Read the Docs Template Documentation

Release 2023.7

Read the Docs

Apr 22, 2024

GETTING STARTED

Glider tools is a Python 3.8+ package designed to process data from the first level of processing to a science ready dataset. The package is designed to easily import data to a standard column format: `numpy.ndarray`, `pandas.DataFrame` or `xarray.DataArray` (we recommend the latter which has full support for metadata). Cleaning and smoothing functions are flexible and can be applied as required by the user. We provide examples and demonstrate best practices as developed by the SOCCO Group.

For the original publication of this package see: https://doi.org/10.3389/fmars.2019.00738.

For recommendations or bug reports, please visit https://github.com/GliderToolsCommunity/GliderTools/issues/new

CHAPTER

ONE

INSTALLATION

Notes on how to install

1.1 Conda

The easiest way to install the packge is with conda: `conda install -c conda-forge glidertools`.

1.2 PyPI

You can also install with pip: `pip install glidertools`.

1.3 GitHub

For the most up to date version of GliderTools, you can install directly from the online repository hosted on GitLab.

1. Clone glidertools to your local machine: `git clone https://github.com/GliderToolsCommunity/GliderTools`
2. Change to the parent directory of GliderTools
3. Install glidertools with `pip install -e ./GliderTools`. This will allow changes you make locally, to be reflected when you import the package in Python

CHAPTER

TWO

CHEAT SHEET

GLIDER TOOLS

CHEAT SHEET

.load

seaglider_show_variables displays variable names, dimension and units for glider files

seaglider_basesation_netCDFs loads, concatenates and merges variables from a list of filenames

.plot

pcolormesh, contourf an automatically gridded section if x- and y-coordinates are given. *x* should be profile number or average time per profile

scatter shows the yaw of the glider if time and depth are given as x and y-coordinates

glider_tools

glider_tools contains high level functions that incorporate several low level functions to speed up processing and establish a standard procedure for processing variables. These are:

calc_physics
calc_par
calc_backscatter
calc_fluorescence
calc_oxygen

Please see the documentation for more help on these functions: glidertools.rtfd.io

Note that this cheat sheet only shows the essential functions to process glider data.

.cleaning

FILTERING

outlier_bounds_std standard deviations

outlier_bounds_iqr interquartile ranges

horizontal_diff_outliers Find horizontal outliers below a depth threshold. Masks dive if a large fraction is marked an outlier.

mask_bad_dive_fraction Find bad dives - where more than a fraction of the dive is masked

blob_outliers excludes points if they are outside a data density cloud threshold. Not a standard statistical method, thus last resort.

DESPIKING AND SMOOTHING

despike from Briggs et al. (2011): a rolling minimum/ median is applied to find a baseline. The difference between the measurements and the baseline are returned as spikes.

rolling_windowapplies a statistical function (mean, median, std, percentile) to a rolling window. Also works if data has NaNs.

savitzky_golay fits a low order polynomial to a rolling window of the time series. Has the result of smoothing the data.

.physics

mixed_layer_depth calculates the mixed layer depth for each dive and returns a mask or array of depths

potential_density calculates potential density with the correct parameterisations and conversions (wrapper for `gsw.pot_rho_t_exact`)

brunt_vaisala calculates a wrapper around the gsw calculation of `gsw.Nsquared` to account for NaNs

.mapping

grid_dat bin data to standard depths taking the average per depth bin. Custom bins can be given or optimal bins (per 50 m) are calculated automatically if bins not specified.

interp_obj interpolate data objectively (a.k.a. Kriging). (the same as the MATLAB objmap function).

.optics

NOTE: there are functions to calculate backscatter, chlorophyll and PAR from raw to scaled units with the prefix `gt.calc_*`. These apply several functions in the `optics`, `filtering` and `flow_functions` to process data. We recommend creating custom functions such as these for your own lab.

photic_depth returns the euphotic depth and attenuation coefficient (Kd), based upon the linear fit of the natural log of par with depth. Default reference percentage is 1%.

par_fill_surface use exponential regression to

.utils

time_average_per_dive use to calculate the average time for a dive to create pseudo-discrete times for plotting

CHAPTER

THREE

LOADING DATA

To start using Glider Tools you first need to import the package to the interactive workspace.

3.1 Import `GliderTools`

```
# pylab for more MATLAB like environment and inline displays plots below cells
%pylab inline

# if gsw Warning shows, manually install gsw if possible - will still work without
import glidertools as gt
from cmocean import cm as cmo  # we use this for colormaps
```

```
Populating the interactive namespace from numpy and matplotlib
```

3.2 Working with Seaglider base station files

GliderTools supports loading Seaglider files, including `scicon` data (different sampling frequencies). There is a function that makes it easier to find variable names that you'd like to load: `gt.load.seaglider_show_variables`

This function is demonstrated in the cell below. The function accepts a **list of file names** and can also receive a string with a wildcard placeholder (*) and basic regular expressions are also supported. In the example below we use a simple asterisk placeholder for all the files.

Note that the function chooses only one file from the passed list or glob string - this file name will be shown. The returned table shows the variable name, dimensions, units and brief comment if it is available.

```
filenames = '/Users/luke/Work/Data/sg542/p5420*.nc'

gt.load.seaglider_show_variables(filenames)
```

```
information is based on file: /Users/luke/Work/Data/sg542/p5420177.nc

<table will be displayed here>
```

3.3 Working with VOTO Seaexplorer files or xarray-datasets

Glidertools supports loading Seaexplorer files. This is implemented and tested with `VOTO https://observations.voiceoftheocean.org_` datasets in mind currently, but we are happy about feedback/pullrequests how it works for other SeaExplorer datasets. VOTO data can either be downloaded from the website using a browser or, more comfortable, from an `ERDAP server https://erddap.observations.voiceoftheocean.org/erddap/index.html_` . See the `demo notebook <https://github.com/voto-ocean-knowledge/download_glider_data>_` to get started with downloads over the API.

After download of a .nc file or xarray-Dataset, it can be read into Glidertools by calling `gt.load.voto_seaexplorer_nc` or `gt.load.voto_seaexplorer_dataset` respectively. Resulting datasets can be merged by calling `gt.load.voto_concat_datasets`. The import of the data into GliderTools is hereby finished, remaining steps on this wiki-page are optional.

3.4 Load variables

From the variable listing, one can choose multiple variables to load. Note that one only needs the variable name to load the data. Below, we've created a list of variables that we'll be using for this demo.

The `gt.load.seaglider_basestation_netCDFs` function is used to load a list of variables. It requires the filename string or list (as described above) and keys. It may be that these variables are not sampled at the same frequency. In this case, the loading function will load the sampling frequency dimensions separately. The function will try to find a time variable for each sampling frequency/dimension.

3.4.1 Coordinates and automatic *time* fetching

All associated coordinate variables will also be loaded with the data if coordinates are documented. These may included *latitude, longitude, depth* and *time* (naming may vary). If time cannot be found for a dimension, a *time* variable from a different dimension with the same number of observations is used instead. This insures that data can be merged based on the time of sampling.

3.4.2 Merging data based on time

If the `return_merged` is set to *True*, the function will merge the dimensions if the dimension has an associated *time* variable.

The function returns a dictionary of `xarray.Datasets` - a Python package that deals with coordinate indexed multi-dimensional arrays. We recommend that you read the documentation (http://xarray.pydata.org/en/stable/) as this package is used throughout *GliderTools*. This allows the original metadata to be copied with the data. The dictionary keys are the names of the dimensions. If `return_merged` is set to *True* an additional entry under the key `merged` will be included.

The structure of a dimension output is shown below. Note that the merged data will use the largest dimension as the primary dataset and the other data will be merged onto that time index. Data is linearly interpolated to the nearest time measurement of the primary index, but only by one measurement to ensure transparancy.

```
names = [
    'ctd_depth',
    'ctd_time',
    'ctd_pressure',
    'salinity',
```

(continues on next page)

(continued from previous page)

```
    'temperature',
    'eng_wlbb2flvmt_Chlsig',
    'eng_wlbb2flvmt_wl470sig',
    'eng_wlbb2flvmt_wl700sig',
    'aanderaa4330_dissolved_oxygen',
    'eng_qsp_PARuV',
]

ds_dict = gt.load.seaglider_basestation_netCDFs(
    filenames, names,
    return_merged=True,
    keep_global_attrs=False
)
```

```
DIMENSION: sg_data_point
{
    ctd_pressure, eng_wlbb2flvmt_wl470sig, eng_wlbb2flvmt_wl700sig, temperature,
    ctd_time, ctd_depth, latitude, aanderaa4330_dissolved_oxygen, salinity,
    eng_wlbb2flvmt_Chlsig, longitude
}


100%|| 336/336 [00:04<00:00, 73.66it/s]



DIMENSION: qsp2150_data_point
{eng_qsp_PARuV, time}


100%|| 336/336 [00:01<00:00, 181.67it/s]



Merging dimensions on time indicies: sg_data_point, qsp2150_data_point,
```

The returned data contains the dimensions of the requested variables a `merged` object is also returned if return_merged=True

```

print(ds_dict.keys())
```

```
dict_keys(['sg_data_point', 'qsp2150_data_point', 'merged'])
```

3.4.3 Metadata handling

If the keyword arguement `keep_global_attrs=True`, the attributes from the original files (for all that are the same) are passed on to the output *Datasets* from the original netCDF attributes. The variable attributes (units, comments, axis…) are passed on by default, but can also be set to False if not wanted. GliderTools functions will automatically pass on these attributes to function outputs if a `xarray.DataArray` with attributes is given. All functions applied to data will also be recorded under the variable attribute `processing`.

The merged dataset contains all the data interpolated to the nearest observation of the longest dimension the metadata is also shown for the example below

```
ds_dict['merged']
```

```
xarray.Dataset>
Dimensions:                         (merged: 382151)
Coordinates:
    ctd_depth                       (merged) float64 -0.08821 0.018 ... -0.1422
    latitude                        (merged) float64 -42.7 -42.7 ... -43.0 -43.0
    longitude                       (merged) float64 8.744 8.744 ... 8.5 8.5
    ctd_time_dt64                   (merged) datetime64[ns] 2015-12-08T07:36:16 ...

Dimensions without coordinates: merged
Data variables:
    ctd_pressure                    (merged) float64 -0.08815 0.01889 ... -0.1432
    eng_wlbb2flvmt_wl470sig         (merged) float64 375.0 367.0 ... 98.0 91.0
    eng_wlbb2flvmt_wl700sig         (merged) float64 2.647e+03 ... 137.0
    temperature                     (merged) float64 11.55 11.54 ... 11.06 10.97
    ctd_time                        (merged) float64 1.45e+09 ... 1.455e+09
    aanderaa4330_dissolved_oxygen   (merged) float64 nan nan nan ... 269.1 269.1
    salinity                        (merged) float64 nan nan nan ... 34.11 34.11
    eng_wlbb2flvmt_Chlsig           (merged) float64 145.0 126.0 ... 215.0 215.0
    dives                           (merged) float64 1.0 1.0 1.0 ... 344.5 344.5
    eng_qsp_PARuV                   (merged) float64 0.551 0.203 ... 0.021 0.023
    time                            (merged) float64 1.45e+09 ... 1.455e+09
    time_dt64                       (merged) datetime64[ns] 2015-12-08T07:36:16 ...

Attributes:
    date_created:              2019-07-11 14:08:40
    number_of_dives:           344.0
    files:                     ['p5420001.nc', 'p5420002.nc', 'p5420004.nc', '...
    time_coverage_start:       2015-12-08 07:36:16
    time_coverage_end:         2016-02-08 04:39:04
    geospatial_vertical_min:   -0.6323553853732649
    geospatial_vertical_max:   1011.1000623417478
    geospatial_lat_min:        -43.085757609206
    geospatial_lat_max:        -42.70088638031523
    geospatial_lon_min:        8.29983279020758
    geospatial_lon_max:        8.7753734452125
    processing:                [2019-07-11 14:08:40] imported data with Glider...
```

3.4.4 Renaming for ease of access

When renaming, just be sure that there are no variables with names that you are trying to replace. In the example below we remove `time` in case it exists in the files.

```
# Here we drop the time variables imported for the PAR variable
# we don't need these anymore. You might have to change this
# depening on the dataset
merged = ds_dict['merged']
if 'time' in merged:
    merged = merged.drop(["time", "time_dt64"])


# To make it easier and clearer to work with, we rename the
# original variables to something that makes more sense. This
# is done with the xarray.Dataset.rename({}) function.
# We only use the merged dataset as this contains all the
# imported dimensions.
# NOTE: The renaming has to be specific to the dataset otherwise an error will occur
dat = merged.rename({
    'salinity': 'salt_raw',
    'temperature': 'temp_raw',
    'ctd_pressure': 'pressure',
    'ctd_depth': 'depth',
    'ctd_time_dt64': 'time',
    'ctd_time': 'time_raw',
    'eng_wlbb2flvmt_wl700sig': 'bb700_raw',
    'eng_wlbb2flvmt_wl470sig': 'bb470_raw',
    'eng_wlbb2flvmt_Chlsig': 'flr_raw',
    'eng_qsp_PARuV': 'par_raw',
    'aanderaa4330_dissolved_oxygen': 'oxy_raw',
})

print(dat)

# variable assignment for conveniant access
depth = dat.depth
dives = dat.dives
lats = dat.latitude
lons = dat.longitude
time = dat.time
pres = dat.pressure
temp = dat.temp_raw
salt = dat.salt_raw
par = dat.par_raw
bb700 = dat.bb700_raw
bb470 = dat.bb470_raw
fluor = dat.flr_raw

# name coordinates for quicker plotting
x = dat.dives
y = dat.depth
```

CHAPTER

FOUR

QUALITY CONTROL

Note that this summary carries on from the *Loading data* page.

