16.4 15.2
    ...                    ...
    telemetry_data        (time) uint8 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0
    boost_running         (time) uint8 0 0 0 0 0 0 0 0 1 0 ... 0 1 0 0 0 0 0 0 1
    heading               (time) float32 -12.52 -12.51 -12.51 ... -12.52 -12.5
    pitch                 (time) float32 -0.065 -0.06 -0.06 ... -0.06 -0.05 -0.05
    roll                  (time) float32 -7.425 -7.42 -7.42 ... -6.45 -6.45 -6.45
    beam2inst_orientmat   (x, x*) float32 1.183 0.0 -1.183 ... 0.5518 0.0 0.5518
Attributes: (12/33)
    filehead_config:       {'CLOCKSTR': {'TIME': '"2020-08-13 13:56:21"'}, 'I...
    inst_model:            Signature1000
    inst_make:             Nortek
    inst_type:             ADCP
    rotate_vars:           ['vel', 'accel', 'accel_b5', 'angrt', 'angrt_b5', ...
    burst_config:          {'press_valid': True, 'temp_valid': True, 'compass...
    ...                    ...
    proc_idle_less_3pct:   0
    proc_idle_less_6pct:   0
    proc_idle_less_12pct:  0
    coord_sys:             earth
    has_imu:               1
    fs:                    1
```

A second way provided to look at data is through the *DOLfYN view*. This view has several convenience methods, shortcuts, and functions built-in. It includes an alternate – and somewhat more informative/compact – description of the data object when in interactive mode. This can be accessed using

[4]:
```
ds_dolfyn = ds.velds
ds_dolfyn
```

[4]:
```
<ADCP data object>:  Nortek Signature1000
  . 15.28 hours (started: Aug 15, 2020 00:20)
  . earth-frame
  . (55000 pings @ 1Hz)
  Variables:
  - time ('time',)
  - time_b5 ('time_b5',)
  - vel ('dir', 'range', 'time')
  - vel_b5 ('range_b5', 'time_b5')
  - range ('range',)
  - orientmat ('earth', 'inst', 'time')
  - heading ('time',)
  - pitch ('time',)
  - roll ('time',)
  - temp ('time',)
  - pressure ('time',)
  - amp ('beam', 'range', 'time')
  - amp_b5 ('range_b5', 'time_b5')
  - corr ('beam', 'range', 'time')
  - corr_b5 ('range_b5', 'time_b5')
  - accel ('dirIMU', 'time')
  - angrt ('dirIMU', 'time')
  - mag ('dirIMU', 'time')
  ... and others (see `<obj>.variables`)
```

### 2.1.2 First Steps and QC'ing Data

#### 1.) Set deployment height

Because this is a Nortek instrument, the deployment software doesn't take into account the deployment height, aka where in the water column the ADCP is. The center of the first depth bin is located at a distance = deployment height + blanking distance + cell size, so the `range` coordinate needs to be corrected so that '0' corresponds to the seafloor. This can be done in DOLfYN using the `set_range_offset` function. This same function can be used to account for the depth of a down-facing instrument below the water surface.

Note, if using a Teledyne RDI ADCP, TRDI's deployment software asks the user to enter the deployment height/depth during configuration. If needed, this can be adjusted after-the-fact using `set_range_offset` as well.

[5]:
```
# The ADCP transducers were measured to be 0.6 m from the feet of the lander
api.clean.set_range_offset(ds, 0.6)
```

So, the center of bin 1 is located at 1.2 m:

[6]:
```
ds.range
```

[6]:
```
<xarray.DataArray 'range' (range: 28)>
array([ 1.2,  1.7,  2.2,  2.7,  3.2,  3.7,  4.2,  4.7,  5.2,  5.7,  6.2,  6.7,
        7.2,  7.7,  8.2,  8.7,  9.2,  9.7, 10.2, 10.7, 11.2, 11.7, 12.2, 12.7,
       13.2, 13.7, 14.2, 14.7])
Coordinates:
```

(continues on next page)

(continued from previous page)

```
  * range    (range) float64 1.2 1.7 2.2 2.7 3.2 ... 12.7 13.2 13.7 14.2 14.7
Attributes:
    units:    m
```

### 2.) Remove data beyond surface level

To reduce the amount of data the code must run through, we can remove all data at and above the water surface. Because the instrument was looking up, we can use the pressure sensor data and the function `find_surface_from_P`. This does require that the pressure sensor was 'zeroed' prior to deployment. If the instrument is looking down or lacks pressure data, use the function `find_surface` to detect the seabed or water surface.

ADCPs don't measure water salinity, so it will need to be given to the function. The returned dataset contains the an additional variable "depth". If `find_surface_from_P` is run after `set_range_offset`, depth is the distance of the water surface away from the seafloor; otherwise it is the distance to the ADCP pressure sensor.

After calculating depth, data in depth bins at and above the physical water surface can be removed using `nan_beyond_surface`. Note that this function returns a new dataset.

[7]:
```
api.clean.find_surface_from_P(ds, salinity=31)
ds = api.clean.nan_beyond_surface(ds)
```

### 3.) Correlation filter

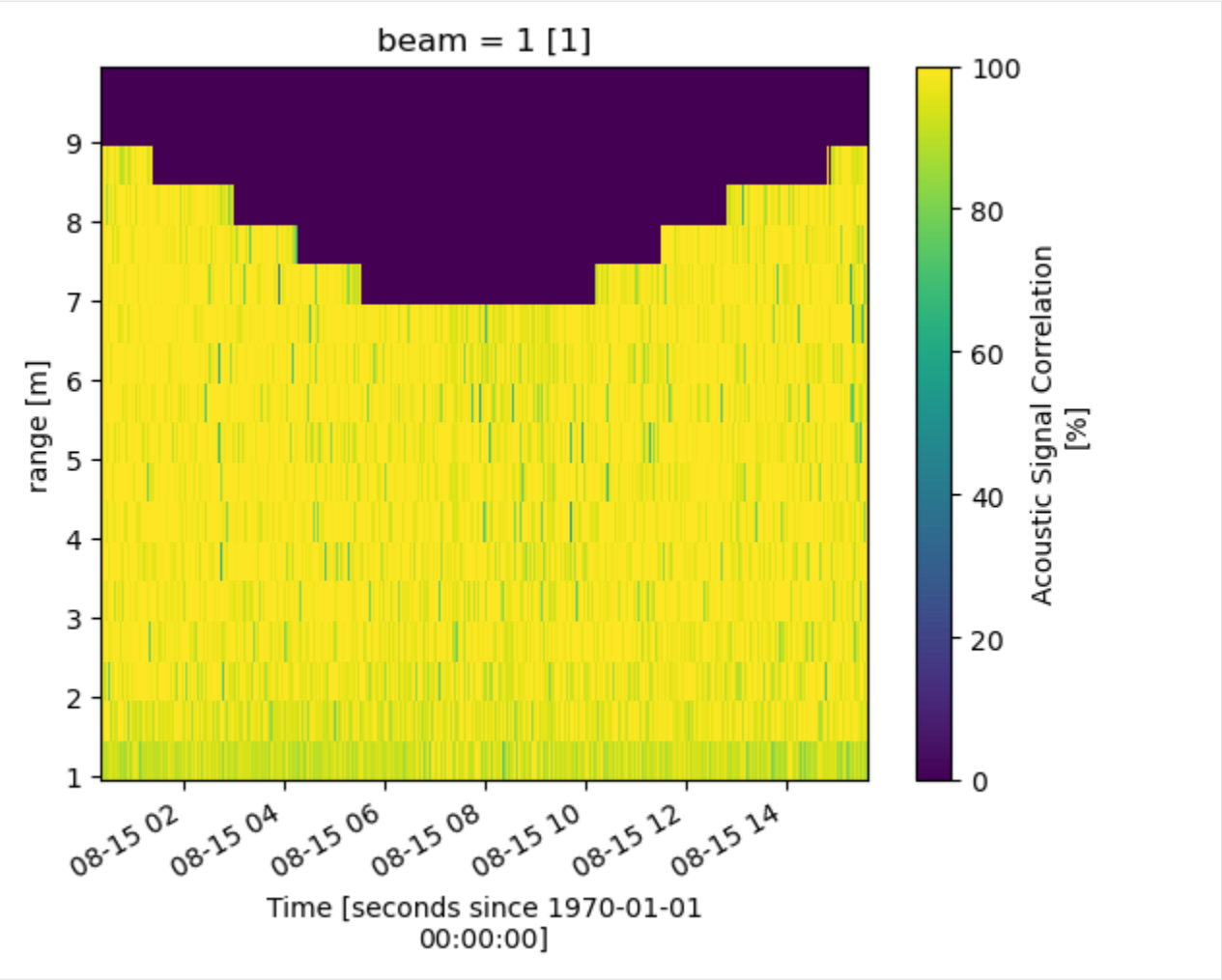
Once beyond-surface bins have been removed, ADCP data is typically filtered by acoustic signal correlation to clear out spurious velocity datapoints (caused by bubbles, kelp, fish, etc moving through one or multiple beams).

We can take a quick look at the data to see about where this value should be using xarray's built-in plotting. In the following line of code, we use xarray's slicing capabilities to show data from beam 1 between a range of 0 to 10 m from the ADCP.

Not all ADCPs return acoustic signal correlation, which in essence is a quantitative measure of signal quality. ADCPs with older hardware do not provide a correlation measurement, so this step will be skipped with these instruments.

[8]:
```
%matplotlib inline
ds['corr'].sel(beam=1, range=slice(0,10)).plot()
```

[8]:
```
<matplotlib.collections.QuadMesh at 0x22bc6980cd0>
```




It's a good idea to check the other beams as well. Much of this data is high quality, and to not lose data will low correlation caused by natural variation, we'll use the `correlation_filter` to set velocity values corresponding to correlations below 50% to NaN.

Note that this threshold is dependent on the deployment environment and instrument, and it isn't uncommon to use a value as low as 30%, or to pass on this function completely.

[9]:
```
ds = api.clean.correlation_filter(ds, thresh=50)
```

### 2.1.3 Review the Data

Now that the data has been cleaned, the next step is to rotate the velocity data into true East, North, Up coordinates.

ADCPs use an internal compass or magnetometer to determine magnetic ENU directions. The `set_declination` function takes the user supplied magnetic declination (which can be looked up online for specific coordinates) and adjusts the velocity data accordingly.

Instruments save vector data in the coordinate system specified in the deployment configuration file. To make the data useful, it must be rotated through coordinate systems ("beam"<->"inst"<->"earth"<->"principal"), done through the `rotate2` function. If the "earth" (ENU) coordinate system is specified, DOLfYN will automatically rotate the dataset through the necessary coordinate systems to get there. The `inplace` set as true will alter the input dataset "in place", a.k.a. it not create a new dataset.

Because this ADCP data was already in the "earth" coordinate system, `rotate2` will return the input dataset. `set_declination` will run correctly no matter the coordinate system.

[10]:
```
dolfyn.set_declination(ds, declin=15.8, inplace=True) # 15.8 deg East
dolfyn.rotate2(ds, 'earth', inplace=True)
```