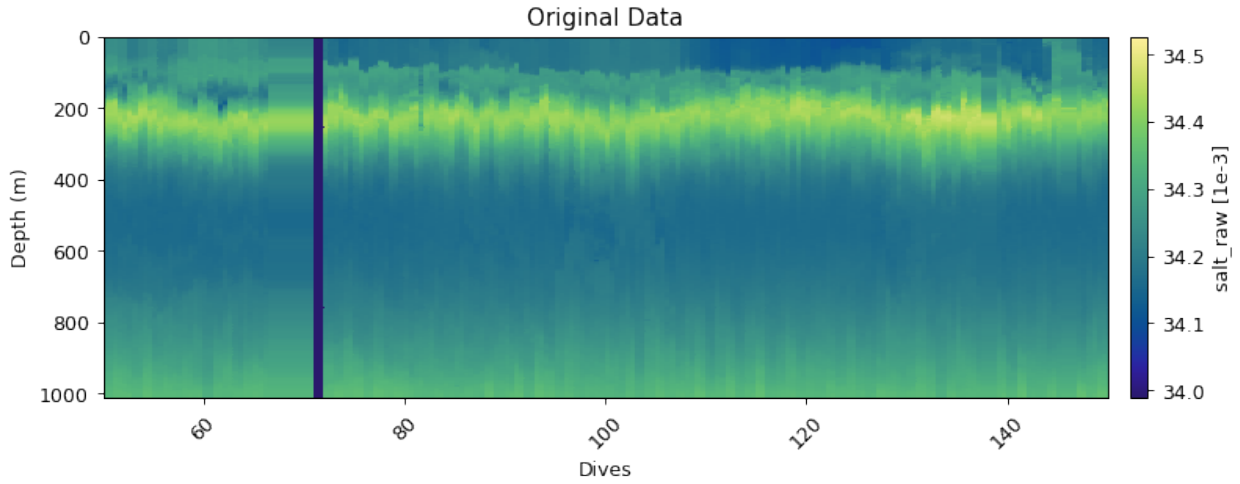
The `cleaning` module contains several tools that help to remove erroneous data - profiles or points. These filters can be applied *globally* (IQR and standard devation limits), *vertically* (running average filters) or *horizontally* (horizontal filters on gridded data only).

There are also two approaches one can use to clean data: 1) filtering out bad points/dives; 2) smoothing data.

4.1 Original Data

Below we use **salinity** to demonstrate the different functions available to users.

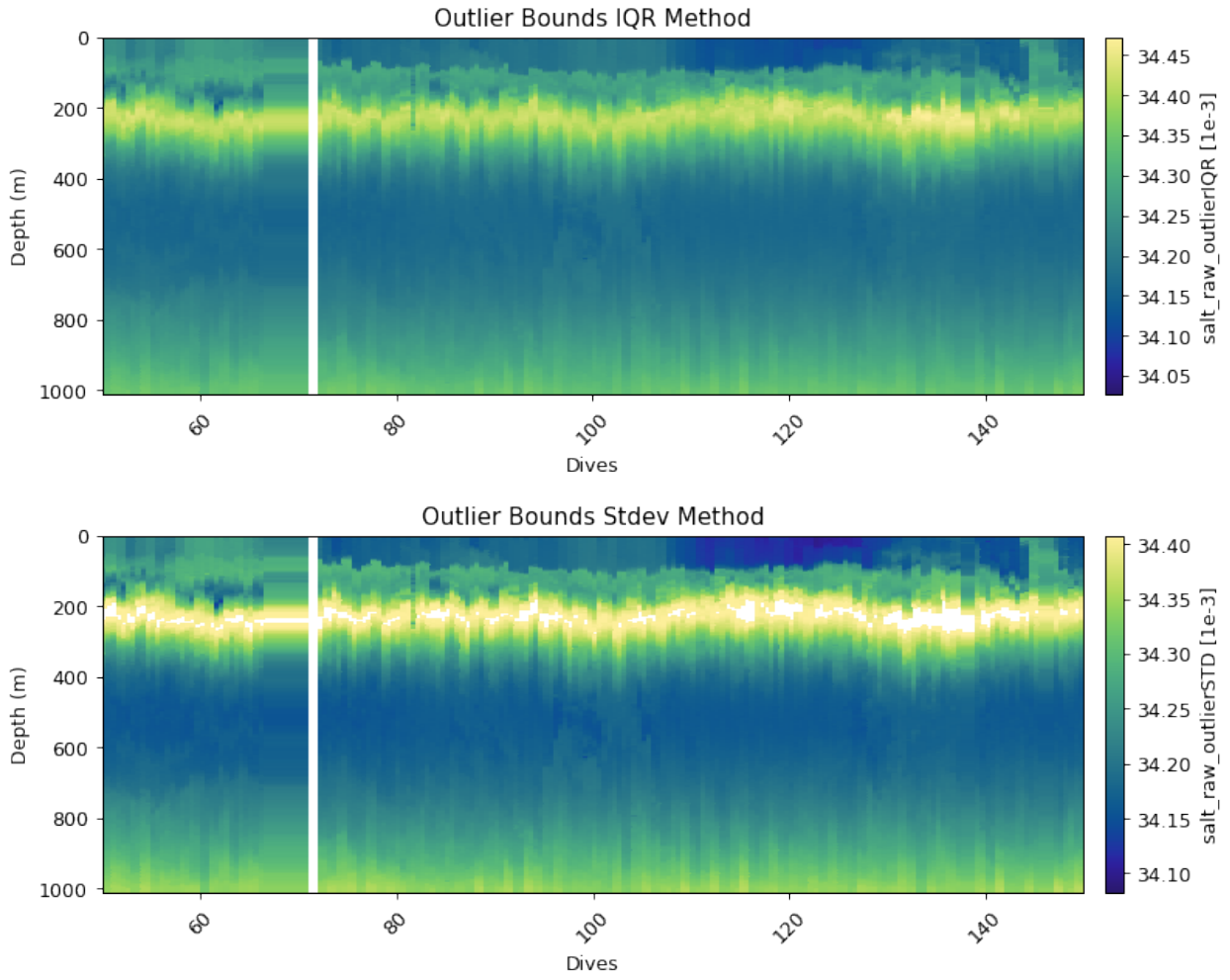
```
dives = dat.dives
depth = dat.depth
salt = dat.salinity_raw

x = np.array(dives)  # ensures these are arrays
y = np.array(depth)

gt.plot(dives, depth, salt, cmap=cmo.haline, robust=True)
plt.xlim(50, 150)
plt.title('Original Data')
plt.show()
```



4.2 Global filtering: outlier limits (IQR & STD)

These functions find upper and lower limits for data outliers using interquartile range and standard deviations of the entire dataset. Multipliers can be set to make the filters more or less strict

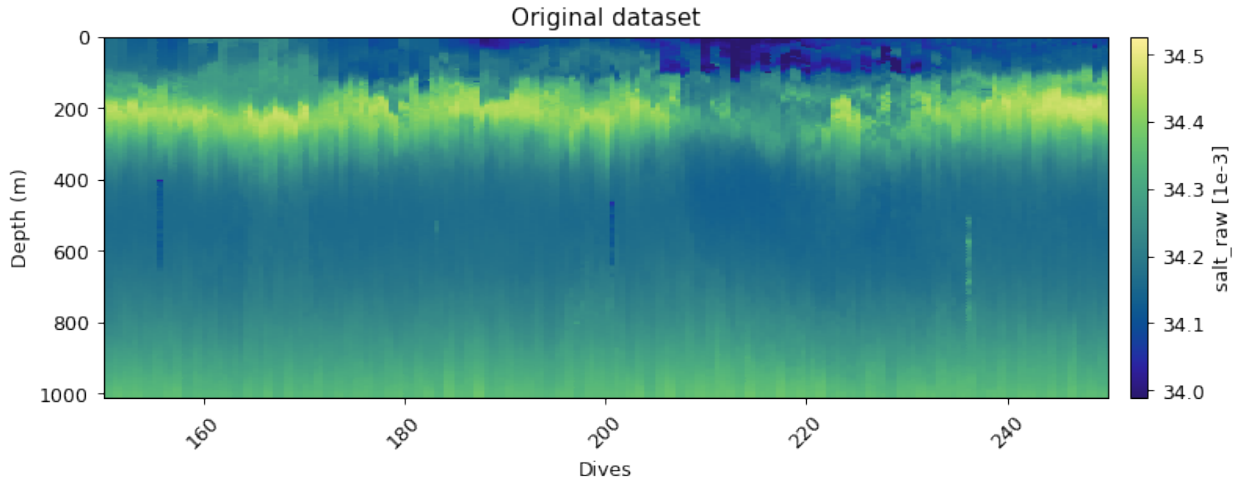
```
salt_iqr = gt.cleaning.outlier_bounds_iqr(salt, multiplier=1.5)
salt_std = gt.cleaning.outlier_bounds_std(salt, multiplier=1.5)

# Plotting
gt.plot(x, y, salt_iqr, cmap=cmo.haline, robust=True)
plt.title('Outlier Bounds IQR Method')
plt.xlim(50,150)

gt.plot(x, y, salt_std, cmap=cmo.haline, robust=True)
plt.title('Outlier Bounds Stdev Method')
plt.xlim(50,150)

plt.show()
```

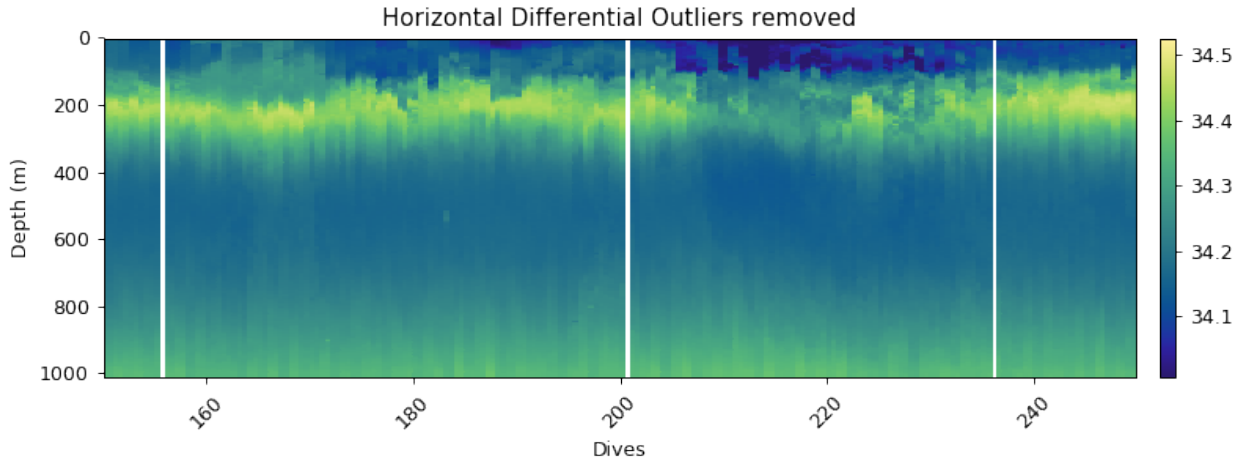


4.3 Horizontal filtering: differential outliers

Erroneous measurements often occur sequentially - i.e. in the vertical. The vertical filtering approaches would thus miss any outliers as rolling windows are often used. It is thus useful to have an approach that compares dives in the horizontal. The `horizontal_diff_outliers` first grids data and then calculates where gradients (rolling mean - measurement) are outliers (same as `outlier_bounds_std`). If a certain fraction of measurements in a dive exceed the threshold, then that dive is deemed a bad dive. The example below shows three dives that have anomalous measurements. These fall well within the global bounds of acceptable data, but horizontally that are masked out.

```
salt_horz = gt.cleaning.horizontal_diff_outliers(
    x, y, salt,
    multiplier=3,
    depth_threshold=400,
    mask_frac=0.1
)

gt.plot(x, y, salt, cmap=cmo.haline)
plt.title('Original dataset')
plt.xlim(150,250)
plt.show()

gt.plot(x, y, salt_horz, cmap=cmo.haline)
plt.title('Horizontal Differential Outliers removed')
plt.xlim(150,250)
plt.show()
```





4.4 Vertical smoothing approaches

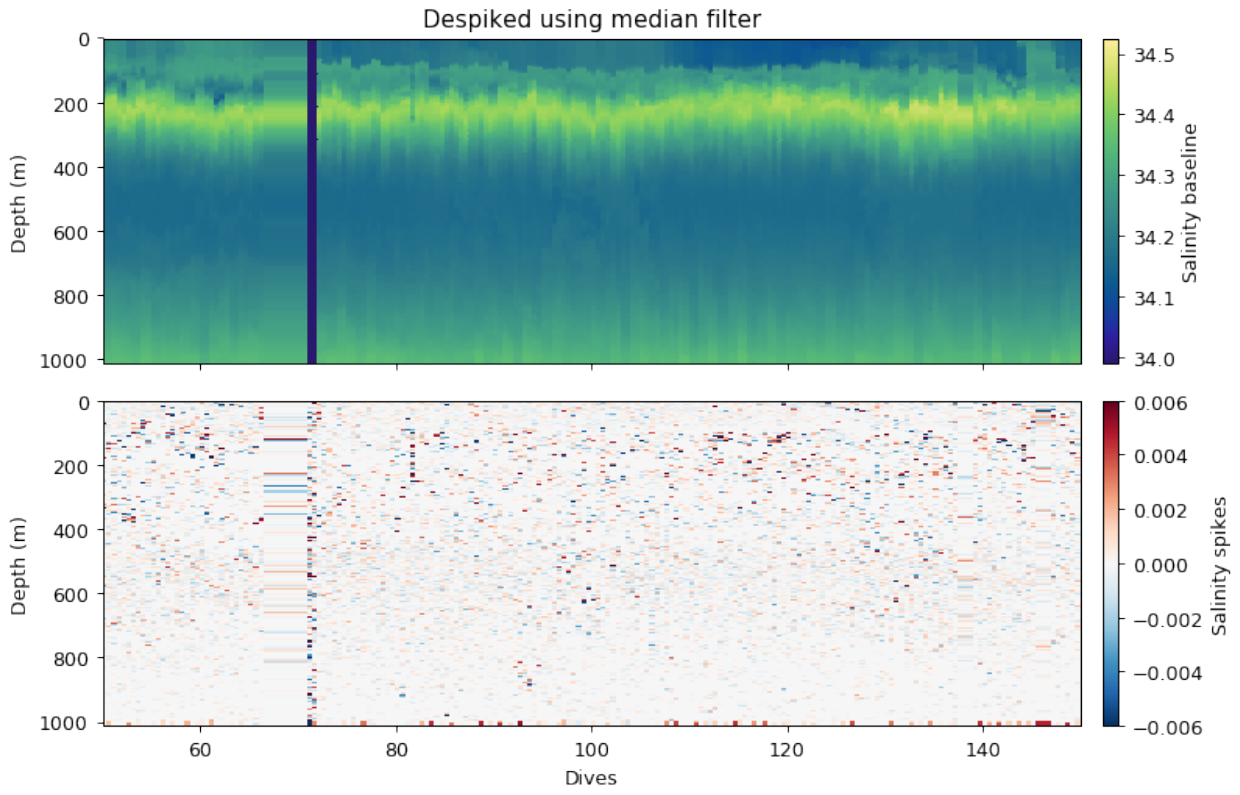
4.4.1 Despiking

This approach was used by Briggs et al. (2010). The idea is to apply a rolling filter to the data (along the time dimension). This forms the baseline. The difference from the original data are spikes.