```
Data is already in the earth coordinate system
```

To rotate into the principal frame of reference (streamwise, cross-stream, vertical), if desired, we must first calculate the depth-averaged principal flow heading and add it to the dataset attributes. Then the dataset can be rotated using the same `rotate2` function. We use `inplace=False` because we do not want to alter the input dataset.

[11]:
```
ds.attrs['principal_heading'] = dolfyn.calc_principal_heading(ds['vel'].mean('range'))
ds_streamwise = dolfyn.rotate2(ds, 'principal', inplace=False)
```

Because this deployment was set up in "burst mode", the next standard step in this analysis is to average the velocity data into time bins.

If an instrument was set up to record velocity data in an "averaging mode" (a specific profile and/or average interval, e.g. take 5 minutes of data every 30 minutes), this step was completed within the ADCP during deployment and can be skipped.

To average the data into time bins (aka ensembles), start by initiating the binning tool `VelBinner`. "n_bin" is the number of data points in each ensemble, in this case 300 seconds worth of data, and "fs" is the sampling frequency, which is 1 Hz for this deployment. Once initiated, average the data into ensembles using the binning tool's `do_avg` function.

[12]:
```
avg_tool = api.VelBinner(n_bin=ds.fs*300, fs=ds.fs)
ds_avg = avg_tool.do_avg(ds)
```

Two more variables not automatically provided that may be of interest are the horizontal velocity magnitude (speed) and its direction, respectively `U_mag` and `U_dir`. There are included as "shortcut" functions, and are accessed through the keyword `velds`, as shown in the code block below. The full list of "shorcuts" are listed here.

[13]:
```
ds_avg['U_mag'] = ds_avg.velds.U_mag
ds_avg['U_dir'] = ds_avg.velds.U_dir
```

Plotting can be accomplished through the user's preferred package. Matplotlib is shown here for simplicity, and flow speed and direction are plotted below with a blue line delineating the water surface level.

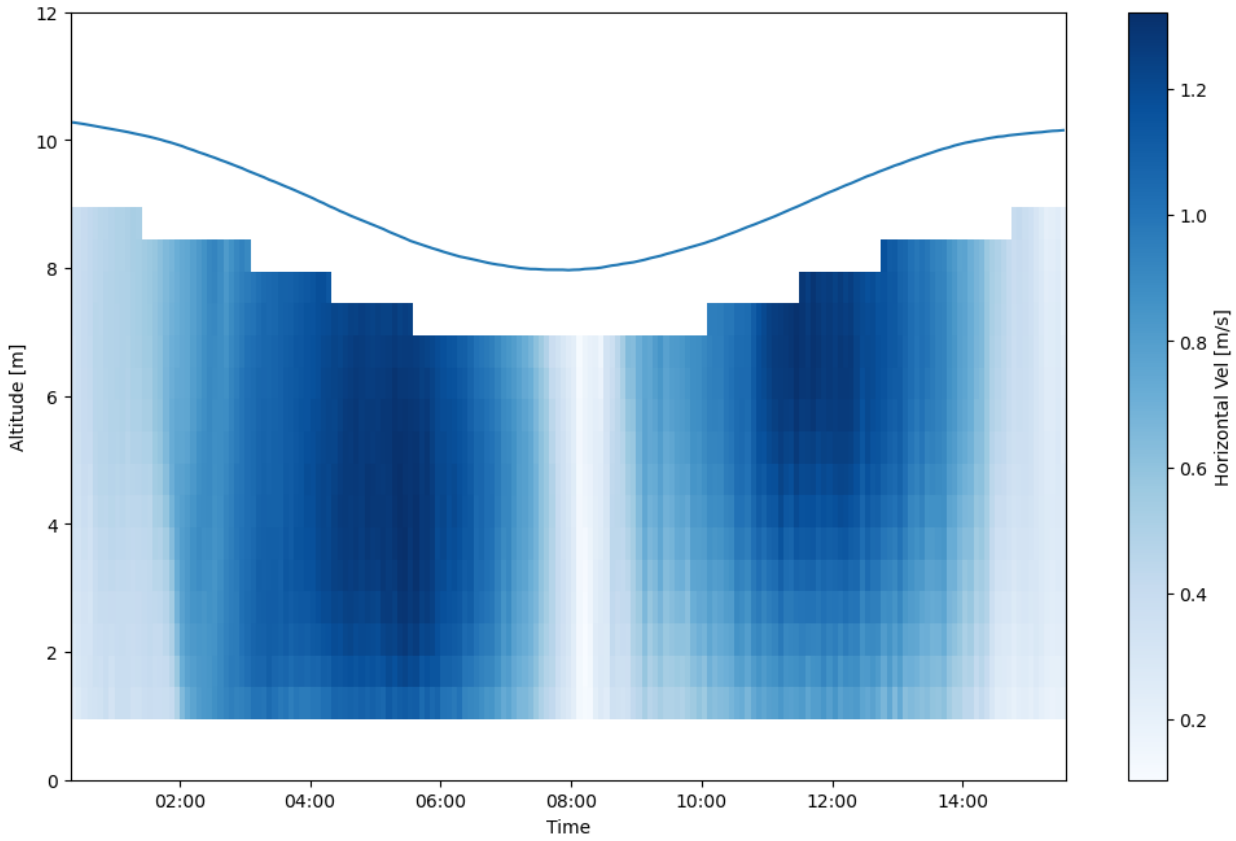
[14]:
```
%matplotlib inline
from matplotlib import pyplot as plt
import matplotlib.dates as dt

ax = plt.figure(figsize=(12,8)).add_axes([.14, .14, .8, .74])
# Plot flow speed
t = dolfyn.time.dt642date(ds_avg.time)
plt.pcolormesh(t, ds_avg['range'], ds_avg['U_mag'], cmap='Blues', shading='nearest')
# Plot the water surface
ax.plot(t, ds_avg['depth'])

# Set up time on x-axis
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(dt.DateFormatter('%H:%M'))

ax.set_ylabel('Altitude [m]')
```

(continues on next page)

(continued from previous page)

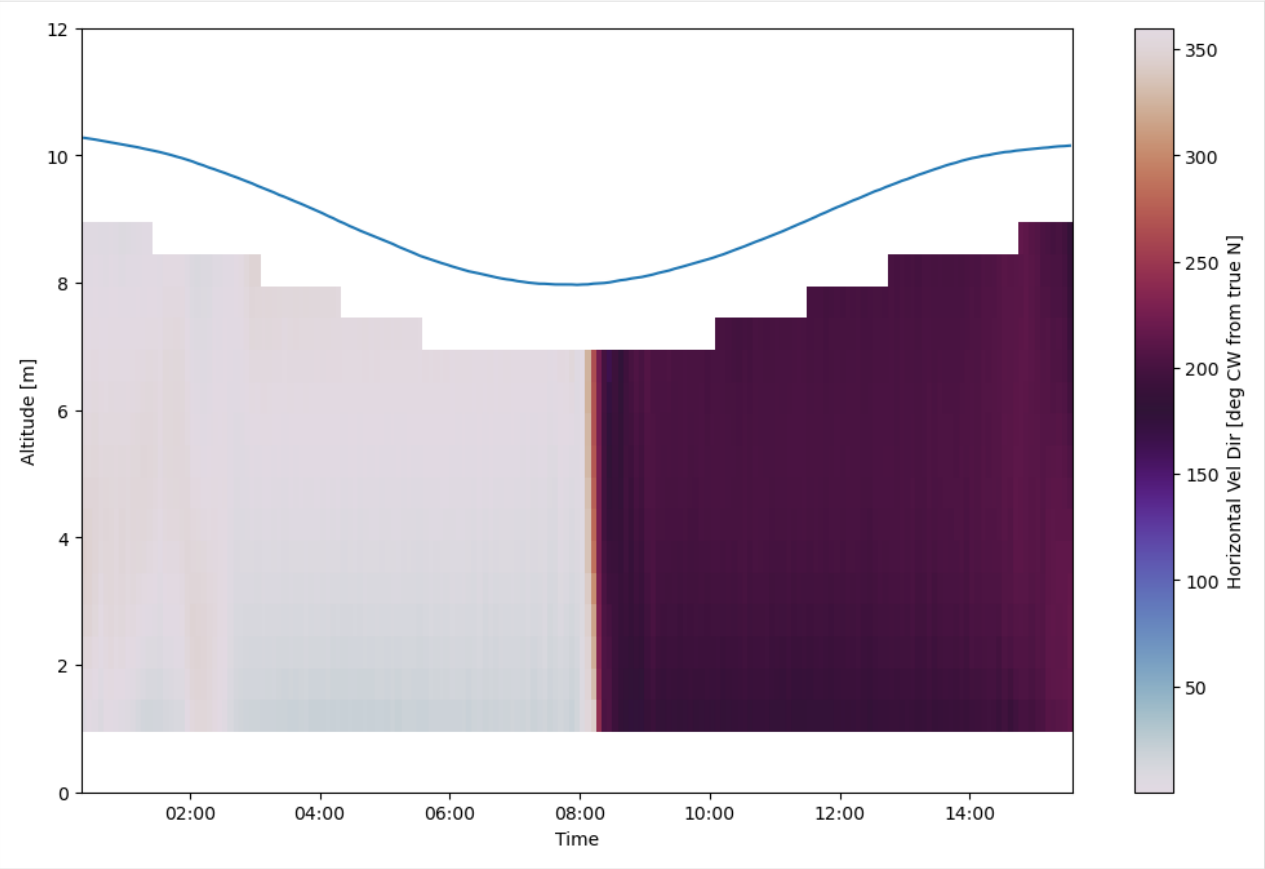
```
ax.set_ylim([0, 12])
plt.colorbar(label='Horizontal Vel [m/s]')
```

[14]:
```
<matplotlib.colorbar.Colorbar at 0x22bc076b3a0>
```




[15]:
```
ax = plt.figure(figsize=(12,8)).add_axes([.14, .14, .8, .74])
# Plot flow direction
plt.pcolormesh(t, ds_avg['range'], ds_avg['U_dir'], cmap='twilight', shading='nearest')
# Plot the water surface
ax.plot(t, ds_avg['depth'])

# set up time on x-axis
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(dt.DateFormatter('%H:%M'))

ax.set_ylabel('Altitude [m]')
ax.set_ylim([0, 12])
plt.colorbar(label='Horizontal Vel Dir [deg CW from true N]')
```

[15]:
```
<matplotlib.colorbar.Colorbar at 0x22bc6204190>
```




### 2.1.4 Saving and Loading DOLfYN datasets

Datasets can be saved and reloaded using the `save` and `load` functions. Xarray is saved natively in netCDF format, hence the ".nc" extension.

Note: DOLfYN datasets cannot be saved using xarray's native `ds.to_netcdf`; however, DOLfYN datasets can be opened using `xarray.open_dataset`.

[16]:
```
# Uncomment these lines to save and load to your current working directory
#dolfyn.save(ds, 'your_data.nc')
#ds_saved = dolfyn.load('your_data.nc')
```

## 2.2 ADV Example

The following example shows a simple workflow for analyzing ADV data using DOLfYN's tools.

A typical ADV data workflow is broken down into 1. Review the raw data - Check timestamps - Look at velocity data quality, particularly for spiking 2. Check for spurious datapoints and remove. Replace bad datapoints using interpolation if desired 3. Rotate the data into principal flow coordinates (streamwise, cross-stream, vertical) 4. Average the data into bins, or ensembles, of a set time length (normally 5 to 10 min) 5. Calculate turbulence statistics (turbulence intensity, TKE, Reynolds stresses) of the measured flowfield

Start by importing the necessary DOLfYN tools:

[1]:
```
# Import core DOLfYN functions
import dolfyn
# Import ADV-specific API tools
from dolfyn.adv import api
```

### 2.2.1 Read Raw Instrument Data

DOLfYN currently only carries support for the Nortek Vector ADV. The example loaded here is a short clip of data from a test deployment to show DOLfN's capabilities.

Start by reading in the raw datafile downloaded from the instrument. The `dolfyn.read` function reads the raw file and dumps the information into an xarray Dataset, which contains three groups of variables:

1. Velocity, amplitude, and correlation of the Doppler velocimetry
2. Measurements of the instrument's bearing and environment
3. Orientation matrices DOLfYN uses for rotating through coordinate frames.

[2]:
```
ds = dolfyn.read('../dolfyn/example_data/vector_data01.VEC')
```

```
Reading file ../dolfyn/example_data/vector_data01.VEC ...
```

There are two ways to see what's in a DOLfYN Dataset. The first is to simply type the dataset's name to see the standard xarray output. To access a particular variable in a dataset, use dict-style (`ds['vel']`) or attribute-style syntax (`ds.vel`). See the xarray docs for more details on how to use the xarray format.

[3]:
```
# print the dataset
ds
```

[3]:
```
<xarray.Dataset>
Dimensions:                (x: 3, x*: 3, time: 122912, dir: 3, beam: 3, earth: 3,
                            inst: 3)
Coordinates:
  * x                      (x) int32 1 2 3
  * x*                     (x*) int32 1 2 3
  * time                   (time) datetime64[ns] 2012-06-12T12:00:02.968749284 ...
  * dir                    (dir) <U1 'X' 'Y' 'Z'
  * beam                   (beam) int32 1 2 3
  * earth                  (earth) <U1 'E' 'N' 'U'
  * inst                   (inst) <U1 'X' 'Y' 'Z'
Data variables: (12/15)
    beam2inst_orientmat    (x, x*) float64 2.709 -1.34 -1.364 ... -0.3438 -0.3499
    batt                   (time) float32 13.2 13.2 13.2 13.2 ... nan nan nan nan
    c_sound                (time) float32 1.493e+03 1.493e+03 ... nan nan
    heading                (time) float32 5.6 10.5 10.51 10.52 ... nan nan nan nan
    pitch                  (time) float32 -31.5 -31.7 -31.69 ... nan nan nan
    roll                   (time) float32 0.4 4.2 4.253 4.306 ... nan nan nan nan
    ...                     ...
    orientation_down       (time) bool True True True True ... True True True True
    vel                    (dir, time) float32 -1.002 -1.008 -0.944 ... nan nan
    amp                    (beam, time) uint8 104 110 111 113 108 ... 0 0 0 0 0
    corr                   (beam, time) uint8 97 91 97 98 90 95 95 ... 0 0 0 0 0 0
    pressure               (time) float64 5.448 5.436 5.484 5.448 ... 0.0 0.0 0.0
    orientmat              (earth, inst, time) float32 0.0832 0.155 ... -0.7065
```

(continues on next page)

(continued from previous page)