There are two rolling filters that can be applied to the data. The *median* approach is the equivalent of a rolling median. The *minmax* approach first applies a rolling minimum and then rolling maximum to data. This is useful particularly for optics data where spikes are particles in the water column and are not normally distributed.

In the case of salinity, the *median* approach is likely best, as "spikes" would be positive and negative (Gaussian distribution).

```
salt_base, salt_spike = gt.cleaning.despike(salt, window_size=5, spike_method='median')

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, salt_base, cmap=cmo.haline, ax=ax[0])
ax[0].set_title('Despiked using median filter')
ax[0].cb.set_label('Salinity baseline')
ax[0].set_xlim(50,150)
ax[0].set_xlabel('')

gt.plot(x, y, salt_spike, cmap=cm.RdBu_r, vmin=-6e-3, vmax=6e-3, ax=ax[1])
ax[1].cb.set_label('Salinity spikes')
ax[1].set_xlim(50,150)

plt.xticks(rotation=0)
plt.show()
```



4.4.2 Rolling window

The rolling window method simply applies an aggregating function (`mean, median, std, min, max`) to the dataset. Because the above example is equivalent to a rolling median, we show what a rolling `75th percentile` looks like instead.

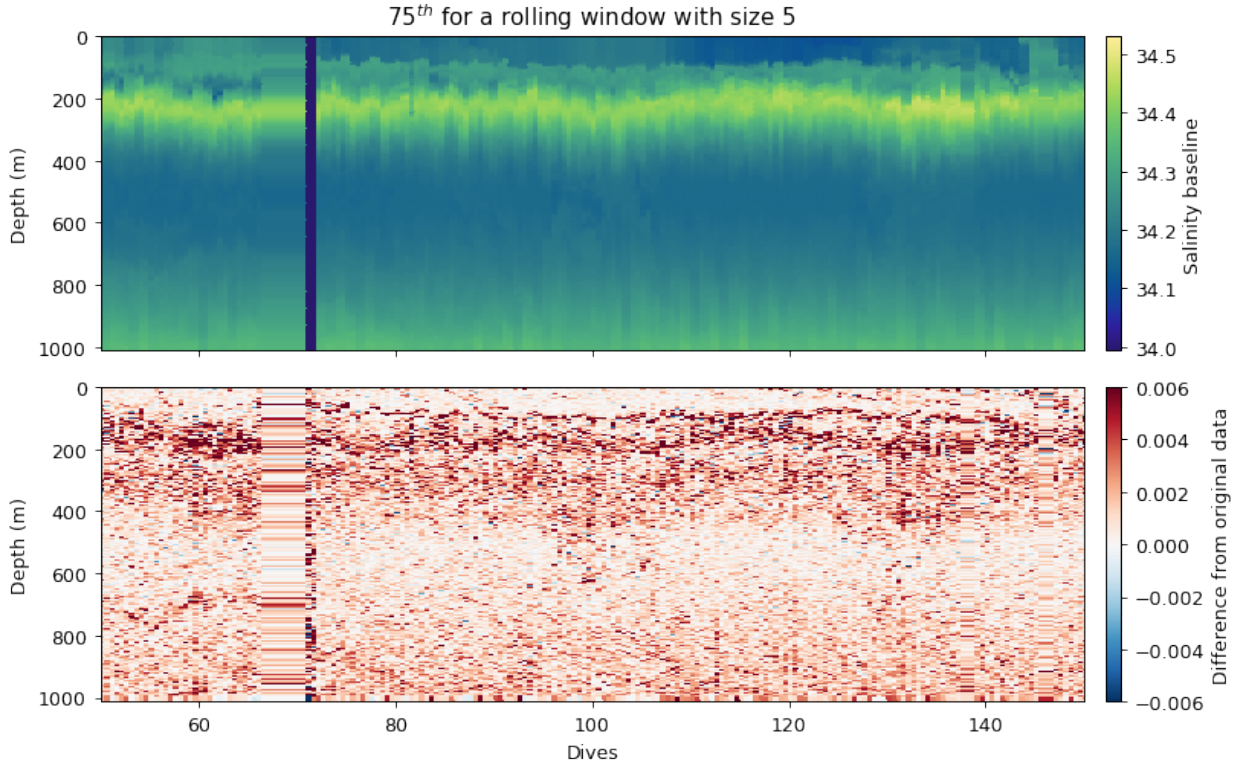
This could be used to create additional filters by users. Note that in this more complex example we create a wrapper function for the percentile so that we can tell the percentile function that we want the 75th percentile and we want to calculate this along the nth axis.

```
def seventyfith(x, axis=0):
    # wrapper function so we can pass axis and percentile to
    # the input function
    return np.percentile(x, 75, axis=axis)

# other numpy functions also work: np.mean, np.median, np.std
salt_roll75 = gt.cleaning.rolling_window(salt, seventyfith, window=5)
salt_rollavg = gt.cleaning.rolling_window(salt, mean, window=5)

# PLOTTING
fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, salt_roll75, cmap=cmo.haline, ax=ax[0])
ax[0].set_title('75$^{th}$ for a rolling window with size 5')
ax[0].cb.set_label('Salinity baseline')
ax[0].set_xlim(50,150)
ax[0].set_xlabel('')
```

(continues on next page)

(continued from previous page)

```
gt.plot(x, y, salt_roll75 - salt, cmap=cm.RdBu_r, vmin=-6e-3, vmax=6e-3, ax=ax[1])
ax[1].cb.set_label('Difference from original data')
ax[1].set_xlim(50,150)

plt.xticks(rotation=0)
plt.show()
```



4.4.3 Savitzky-Golay

The Savitzky-Golay function fits a low order polynomial to a rolling window of the time series. This has the result of smoothing the data. A larger window with a lower order polynomial with have a smoother fit.

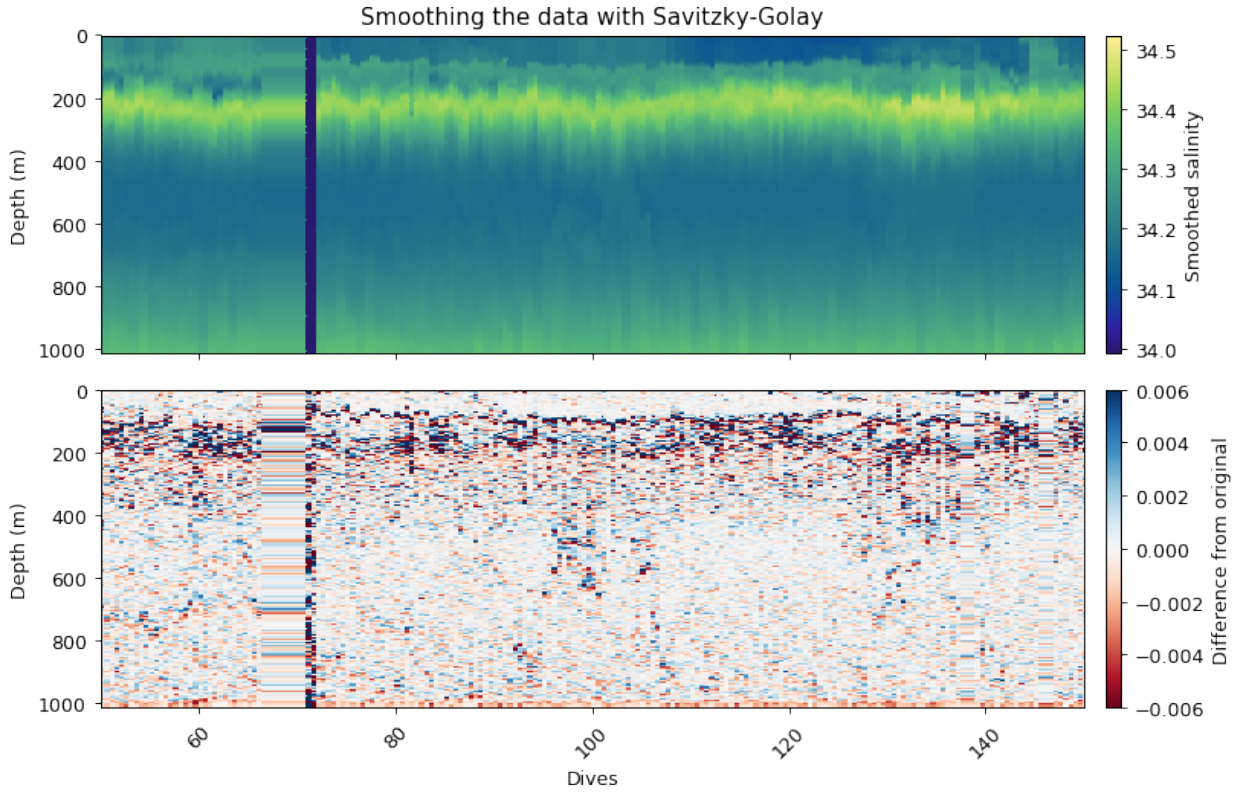
We recommend a 2nd order kernel. Here we use first order to show that the difference can be quite big.

```
salt_savgol = gt.cleaning.savitzky_golay(salt, window_size=11, order=1)

# PLOTTING
fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, salt_savgol, cmap=cmo.haline, ax=ax[0])
ax[0].set_title('Smoothing the data with Savitzky-Golay')
ax[0].cb.set_label('Smoothed salinity')
ax[0].set_xlim(50,150)
ax[0].set_xlabel('')
```

(continues on next page)

(continued from previous page)

```
gt.plot(x, y, salt_savgol - salt, cmap=cm.RdBu, vmin=-6e-3, vmax=6e-3, ax=ax[1])
ax[1].cb.set_label('Difference from original')
ax[1].set_xlim(50,150)

plt.show()
```



4.5 Wrapper functions

Wrapper functions have been designed to make this process more efficient, which is demonstrated below with **temperature** and **salinity**.

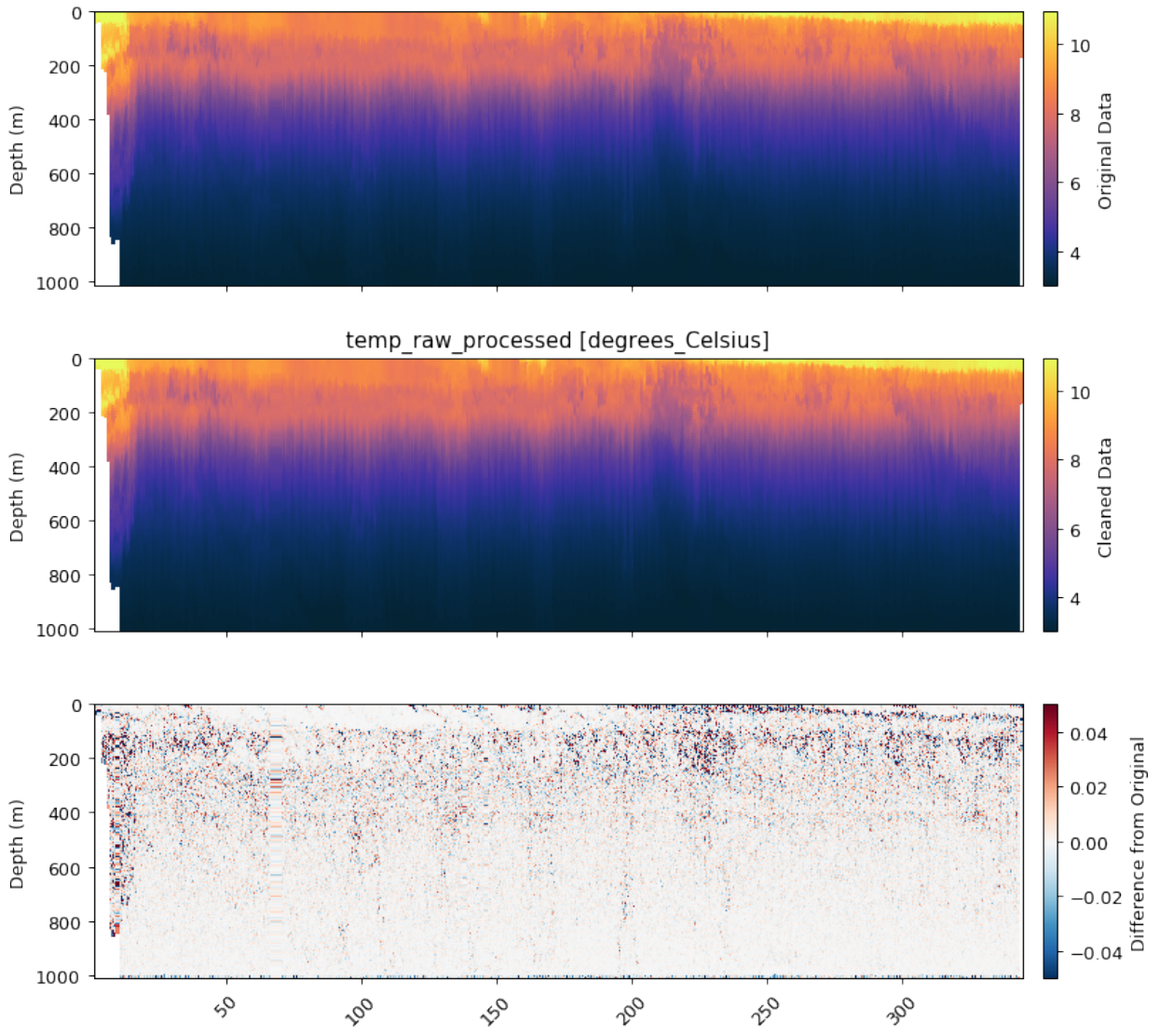
```
temp_qc = gt.calc_physics(temp, x, y,
                          iqr=1.5, depth_threshold=0,
                          spike_window=5, spike_method='median',
                          savitzky_golay_window=11, savitzky_golay_order=2)

# PLOTTING
fig, ax = plt.subplots(3, 1, figsize=[9, 8.5], sharex=True, dpi=90)

gt.plot(x, y, temp, cmap=cmo.thermal, ax=ax[0])
gt.plot(x, y, temp_qc, cmap=cmo.thermal, ax=ax[1])
gt.plot(x, y, temp_qc - temp, cmap=cm.RdBu_r, vmin=-0.05, vmax=0.05, ax=ax[2])

[a.set_xlabel('') for a in ax]
```

(continues on next page)

(continued from previous page)

```
ax[0].cb.set_label('Original Data')
ax[1].cb.set_label('Cleaned Data')
ax[2].cb.set_label('Difference from Original')

plt.show()
```

```
================================================
Physics Variable:
    Removing outliers with IQR * 1.5: 0 obs
    Removing spikes with rolling median (spike window=5)
    Smoothing with Savitzky-Golay filter (window=11, order=2)
```



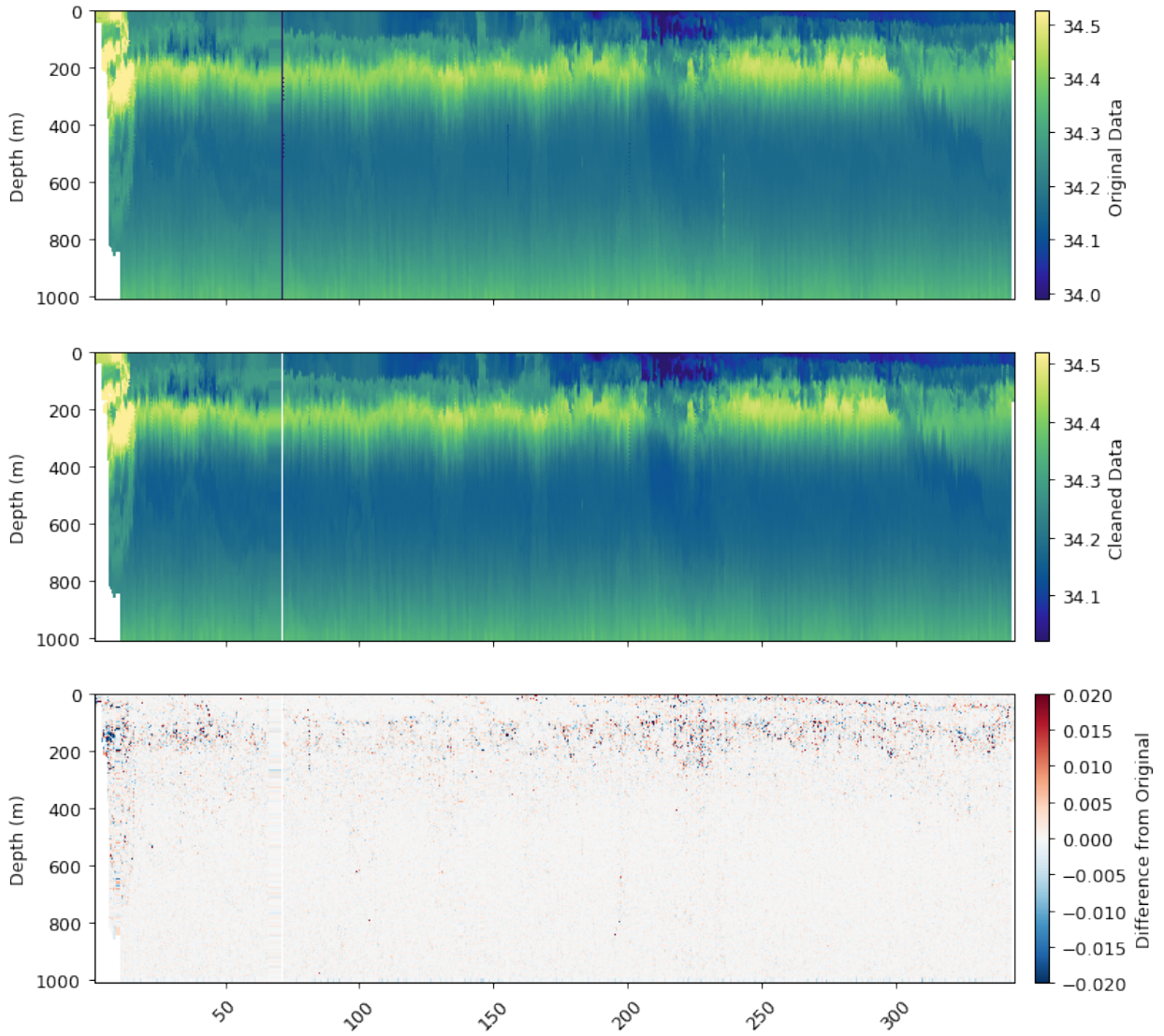

```
salt_qc = gt.calc_physics(salt, x, y,
                          mask_frac=0.2, iqr=2.5,
                          spike_window=5, spike_method='median',
```

(continues on next page)

(continued from previous page)

```
                            savitzky_golay_window=11, savitzky_golay_order=2)

# PLOTTING
fig, ax = plt.subplots(3, 1, figsize=[9, 8.5], sharex=True, dpi=90)

gt.plot(x, y, salt, cmap=cmo.haline, ax=ax[0])
gt.plot(x, y, salt_qc, cmap=cmo.haline, ax=ax[1])
gt.plot(x, y, salt_qc - salt, cmap=cm.RdBu_r, vmin=-0.02, vmax=0.02, ax=ax[2])

[a.set_xlabel('') for a in ax]

ax[0].cb.set_label('Original Data')
ax[1].cb.set_label('Cleaned Data')
ax[2].cb.set_label('Difference from Original')

plt.show()
```

```
================================================
Physics Variable:
    Removing outliers with IQR * 2.5: 1551 obs
    Removing spikes with rolling median (spike window=5)
    Removing horizontal outliers (fraction=0.2, multiplier=2.5)
    Smoothing with Savitzky-Golay filter (window=11, order=2)
```




```
dat['temp_qc'] = temp
dat['salt_qc'] = salt
```

CHAPTER

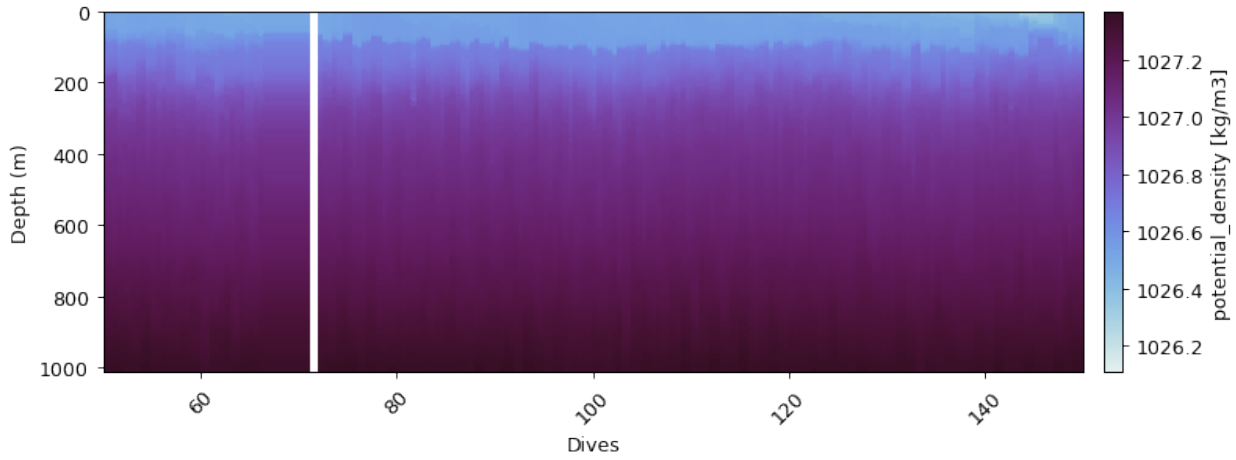
FIVE

SECONDARY PHYSICAL VARIABLES

5.1 Density

GliderTools provides a wrapper to calculate potential density. This is done by first calculating potential temperature and then calculating absolute salinity. A reference depth of `0` is used by default

```
dens0 = gt.physics.potential_density(salt_qc, temp_qc, pres, lats, lons)
dat['density'] = dens0
gt.plot(dat.dives, dat.depth, dens0, cmap=cmo.dense)
plt.xlim(50,150)
plt.show()
```



5.2 Mixed Layer Depth

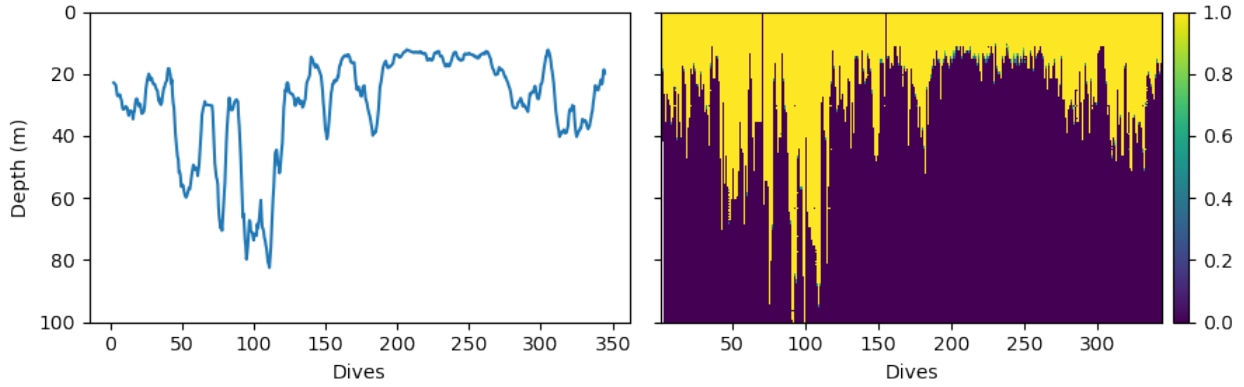
```
import matplotlib.pyplot as plt
mld = gt.physics.mixed_layer_depth(ds, 'density', verbose=False)
mld_smoothed = mld.rolling(10, min_periods=3).mean()

mld_mask = gt.utils.mask_below_depth(ds, mld)
mld_grid = gt.grid_data(ds.dives, ds.depth, mld_mask, verbose=False)

fig, ax = plt.subplots(1, 2, figsize=[9, 3], dpi=100, sharey=True)
```

(continues on next page)

(continued from previous page)

```
mld_smoothed.plot(ax=ax[0])
gt.plot(mld_grid, ax=ax[1])

[a.set_ylim(100, 0) for a in ax]

ax[0].set_ylabel('Depth (m)')
[a.set_xlabel('Dives') for a in ax]
plt.xticks(rotation=0)

fig.tight_layout()
```

```
/Users/luke/Git/GliderTools/glidertools/helpers.py:61: GliderToolsWarning:

Primary input variable is not xr.DataArray data type - no metadata to pass on.
```



CHAPTER

SIX

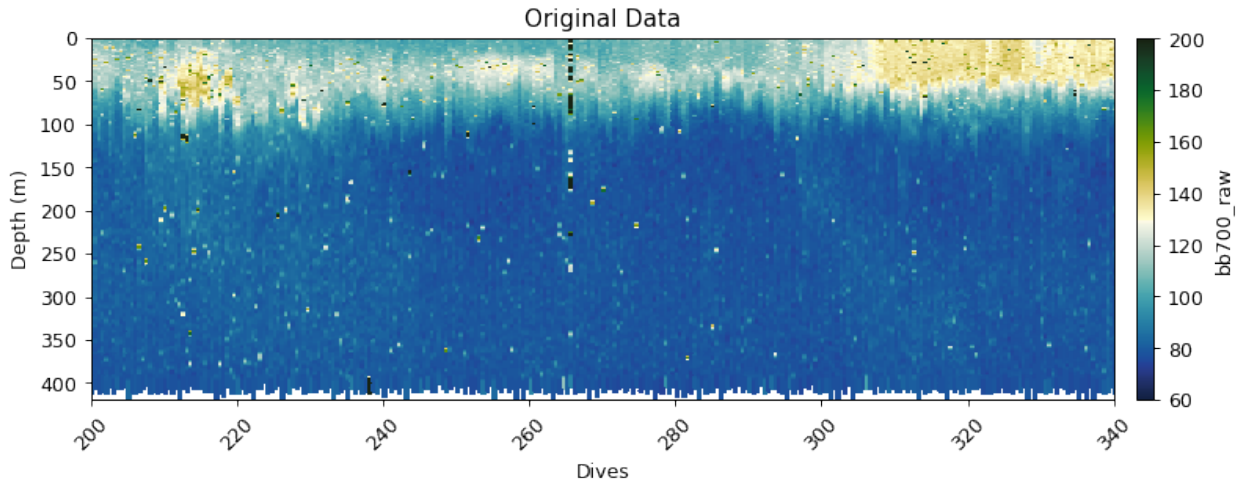
OPTICS (BB, PAR, CHL)

The optics module contains functions that process backscatter, PAR and fluorescence.

There is a wrapper function for each of these variables that applies several functions related to cleaning and processing. We show each step of the wrapper function seperately and then summarise with the wrapper function.