```
Attributes: (12/39)
    inst_make:                  Nortek
    inst_model:                 Vector
    inst_type:                  ADV
    rotate_vars:                ['vel']
    n_beams:                    3
    profile_mode:               continuous
    ...                         ...
    recorder_size_bytes:        4074766336
    vel_range:                  normal
    firmware_version:           3.34
    fs:                         32.0
    coord_sys:                  inst
    has_imu:                    0
```

A second way provided to look at data is through the *DOLfYN view*. This view has several convenience methods, shortcuts, and functions built-in. It includes an alternate – and somewhat more informative/compact – description of the data object when in interactive mode. This can be accessed using

[4]:
```
ds_dolfyn = ds.velds
ds_dolfyn
```

[4]:
```
<ADV data object>:  Nortek Vector
  . 1.07 hours (started: Jun 12, 2012 12:00)
  . inst-frame
  . (122912 pings @ 32.0Hz)
  Variables:
  - time ('time',)
  - vel ('dir', 'time')
  - orientmat ('earth', 'inst', 'time')
  - heading ('time',)
  - pitch ('time',)
  - roll ('time',)
  - temp ('time',)
  - pressure ('time',)
  - amp ('beam', 'time')
  - corr ('beam', 'time')
  ... and others (see `<obj>.variables`)
```

### 2.2.2 QC'ing Data

ADV velocity data tends to have spikes due to Doppler noise, and the common way to "despike" the data is by using the phase-space algorithm by Goring and Nikora (2002). DOLfYN integrates this function using a 2-step approach: create a logical mask where True corresponds to a spike detection, and then utilize an interpolation function to replace the spikes.

[5]:
```
# Clean the file using the Goring+Nikora method:
mask = api.clean.GN2002(ds['vel'], npt=5000)
# Replace bad datapoints via cubic spline interpolation
ds['vel'] = api.clean.clean_fill(ds['vel'], mask, npt=12, method='cubic', maxgap=None)
```

(continues on next page)

(continued from previous page)

```
print('Percent of data containing spikes: {0:.2f}%'.format(100*mask.mean()))

# If interpolation isn't desired:
ds_nan = ds.copy(deep=True)
ds_nan.coords['mask'] = (('dir','time'), ~mask)
ds_nan['vel'] = ds_nan['vel'].where(ds_nan.mask)
```

```
Percent of data containing spikes: 0.73%
```

### 2.2.3 Coordinate Rotations

Now that the data has been cleaned, the next step is to rotate the velocity data into true East, North, Up (ENU) coordinates.

ADVs use an internal compass or magnetometer to determine magnetic ENU directions. The `set_declination` function takes the user supplied magnetic declination (which can be looked up online for specific coordinates) and adjusts the orientation matrix saved within the Dataset. (Note: the "heading" variable will not change).

Instruments save vector data in the coordinate system specified in the deployment configuration file. To make the data useful, it must be rotated through coordinate systems ("beam"<->"inst"<->"earth"<->"principal"), done through the `rotate2` function. If the "earth" (ENU) coordinate system is specified, DOLfYN will automatically rotate the dataset through the necessary coordinate systems to get there. The `inplace` set as true will alter the input dataset "in place", a.k.a. it not create a new dataset.

[6]:
```
# First set the magnetic declination
dolfyn.set_declination(ds, declin=10, inplace=True) # declination points 10 degrees East

# Rotate that data from the instrument to earth frame (ENU):
dolfyn.rotate2(ds, 'earth', inplace=True)
```

Once in the true ENU frame of reference, we can calculate the principal flow direction for the velocity data and rotate it into the principal frame of reference (streamwise, cross-stream, vertical). Principal flow directions are aligned with and orthogonal to the flow streamlines at the measurement location.

First, the principal flow direction must be calculated through `calc_principal_heading`. As a standard for DOLfYN functions, those that begin with "calc_*" require the velocity data for input. This function is different from others in DOLfYN in that it requires place the output in an attribute called "principal_heading", as shown below.

Again we use `rotate2` to change coordinate systems.

[7]:
```
ds.attrs['principal_heading'] = dolfyn.calc_principal_heading(ds['vel'])
dolfyn.rotate2(ds, 'principal')
```

### 2.2.4 Averaging Data

The next step in ADV analysis is to average the velocity data into time bins (ensembles) and calculate turbulence statistics. There are a couple ways to do this, and both of these methods use the same variable inputs and return identical datasets.

1. Define an averaging object, create a binned dataset and calculate basic turbulence statistics. This is done by initiating an object from the `ADVBinner` class, and subsequently supplying that object with our dataset.
2. Alternatively, the functional version of ADVBinner, `calc_turbulence`.

Function inputs shown here are the dataset itself; "n_bin", the number of elements in each bin; "fs", the ADV's sampling frequency in Hz; "n_fft", optional, the number of elements per FFT for spectral analysis; "freq_units", optional, either in Hz or rad/s, of the calculated spectral frequency vector.

All of the variables in the returned dataset have been bin-averaged, where each average is computed using the number of elements specified in "n_bins". Additional variables in this dataset include the turbulent kinetic energy (TKE) vector ("ds_binned.tke_vec"), the Reynold's stresses ("ds_binned.stress"), and the power spectral densities ("ds_binned.psd"), calculated for each bin.

[8]:
```
# Option 1 (standard)
binner = api.ADVBinner(n_bin=ds.fs*600, fs=ds.fs, n_fft=1024)
ds_binned = binner.do_avg(ds)