6.1 Backscatter

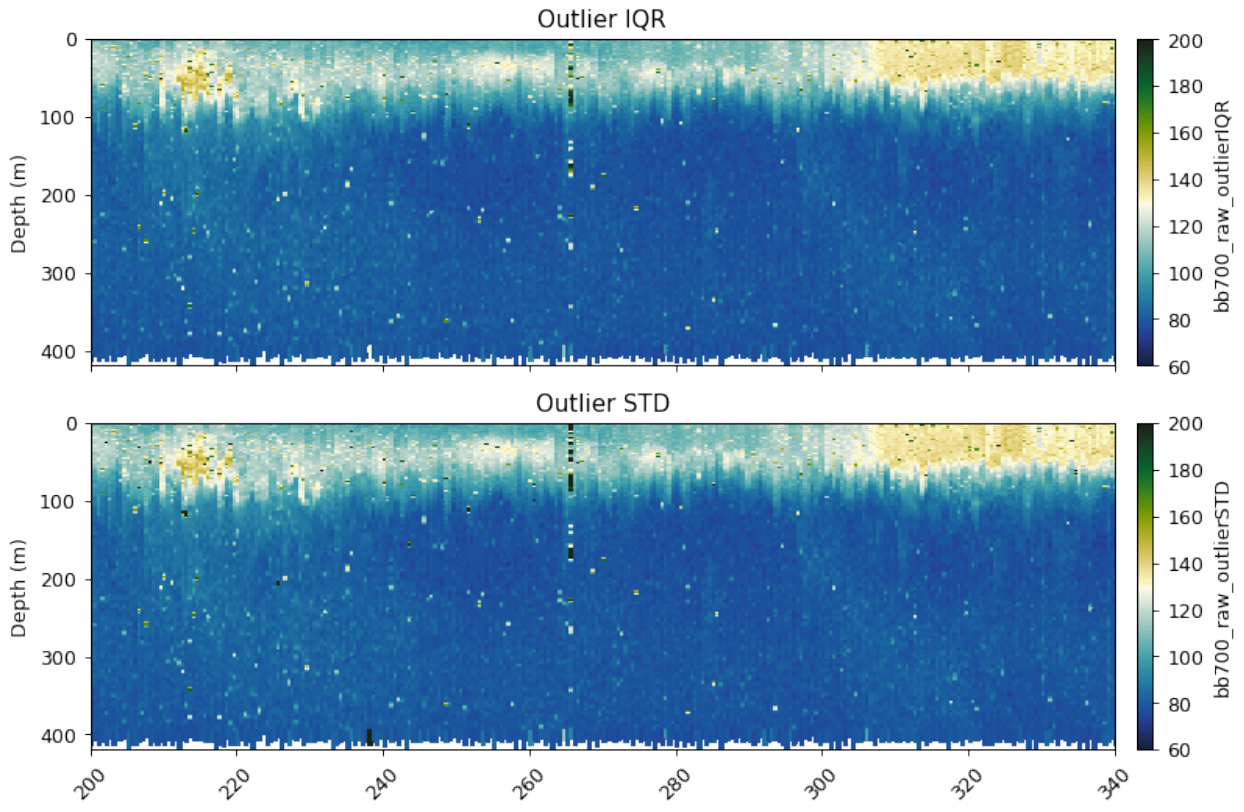
```
theta = 124
xfactor = 1.076

gt.plot(x, y, bb700, cmap=cmo.delta, vmin=60, vmax=200)
xlim(200,340)
title('Original Data')
show()
```



6.1.1 Outlier bounds method

See the cleaning section for more information on `gt.cleaning.outlider_bounds_[]`

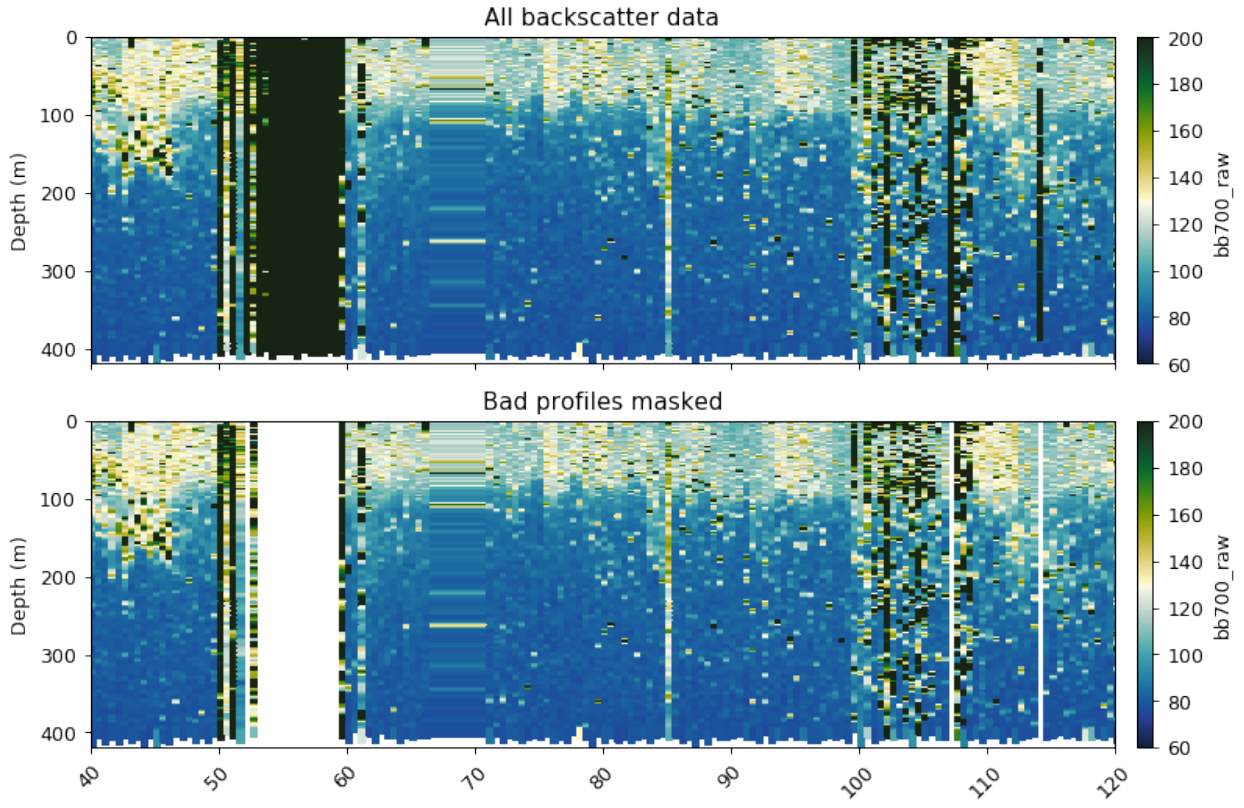
```
bb700_iqr = gt.cleaning.outlier_bounds_iqr(bb700, multiplier=3)
bb700_std = gt.cleaning.outlier_bounds_std(bb700, multiplier=3)

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, bb700_iqr, cmap=cmo.delta, ax=ax[0], vmin=60, vmax=200)
gt.plot(x, y, bb700_std, cmap=cmo.delta, ax=ax[1], vmin=60, vmax=200)

[a.set_xlabel('') for a in ax]
[a.set_xlim(200, 340) for a in ax]

ax[0].set_title('Outlier IQR')
ax[1].set_title('Outlier STD')

plt.show()
```



6.1.2 Removing bad profiles

This function masks bad dives based on mean + std x [1] or median + std x [1] at a reference depth.

```
# find_bad_profiles returns boolean mask and dive numbers
# we index only the mask
bad_profiles = gt.optics.find_bad_profiles(dives, depth, bb700,
                                           ref_depth=300,
                                           stdev_multiplier=1,
                                           method='median')[0]

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)
# ~ reverses True to False and vice versa - i.e. we mask bad bad profiles
gt.plot(x, y, bb700, cmap=cmo.delta, ax=ax[0], vmin=60, vmax=200)
gt.plot(x, y, bb700.where(~bad_profiles), cmap=cmo.delta, ax=ax[1], vmin=60, vmax=200)

[a.set_xlabel('') for a in ax]
[a.set_xlim(40, 120) for a in ax]

ax[0].set_title('All backscatter data')
ax[1].set_title('Bad profiles masked')

plt.show()
```

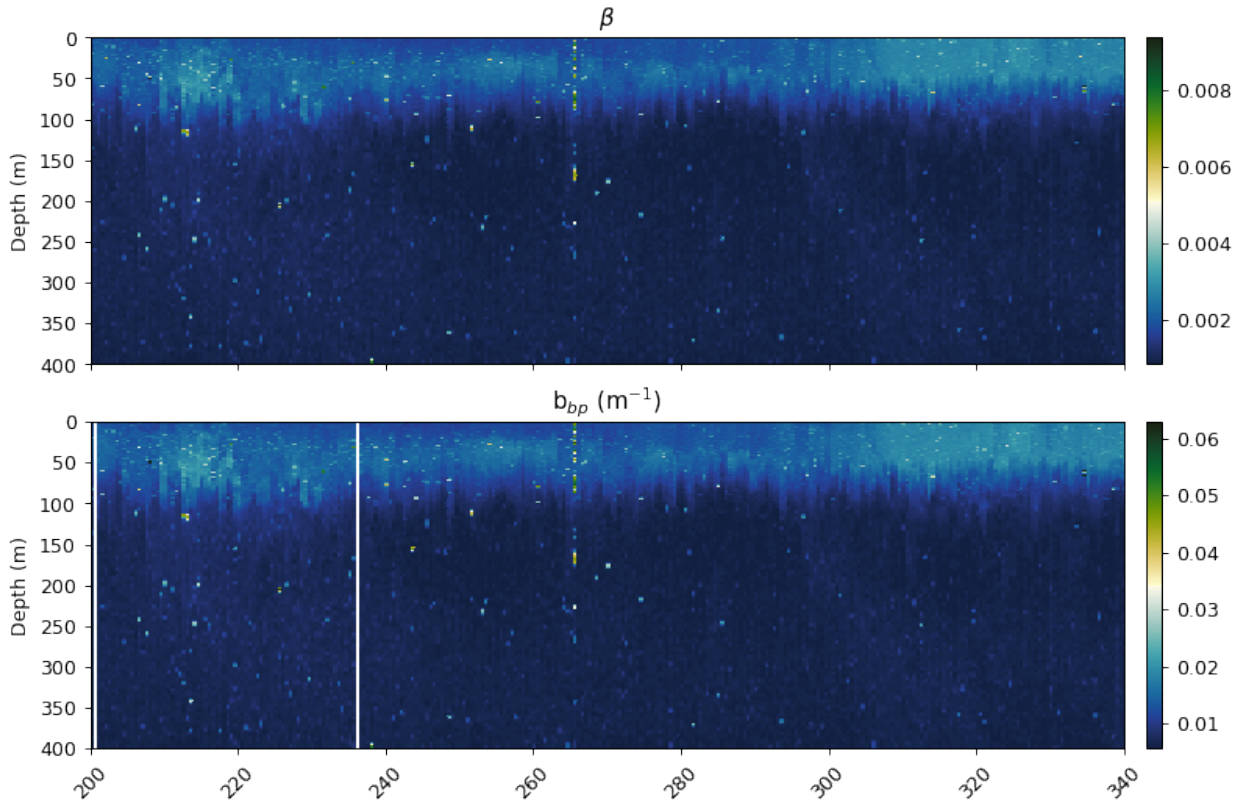


6.1.3 Conversion from counts to total backscatter

The scale and offset function uses the factory calibration dark count and scale factor.

The bback total function uses the coefficients from Zhang et al. (2009) to convert the raw counts into total backscatter (m-1), correcting for temperature and salinity. The χ factor and θ in this example were taken from Sullivan et al. (2013) and Slade & Boss (2015).

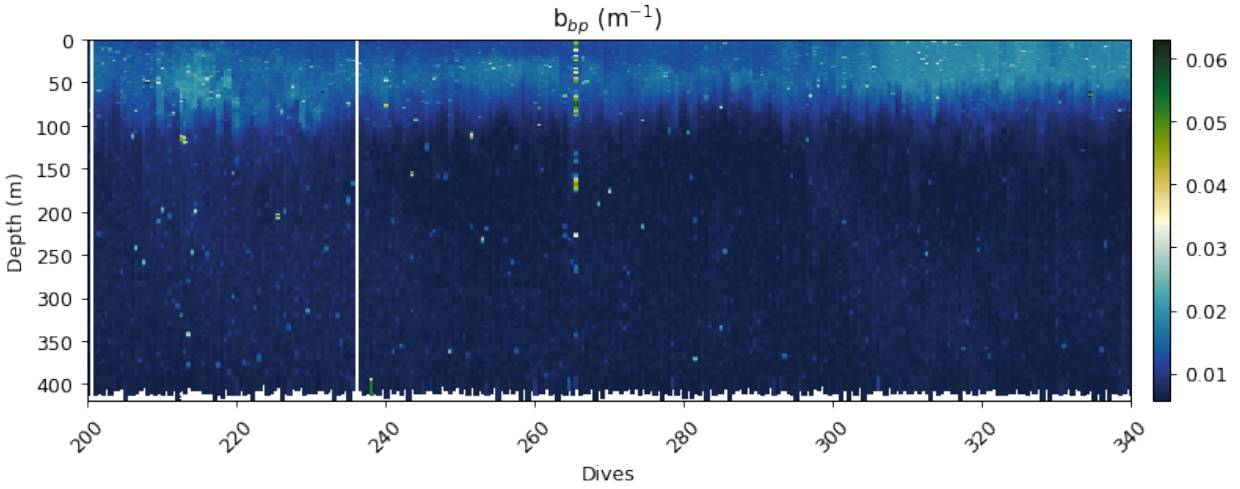
```
beta = gt.flo_functions.flo_scale_and_offset(bb700.where(~bad_profiles), 49, 3.217e-5)
bbp = gt.flo_functions.flo_bback_total(beta, temp_qc, salt_qc, theta, 700, xfactor)

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, beta, cmap=cmo.delta, ax=ax[0], robust=True)
gt.plot(x, y, bbp, cmap=cmo.delta, ax=ax[1], robust=True)

[a.set_xlabel('') for a in ax]
[a.set_xlim(200, 340) for a in ax]
[a.set_ylim(400, 0) for a in ax]

ax[0].set_title('$\u03B2$')
ax[1].set_title('b$_{bp}$ (m$^{-1}$)')

plt.show()
```



6.1.4 Correcting for an in situ dark count

Sensor drift from factory calibration requires an additional correction, the calculation of a dark count in situ. This is calculated from the 95th percentile of backscatter measurements between 200 and 400m.