# Option 2 (simple)
# ds_binned = api.calc_turbulence(ds, n_bin=ds.fs*600, fs=ds.fs, n_fft=1024, freq_units=
↪"Hz")
```

The benefit to using `ADVBinner` is that one has access to all of the velocity and turbulence analysis functions that DOLfYN contains. If basic analysis will suffice, the `calc_turbulence` function is the most convienent. Either option can still utilize DOLfYN's shortcuts.

See the DOLfYN API for the full list of functions and shortcuts. A few examples are shown below.

Some things to know: - All functions operate bin-by-bin. - Some functions will fail if there are NaN's in the data stream (Notably the PSD functions) - `do_*` functions return a full dataset. The first two inputs are the original dataset and the dataset containing the variables calculated by the function. If an output dataset is not given, it will create one. - `calc_*` functions return a data variable, which can be added to the dataset with a variable of your choosing. If inputs weren't specified in `ADVBinner`, they can be called here. Most of these functions can take both 3D and 1D velocity vectors as inputs. - "Shorcuts", as referred to in DOLfYN, are functions accessible by the xarray accessor `velds`, as shown below. The list of "shorcuts" available through `velds` are listed here. Some shorcut variables require the raw dataset, some an averaged dataset.

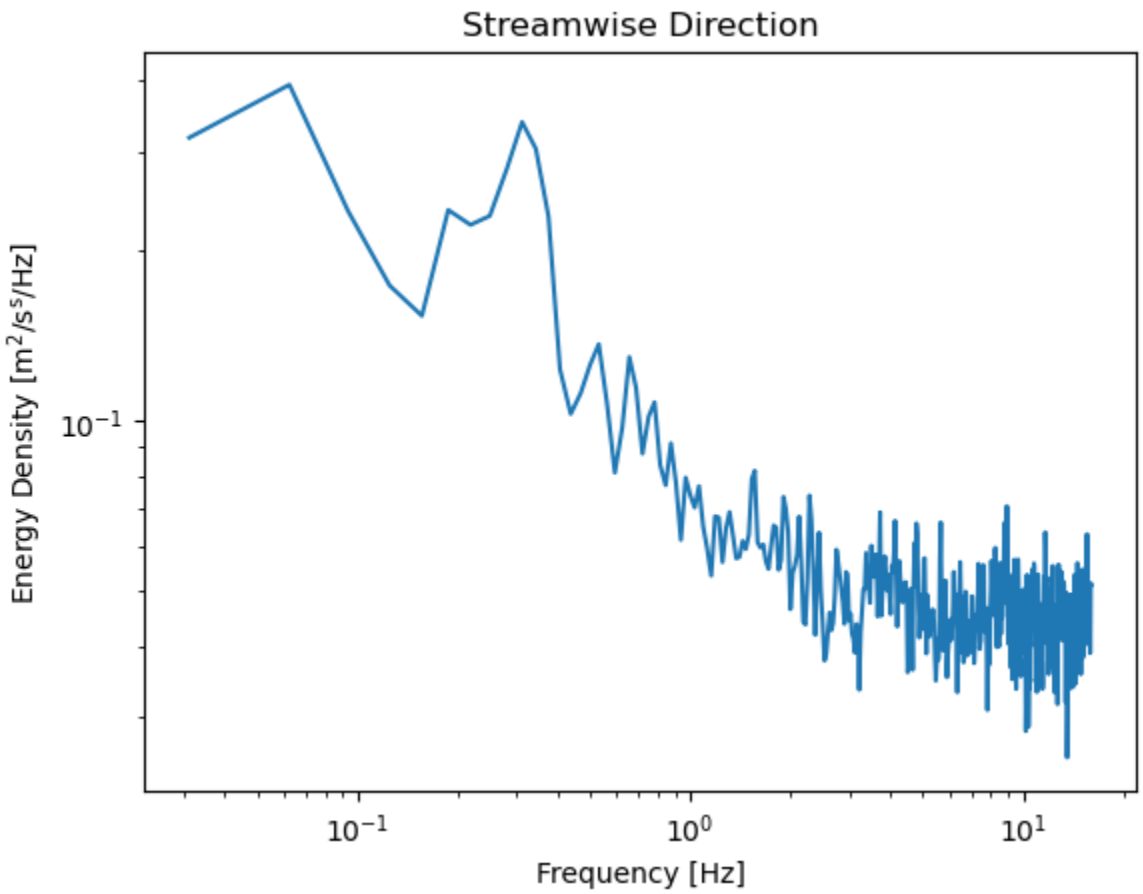
For instance, - `do_var` calculates the binned-variance of each variable in the raw dataset, the complementary to `do_avg`. Variables returned by this function contain a "_var" suffix to their name. - `calc_csd` calculates the cross spectral power density between each direction of the supplied DataArray. Note that inputs specified in creating the `ADVBinner` object can be overridden or additionally specified for a particular function call. - `velds.I` is the shortcut for turbulence intensity. This particular shortcut requires a dataset created by `do_avg`, because it requires bin-averaged data to calculate.

[9]:
```
# Calculate the variance of each variable in the dataset and add to the averaged dataset
ds_binned = binner.do_var(ds, out_ds=ds_binned)

# Calculate the power spectral density
ds_binned['auto_spectra'] = binner.calc_psd(ds['vel'], freq_units='Hz')
# Calculate dissipation rate from isotropic turbulence cascade
ds_binned['dissipation'] = binner.calc_epsilon_LT83(ds_binned['auto_spectra'], ds_binned.
↪velds.U_mag, freq_range=[0.5, 1])

# Calculate the cross power spectral density
ds_binned['cross_spectra'] = binner.calc_csd(ds['vel'], freq_units='Hz', n_fft_coh=512)

# Calculated the turbulence intensity (requires a binned dataset)
ds_binned['TI'] = ds_binned.velds.I
```

Plotting can be taken care of through matplotlib. As an example, the mean spectrum in the streamwise direction is plotted here. This spectrum shows the mean energy density in the flow at a particular flow frequency.

[10]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure()
plt.loglog(ds_binned['freq'], ds_binned['auto_spectra'].sel(S='Sxx').mean(dim='time'))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Energy Density $\mathrm{[m^2/s^s/Hz]}$')
plt.title('Streamwise Direction')
```

[10]:
```
Text(0.5, 1.0, 'Streamwise Direction')
```




### 2.2.5 Saving and Loading DOLfYN datasets

Datasets can be saved and reloaded using the `save` and `load` functions. Xarray is saved natively in netCDF format, hence the ".nc" extension.