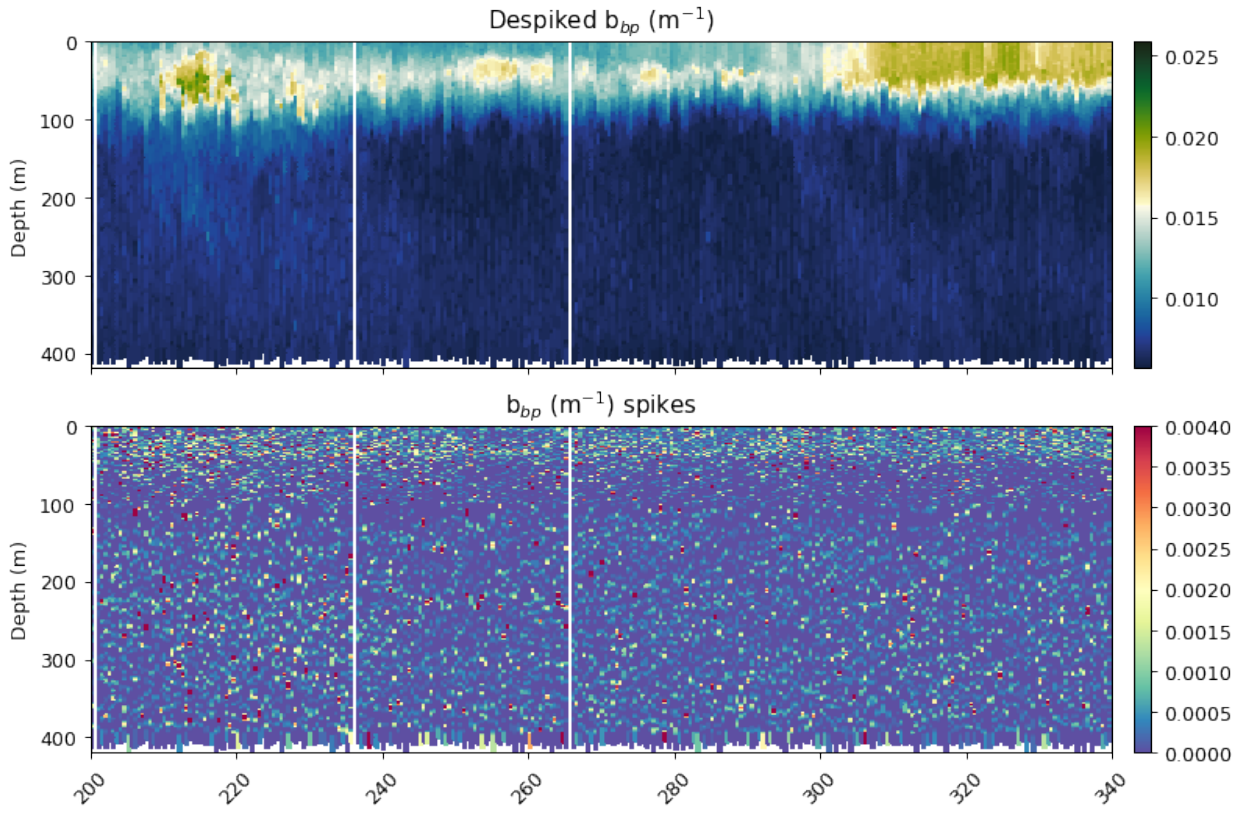
```
bbp = gt.optics.backscatter_dark_count(bbp, depth)

gt.plot(x, y, bbp, cmap=cmo.delta, robust=True)
xlim(200,340)
title('b$_{bp}$ (m$^{-1}$)')
show()
```



6.1.5 Despiking

Following the methods outlined in Briggs et al. (2011) to both identify spikes in backscatter and remove them from the baseline backscatter signal. The spikes are retained as the data can be used to address specific science questions, but their presence can decrease the accuracy of the fluorescence quenching function.

```
bbp_horz = gt.cleaning.horizontal_diff_outliers(x, y, bbp, depth_threshold=10, mask_
↪frac=0.05)
bbp_baseline, bbp_spikes = gt.cleaning.despike(bbp_horz, 7, spike_method='minmax')


fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, bbp_baseline, cmap=cmo.delta, ax=ax[0], robust=True)
gt.plot(x, y, bbp_spikes, ax=ax[1], cmap=cm.Spectral_r, vmin=0, vmax=0.004)

[a.set_xlabel('') for a in ax]
[a.set_xlim(200, 340) for a in ax]

ax[0].set_title('Despiked b$_{bp}$ (m$^{-1}$)')
ax[1].set_title('b$_{bp}$ (m$^{-1}$) spikes')

plt.show()
```



6.1.6 Adding the corrected variables to the original dataframe

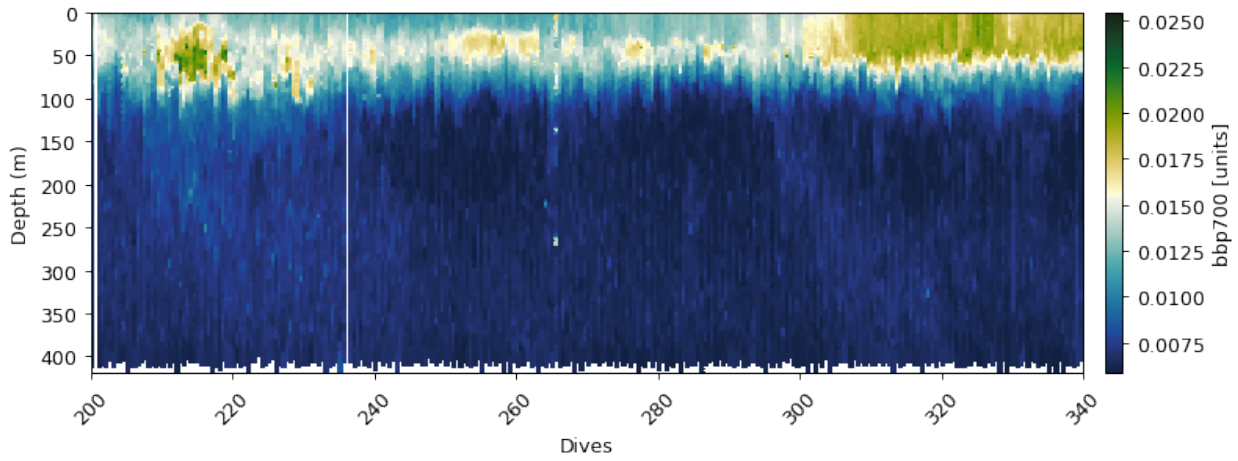
```
dat['bbp700'] = bbp_baseline
dat['bbp700_spikes'] = bbp_spikes
```

6.1.7 Wrapper function demonstration

A wrapper function was also designed, which is demonstrated below with the second wavelength (700 nm). The default option is for verbose to be True, which will provide an output of the different processing steps.

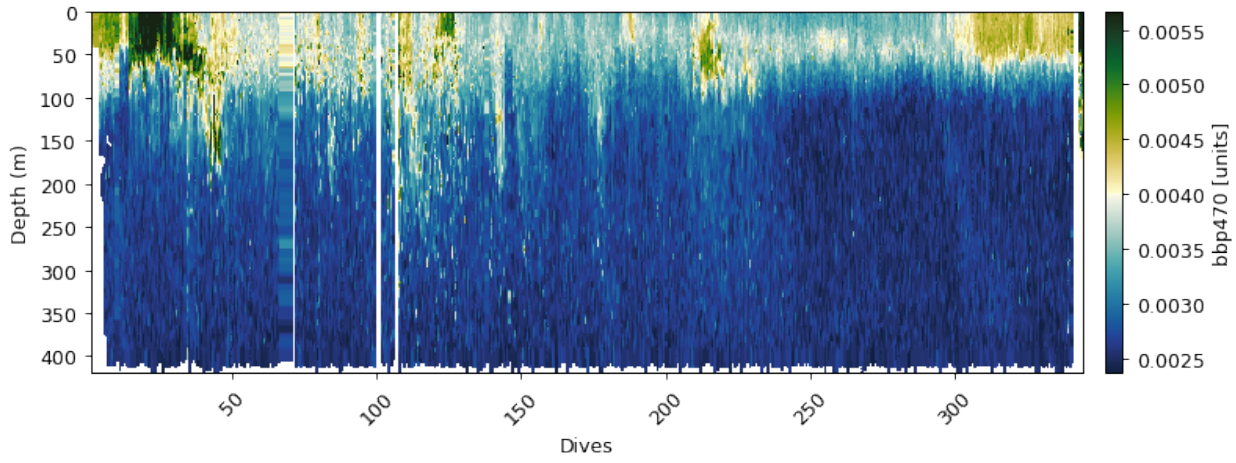
```
bbp_baseline, bbp_spikes = gt.calc_backscatter(
    bb700, temp_qc, salt_qc, dives, depth, 700, 49, 3.217e-5,
    spike_window=11, spike_method='minmax', iqr=2., profiles_ref_depth=300,
    deep_multiplier=1, deep_method='median', verbose=True)

dat['bbp700'] = bbp_baseline
dat['bbp700_spikes'] = bbp_spikes

ax = gt.plot(x, y, dat.bbp700, cmap=cmo.delta),

[a.set_xlim(200, 340) for a in ax]

plt.show()
```

```
================================================
bb700:
        Removing outliers with IQR * 2.0: 8606 obs
        Mask bad profiles based on deep values (depth=300m)
        Number of bad profiles = 27/672
        Zhang et al. (2009) correction
        Dark count correction
        Spike identification (spike window=11)
```




```
bbp_baseline, bbp_spikes = gt.calc_backscatter(
    bb470, temp_qc, salt_qc, dives, depth, 470, 47, 1.569e-5,
    spike_window=7, spike_method='minmax', iqr=3, profiles_ref_depth=300,
    deep_multiplier=1, deep_method='median', verbose=True)

dat['bbp470'] = bbp_baseline
dat['bbp470_spikes'] = bbp_spikes

gt.plot(x, y, dat.bbp470, cmap=cmo.delta)
plt.show()
```

```
================================================
bb470:
        Removing outliers with IQR * 3: 2474 obs
        Mask bad profiles based on deep values (depth=300m)
        Number of bad profiles = 16/672
        Zhang et al. (2009) correction
        Dark count correction
        Spike identification (spike window=7)
```

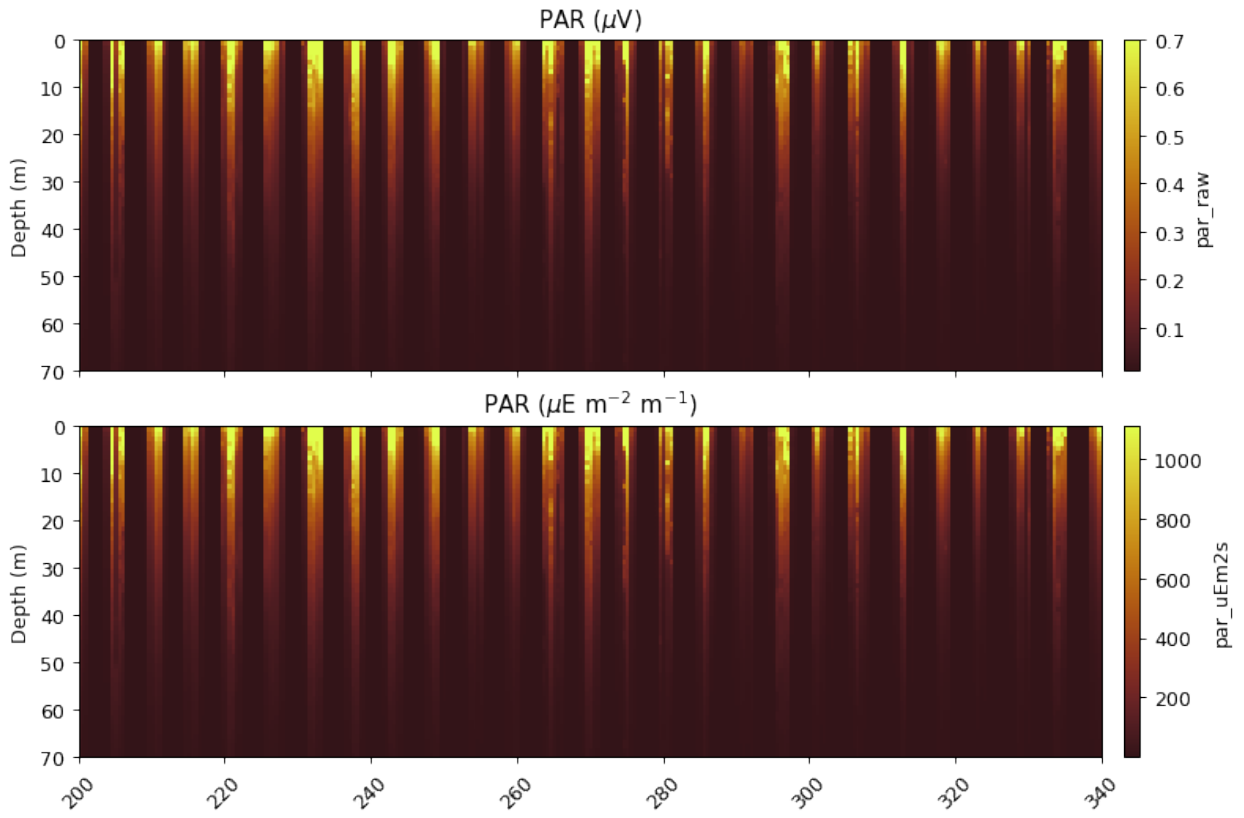


6.2 PAR

6.2.1 PAR Scaling

This function uses the factory calibration to convert from μV to μE m^{-2} s^{-1}.