Note: DOLfYN datasets cannot be saved using xarray's native `ds.to_netcdf`; however, DOLfYN datasets can be opened using `xarray.open_dataset`.

[11]:
```
# Uncomment these lines to save and load to your current working directory
#dolfyn.save(ds, 'your_data.nc')
#ds_saved = dolfyn.load('your_data.nc')
```

CHAPTER

THREE

# INDICES

- genindex
- modindex

# BIBLIOGRAPHY

[Kilcher_etal_2016] Kilcher, L.; Thomson, J.; Talbert, J.; DeKlerk, A.; 2016, "Measuring Turbulence from Moored Acoustic Doppler Velocimeters" National Renewable Energy Lab, Report Number 62979.

[Harding_etal_2017] Harding, S., Kilcher, L., Thomson, J. (2017). Turbulence Measurements from Compliant Moorings. Part I: Motion Characterization. *Journal of Atmospheric and Oceanic Technology*, 34(6), 1235-1247. doi: 10.1175/JTECH-D-16-0189.1

[Kilcher_etal_2017] Kilcher, L., Thomson, J., Harding, S., & Nylund, S. (2017). Turbulence Measurements from Compliant Moorings. Part II: Motion Correction. *Journal of Atmospheric and Oceanic Technology*, 34(6), 1249-1266. doi: 10.1175/JTECH-D-16-0213.1

# PYTHON MODULE INDEX

## d

dolfyn, 26
dolfyn.adp.clean, 27
dolfyn.adp.turbulence, 58
dolfyn.adv.clean, 32
dolfyn.adv.motion, 33
dolfyn.adv.turbulence, 54
dolfyn.binned, 43
dolfyn.io.api, 35
dolfyn.rotate.api, 38
dolfyn.rotate.base, 40
dolfyn.time, 66
dolfyn.tools.misc, 72
dolfyn.tools.psd, 68
dolfyn.velocity, 46

# INDEX

## Symbols


__call__() (*dolfyn.adv.turbulence.ADVBinner method*), 55


## A


ADPBinner (*class in dolfyn.adp.turbulence*), 58
ADVBinner (*class in dolfyn.adv.turbulence*), 54
attrs (*dolfyn.velocity.Velocity property*), 48


## C


C (*dolfyn.velocity.VelBinner attribute*), 51
calc_acov() (*dolfyn.velocity.VelBinner method*), 53
calc_coh() (*dolfyn.velocity.VelBinner method*), 52
calc_csd() (*dolfyn.adv.turbulence.ADVBinner method*), 55
calc_csd_base() (*dolfyn.binned.TimeBinner method*), 46
calc_dissipation_LT83() (*dolfyn.adp.turbulence.ADPBinner method*), 62
calc_dissipation_SF() (*dolfyn.adp.turbulence.ADPBinner method*), 63
calc_doppler_noise() (*dolfyn.adp.turbulence.ADPBinner method*), 60
calc_doppler_noise() (*dolfyn.adv.turbulence.ADVBinner method*), 55
calc_dudz() (*dolfyn.adp.turbulence.ADPBinner method*), 59
calc_dvdz() (*dolfyn.adp.turbulence.ADPBinner method*), 59
calc_dwdz() (*dolfyn.adp.turbulence.ADPBinner method*), 59
calc_epsilon_LT83() (*dolfyn.adv.turbulence.ADVBinner method*), 56
calc_epsilon_SF() (*dolfyn.adv.turbulence.ADVBinner method*), 57
calc_epsilon_TE01() (*dolfyn.adv.turbulence.ADVBinner method*), 57
calc_freq() (*dolfyn.binned.TimeBinner method*), 46
calc_L_int() (*dolfyn.adv.turbulence.ADVBinner method*), 58
calc_phase_angle() (*dolfyn.velocity.VelBinner method*), 52
calc_principal_heading() (*in module dolfyn.rotate.api*), 38
calc_psd() (*dolfyn.velocity.VelBinner method*), 54
calc_psd_base() (*dolfyn.binned.TimeBinner method*), 45
calc_shear2() (*dolfyn.adp.turbulence.ADPBinner method*), 60
calc_stress() (*dolfyn.adv.turbulence.ADVBinner method*), 55
calc_stress_4beam() (*dolfyn.adp.turbulence.ADPBinner method*), 60
calc_stress_5beam() (*dolfyn.adp.turbulence.ADPBinner method*), 61
calc_tilt() (*in module dolfyn.rotate.base*), 41
calc_tke() (*dolfyn.velocity.VelBinner method*), 53
calc_total_tke() (*dolfyn.adp.turbulence.ADPBinner method*), 61
calc_turbulence() (*in module dolfyn.adv.turbulence*), 58
calc_ustar_fit() (*dolfyn.adp.turbulence.ADPBinner method*), 64
calc_velacc() (*dolfyn.adv.motion.CalcMotion method*), 33
calc_velrot() (*dolfyn.adv.motion.CalcMotion method*), 34
calc_xcov() (*dolfyn.velocity.VelBinner method*), 53
CalcMotion (*class in dolfyn.adv.motion*), 33
check_turbulence_cascade_slope() (*dolfyn.adp.turbulence.ADPBinner method*), 62
check_turbulence_cascade_slope() (*dolfyn.adv.turbulence.ADVBinner method*), 56
clean_fill() (*in module dolfyn.adv.clean*), 32

coherence() (*in module dolfyn.tools.psd*), 69
convert_degrees() (*in module dolfyn.tools.misc*), 74
coords (*dolfyn.velocity.Velocity property*), 48
correct_motion() (*in module dolfyn.adv.motion*), 34
correlation_filter() (*in module dolfyn.adp.clean*), 29
cpsd() (*in module dolfyn.tools.psd*), 70
cpsd_quasisync() (*in module dolfyn.tools.psd*), 69

## D

date2dt64() (*in module dolfyn.time*), 66
date2epoch() (*in module dolfyn.time*), 67
date2matlab() (*in module dolfyn.time*), 67
date2str() (*in module dolfyn.time*), 67
demean() (*dolfyn.binned.TimeBinner method*), 44
detrend() (*dolfyn.binned.TimeBinner method*), 44
detrend() (*in module dolfyn.tools.misc*), 72
do_avg() (*dolfyn.velocity.VelBinner method*), 51
do_var() (*dolfyn.velocity.VelBinner method*), 51
dolfyn
    module, 26
dolfyn.adp.clean
    module, 27
dolfyn.adp.turbulence
    module, 58
dolfyn.adv.clean
    module, 32
dolfyn.adv.motion
    module, 33
dolfyn.adv.turbulence
    module, 54
dolfyn.binned
    module, 43
dolfyn.io.api
    module, 35
dolfyn.rotate.api
    module, 38
dolfyn.rotate.base
    module, 40
dolfyn.time
    module, 66
dolfyn.tools.misc
    module, 72
dolfyn.tools.psd
    module, 68
dolfyn.velocity
    module, 46
dt642date() (*in module dolfyn.time*), 66
dt642epoch() (*in module dolfyn.time*), 66

## E

E_coh (*dolfyn.velocity.Velocity property*), 49
epoch2date() (*in module dolfyn.time*), 66
epoch2dt64() (*in module dolfyn.time*), 66
euler2orient() (*in module dolfyn.rotate.base*), 40

## F

fill_nan_ensemble_mean() (*in module dolfyn.adv.clean*), 32
fillgaps() (*in module dolfyn.tools.misc*), 73
fillgaps_depth() (*in module dolfyn.adp.clean*), 30
fillgaps_time() (*in module dolfyn.adp.clean*), 30
find_surface() (*in module dolfyn.adp.clean*), 28
find_surface_from_P() (*in module dolfyn.adp.clean*), 28

## G

GN2002() (*in module dolfyn.adv.clean*), 33
group() (*in module dolfyn.tools.misc*), 72

## I

I (*dolfyn.velocity.Velocity property*), 49
I_tke (*dolfyn.velocity.Velocity property*), 49
interpgaps() (*in module dolfyn.tools.misc*), 73

## L

load() (*in module dolfyn.io.api*), 36
load_mat() (*in module dolfyn.io.api*), 37

## M

matlab2date() (*in module dolfyn.time*), 67
mean() (*dolfyn.binned.TimeBinner method*), 44
medfilt_orient() (*in module dolfyn.adp.clean*), 29
medfiltnan() (*in module dolfyn.tools.misc*), 73
module
    dolfyn, 26
    dolfyn.adp.clean, 27
    dolfyn.adp.turbulence, 58
    dolfyn.adv.clean, 32
    dolfyn.adv.motion, 33
    dolfyn.adv.turbulence, 54
    dolfyn.binned, 43
    dolfyn.io.api, 35
    dolfyn.rotate.api, 38
    dolfyn.rotate.base, 40
    dolfyn.time, 66
    dolfyn.tools.misc, 72
    dolfyn.tools.psd, 68
    dolfyn.velocity, 46

## N

nan_beyond_surface() (*in module dolfyn.adp.clean*), 28

## O

orient2euler() (*in module dolfyn.rotate.base*), 40

## P

phase_angle() (*in module dolfyn.tools.psd*), 71
psd() (*in module dolfyn.tools.psd*), 70
psd_freq() (*in module dolfyn.tools.psd*), 68

## Q

quaternion2orient() (*in module dolfyn.rotate.base*), 41

## R

range_limit() (*in module dolfyn.adv.clean*), 33
read() (*in module dolfyn.io.api*), 35
read_example() (*in module dolfyn.io.api*), 36
reshape() (*dolfyn.adv.motion.CalcMotion method*), 33
reshape() (*dolfyn.binned.TimeBinner method*), 43
rotate2() (*dolfyn.velocity.Velocity method*), 47
rotate2() (*in module dolfyn.rotate.api*), 38

## S

S (*dolfyn.velocity.VelBinner attribute*), 51
save() (*dolfyn.velocity.Velocity method*), 48
save() (*in module dolfyn.io.api*), 36
save_mat() (*in module dolfyn.io.api*), 36
set_declination() (*dolfyn.velocity.Velocity method*), 47
set_declination() (*in module dolfyn.rotate.api*), 39
set_inst2head_rotmat() (*dolfyn.velocity.Velocity method*), 48
set_inst2head_rotmat() (*in module dolfyn.rotate.api*), 39
set_range_offset() (*in module dolfyn.adp.clean*), 27
slice1d_along_axis() (*in module dolfyn.tools.misc*), 72
spike_thresh() (*in module dolfyn.adv.clean*), 33
std() (*dolfyn.binned.TimeBinner method*), 45
stepsize() (*in module dolfyn.tools.psd*), 68

## T

tau (*dolfyn.velocity.VelBinner attribute*), 50
TimeBinner (*class in dolfyn.binned*), 43
tke (*dolfyn.velocity.VelBinner attribute*), 50
tke (*dolfyn.velocity.Velocity property*), 49

## U

U (*dolfyn.velocity.Velocity property*), 49
u (*dolfyn.velocity.Velocity property*), 48
U_dir (*dolfyn.velocity.Velocity property*), 49
U_mag (*dolfyn.velocity.Velocity property*), 49
upup_ (*dolfyn.velocity.Velocity property*), 50
upvp_ (*dolfyn.velocity.Velocity property*), 49
upwp_ (*dolfyn.velocity.Velocity property*), 50

## V

v (*dolfyn.velocity.Velocity property*), 49
val_exceeds_thresh() (*in module dolfyn.adp.clean*), 29
var() (*dolfyn.binned.TimeBinner method*), 44
variables (*dolfyn.velocity.Velocity property*), 48
VelBinner (*class in dolfyn.velocity*), 50
Velocity (*class in dolfyn.velocity*), 46
vpvp_ (*dolfyn.velocity.Velocity property*), 50
vpwp_ (*dolfyn.velocity.Velocity property*), 50

## W

w (*dolfyn.velocity.Velocity property*), 49
wpwp_ (*dolfyn.velocity.Velocity property*), 50