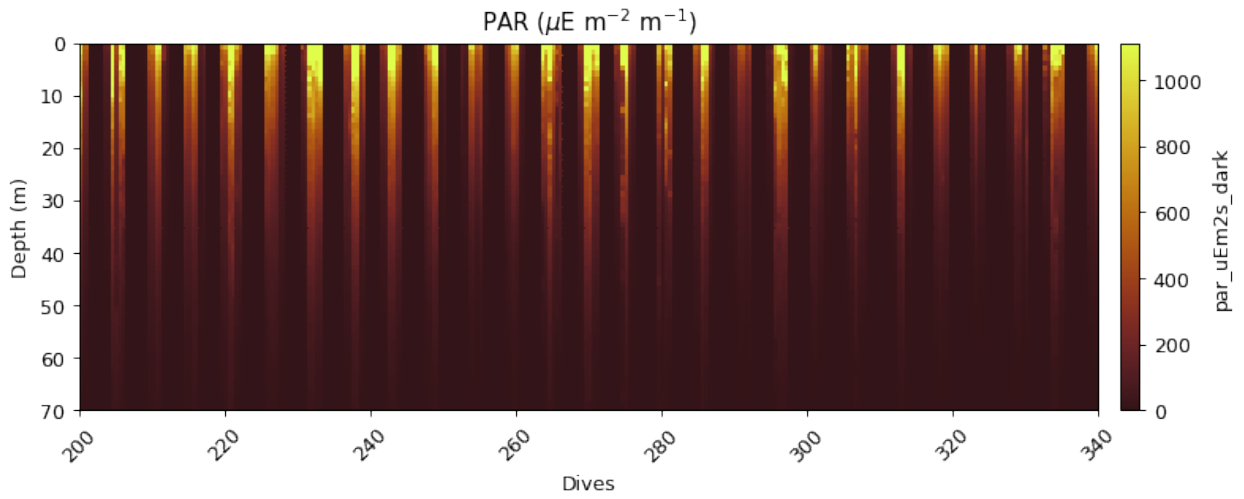
```
par_scaled = gt.optics.par_scaling(par, 6.202e-4, 10.8)

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, par, cmap=cmo.solar, ax=ax[0], robust=True)
gt.plot(x, y, par_scaled, cmap=cmo.solar, ax=ax[1], robust=True)

[a.set_xlabel('') for a in ax]
[a.set_xlim(200, 340) for a in ax]
[a.set_ylim(70, 0) for a in ax]

ax[0].set_title('PAR ($\mu$V)')
ax[1].set_title('PAR ($\mu$E m$^{-2}$ m$^{-1}$)')

plt.show()
```



6.2.2 Correcting for an in situ dark count

Sensor drift from factory calibration requires an additional correction, the calculation of a dark count in situ. This is calculated from the median of PAR measurements, with additional masking applied for values before 23:01 and outside the 90th percentile.

```
par_dark = gt.optics.par_dark_count(par_scaled, dives, depth, time)

gt.plot(x, y, par_dark, robust=True, cmap=cmo.solar)
xlim(200,340)
ylim(70,0)
title('PAR ($\mu$E m$^{-2}$ m$^{-1}$)')
show()
```

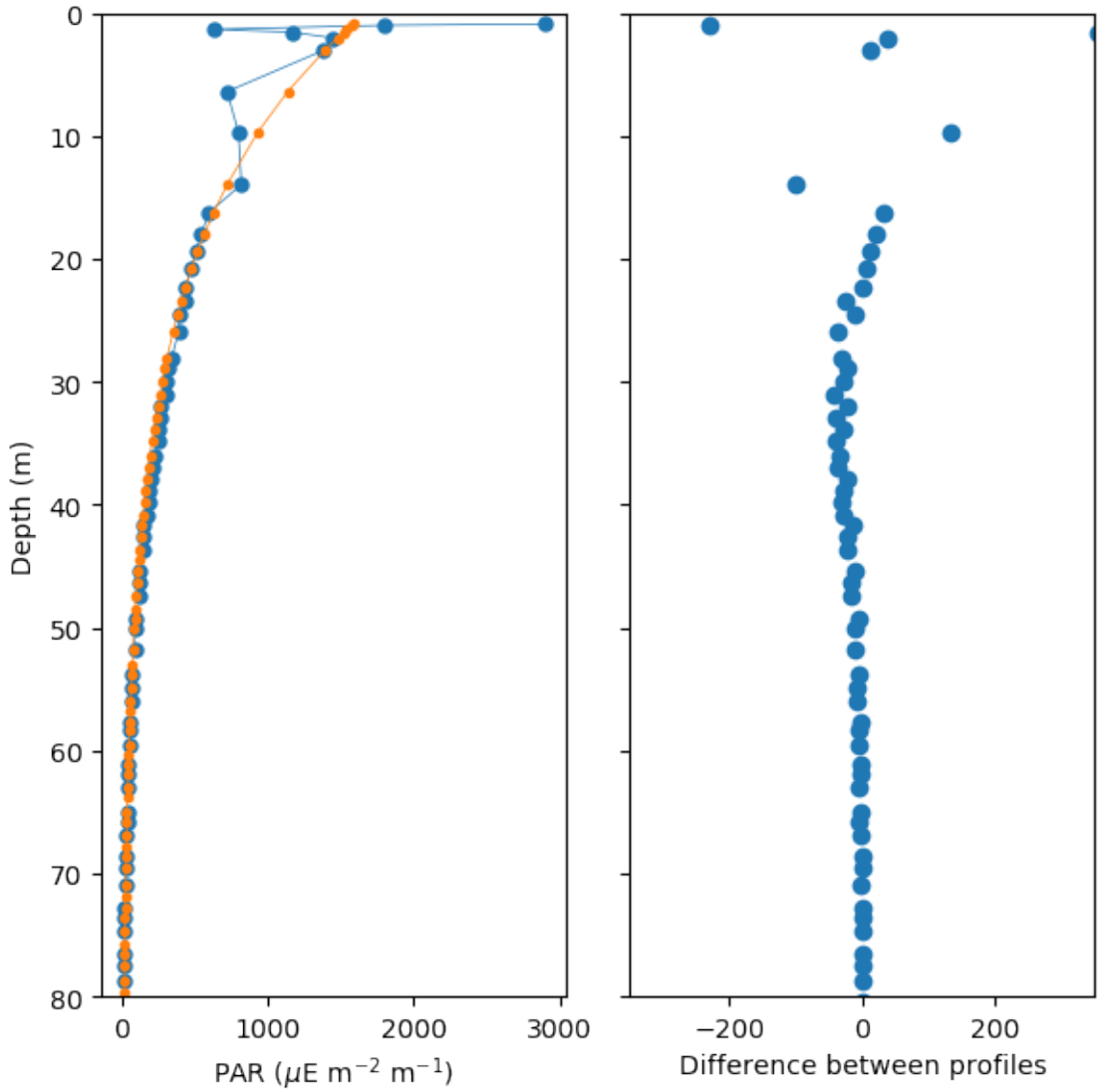


6.2.3 PAR replacement

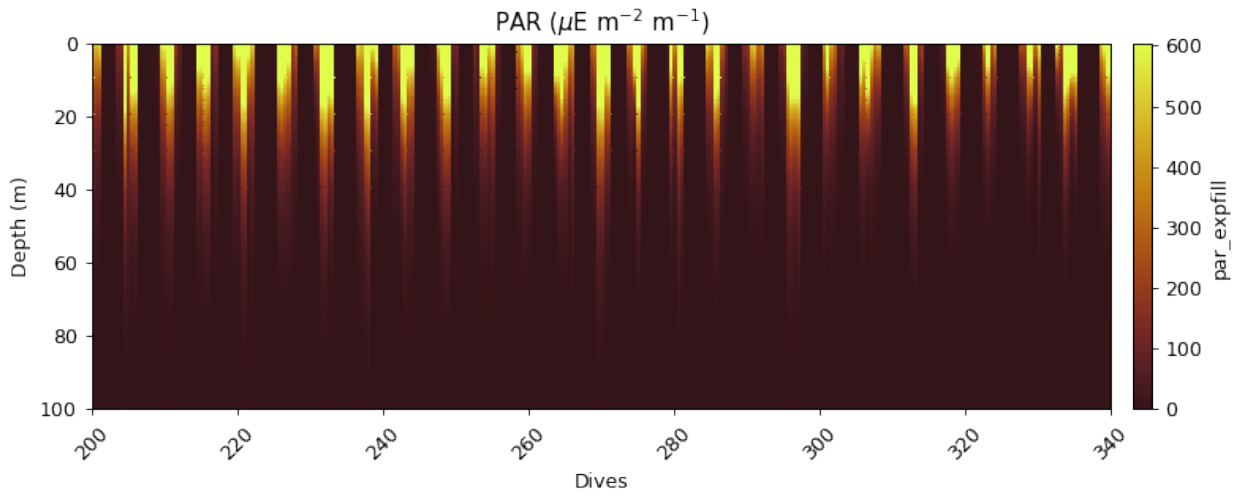
This function removes the top 5 metres from each dive profile, and then algebraically recalculates the surface PAR using an exponential equation.

```
par_filled = gt.optics.par_fill_surface(par_dark, dives, depth, max_curve_depth=80)
par_filled[par_filled < 0] = 0
par_filled = par_filled.fillna(0)
```

```
i = dives == 232

fig, ax = subplots(1, 2, figsize=[6,6], dpi=100)

ax[0].plot(par_dark[i], depth[i], lw=0.5, marker='o', ms=5)
ax[0].plot(par_filled[i], depth[i], lw=0.5, marker='o', ms=3)
ax[1].plot(par_filled[i] - par_dark[i], depth[i], lw=0, marker='o')

ax[0].set_ylim(80,0)
ax[0].set_ylabel('Depth (m)')
ax[0].set_xlabel('PAR ($\mu$E m$^{-2}$ m$^{-1}$)')

ax[1].set_ylim(80,0)
ax[1].set_xlim(-350,350)
ax[1].set_yticklabels('')
ax[1].set_xlabel('Difference between profiles')

fig.tight_layout()
plt.show()
```



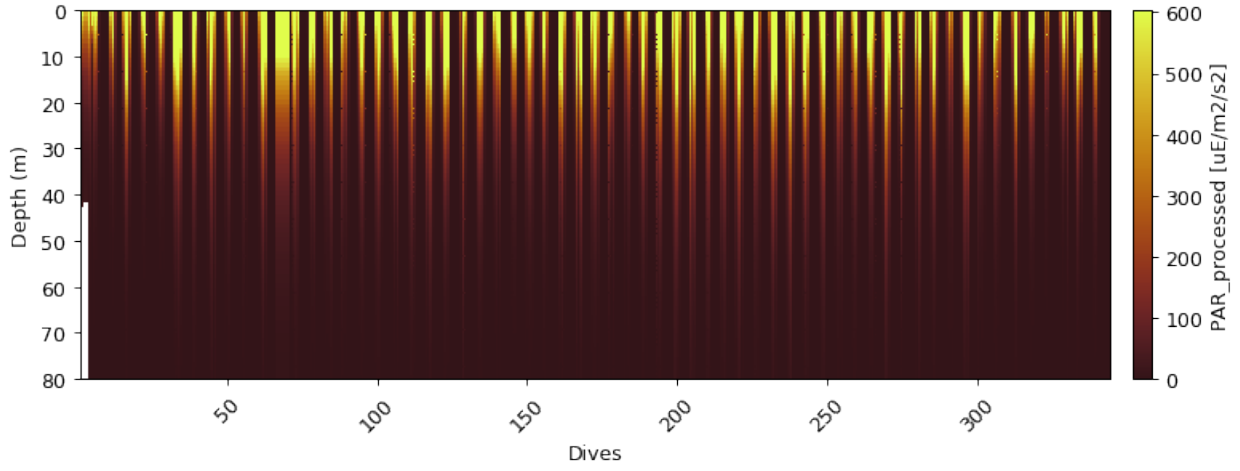

```
gt.plot(x, y, par_filled, robust=True, cmap=cmo.solar)
xlim(200,340)
ylim(100,0)
title('PAR ($\mu$E m$^{-2}$ m$^{-1}$)')
show()
```



6.2.4 Wrapper function demonstration

```
par_qc = gt.calc_par(par, dives, depth, time,
                     6.202e-4, 10.8,
                     curve_max_depth=80,
                     verbose=True).fillna(0)

gt.plot(x, y, par_qc, robust=True, cmap=cmo.solar)
ylim(80, 0)
show()
```

```
================================================
PAR
        Dark correction
        Fitting exponential curve to data
```

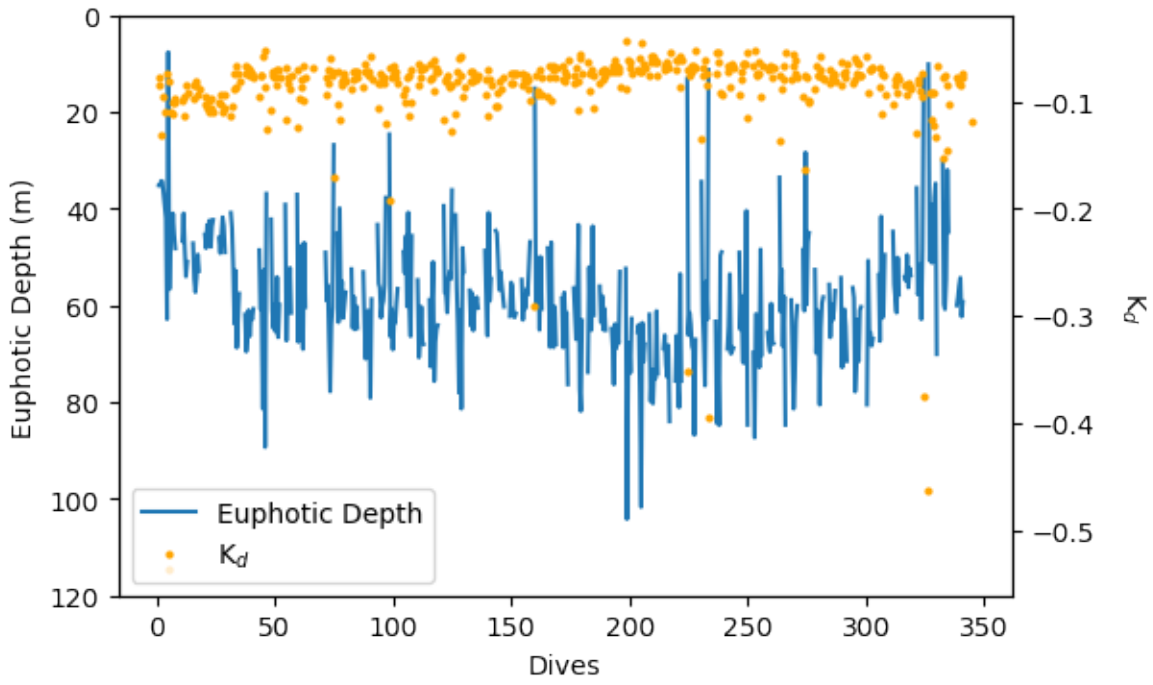


6.2.5 Deriving additional variables

Euphotic Depth and Light attenuation coefficient

```
euphotic_depth, kd = gt.optics.photic_depth(
    par_filled, dives, depth,
    return_mask=False,
    ref_percentage=1
)
```

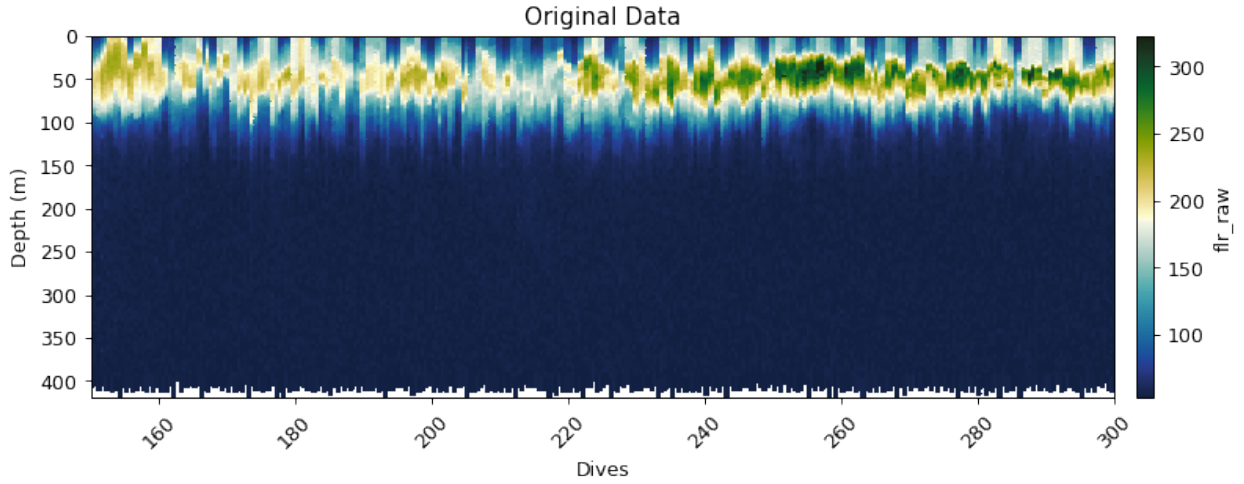
```
fig, ax = subplots(1, 1, figsize=[6,4], dpi=100)
p1 = plot(euphotic_depth.index, euphotic_depth, label='Euphotic Depth')
ylim(120,0)
ylabel('Euphotic Depth (m)')
xlabel('Dives')
ax2 = ax.twinx()
p2 = plot(kd.index, kd, color='orange', lw=0, marker='o', ms=2, label='K$_d$')
ylabel('K$_d$', rotation=270, labelpad=20)

lns = p1+p2
labs = [l.get_label() for l in lns]
ax2.legend(lns, labs, loc=3, numpoints=1)

show()
```



6.3 Fluorescence

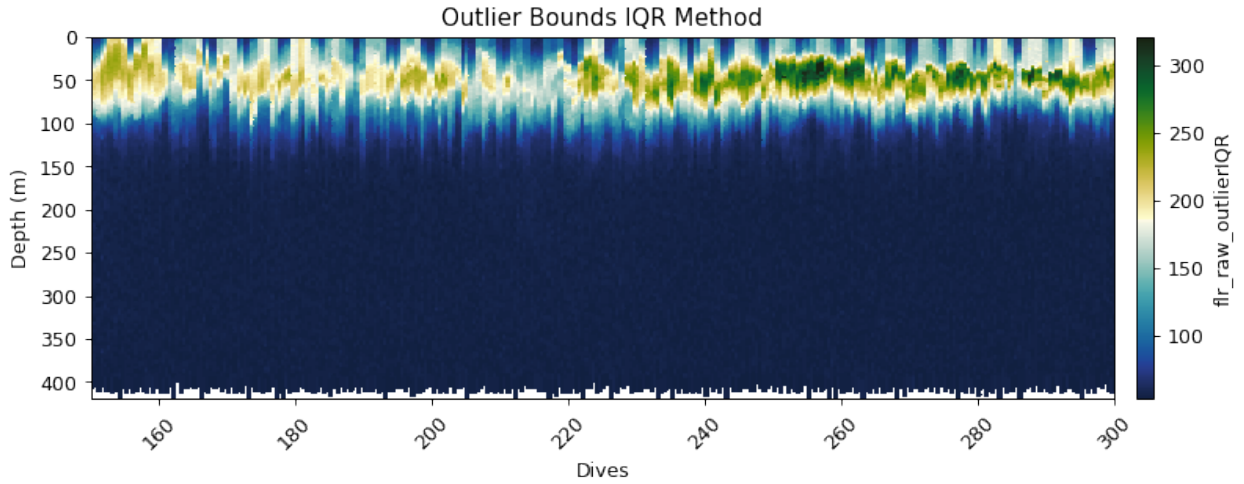
Quenching Correcting Method as outlined in Thomalla et al. (2017)

```
gt.plot(x, y, fluor, cmap=cmo.delta, robust=True)
xlim(150,300)
title('Original Data')
show()
```



6.3.1 Outlier bounds method

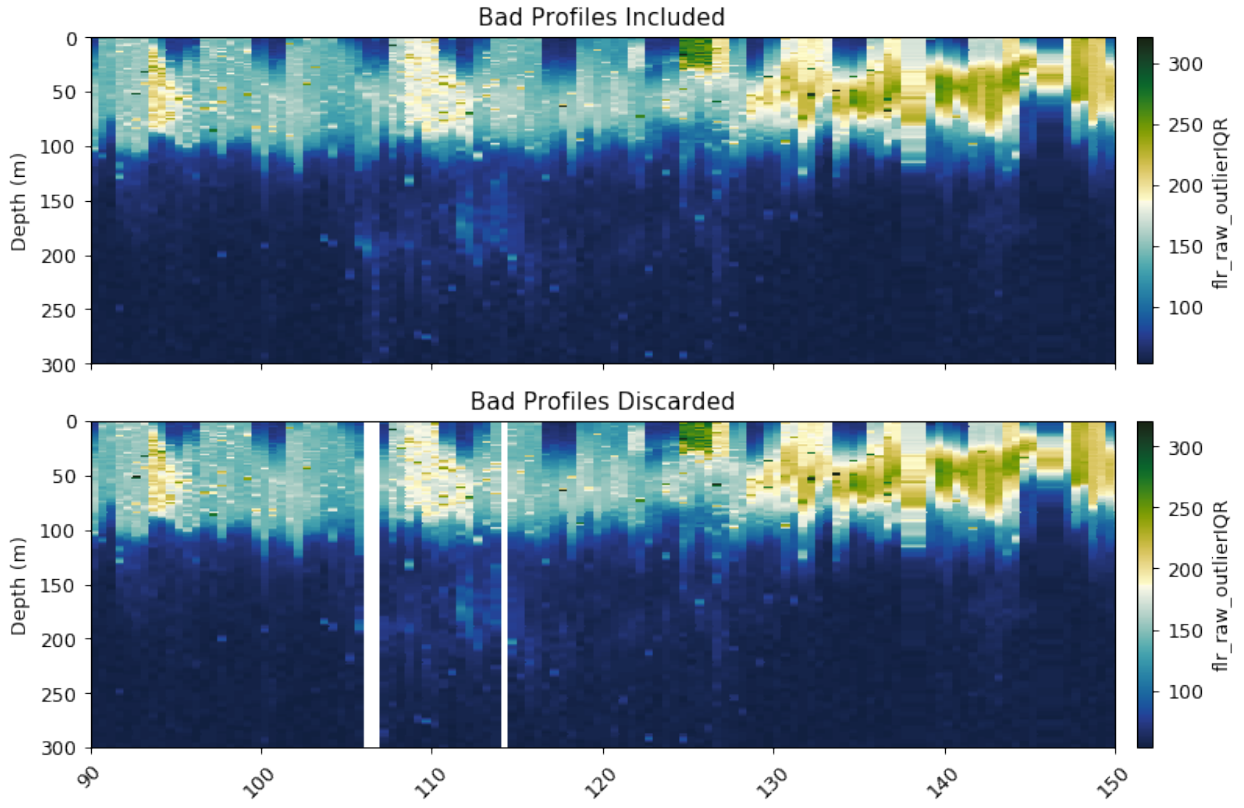
```
flr_iqr = gt.cleaning.outlier_bounds_iqr(fluor, multiplier=3)

gt.plot(x, y, flr_iqr, cmap=cmo.delta, robust=True)
title('Outlier Bounds IQR Method')
xlim(150,300)
show()
```



6.3.2 Removing bad profiles

This function masks bad dives based on mean + std x [3] or median + std x [3] at a reference depth.

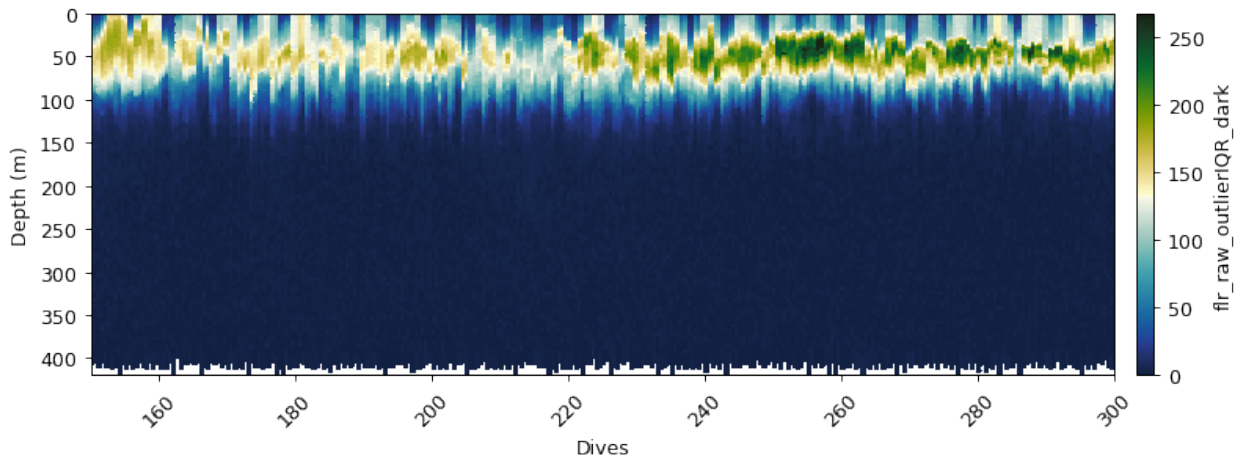
```
bad_profiles = gt.optics.find_bad_profiles(dives, depth, flr_iqr,
                                           ref_depth=300,
                                           stdev_multiplier=4,
                                           method='mean')
flr_goodprof = flr_iqr.where(~bad_profiles[0])

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, flr_iqr, cmap=cmo.delta, ax=ax[0], robust=True)
gt.plot(x, y, flr_goodprof, cmap=cmo.delta, ax=ax[1], robust=True)

[a.set_xlabel('') for a in ax]
[a.set_xlim(90, 150) for a in ax]
[a.set_ylim(300, 0) for a in ax]

ax[0].set_title('Bad Profiles Included')
ax[1].set_title('Bad Profiles Discarded')

plt.show()
```



6.3.3 Correcting for an in situ dark count

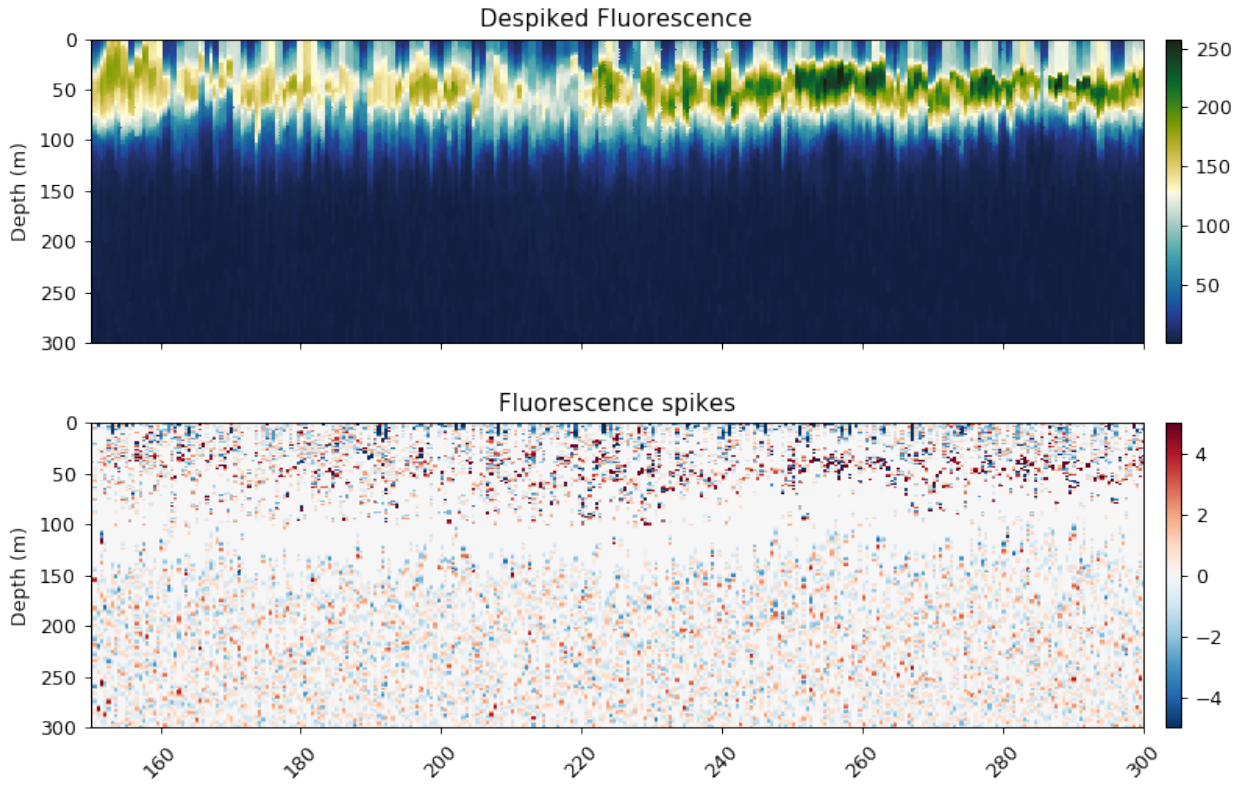
Sensor drift from factory calibration requires an additional correction, the calculation of a dark count in situ. This is calculated from the 95th percentile of fluorescence measurements between 300 and 400m.

```
flr_dark = gt.optics.fluorescence_dark_count(flr_iqr, dat.depth)

gt.plot(x, y, flr_dark, cmap=cmo.delta, robust=True)
xlim(150,300)
show()
```



6.3.4 Despiking

```
flr_base, flr_spikes = gt.cleaning.despike(flr_dark, 11, spike_method='median')

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, flr_base, cmap=cmo.delta, ax=ax[0], robust=True)
gt.plot(x, y, flr_spikes, cmap=cm.RdBu_r, ax=ax[1], vmin=-5, vmax=5)

[a.set_xlabel('') for a in ax]
[a.set_xlim(150, 300) for a in ax]
[a.set_ylim(300, 0) for a in ax]

ax[0].set_title('Despiked Fluorescence')
ax[1].set_title('Fluorescence spikes')

plt.show()
```



6.3.5 Quenching Correction

This function uses the method outlined in Thomalla et al. (2017), briefly it calculates the quenching depth and performs the quenching correction based on the fluorescence to backscatter ratio. The quenching depth is calculated based upon the different between night and daytime fluorescence.

The default setting is for the preceding night to be used to correct the following day's quenching (`night_day_group=True`). This can be changed so that the following night is used to correct the preceding day. The quenching depth is then found from the difference between the night and daytime fluorescence, using the steepest gradient of the {5 minimum differences and the points the difference changes sign (+ve/-ve)}.

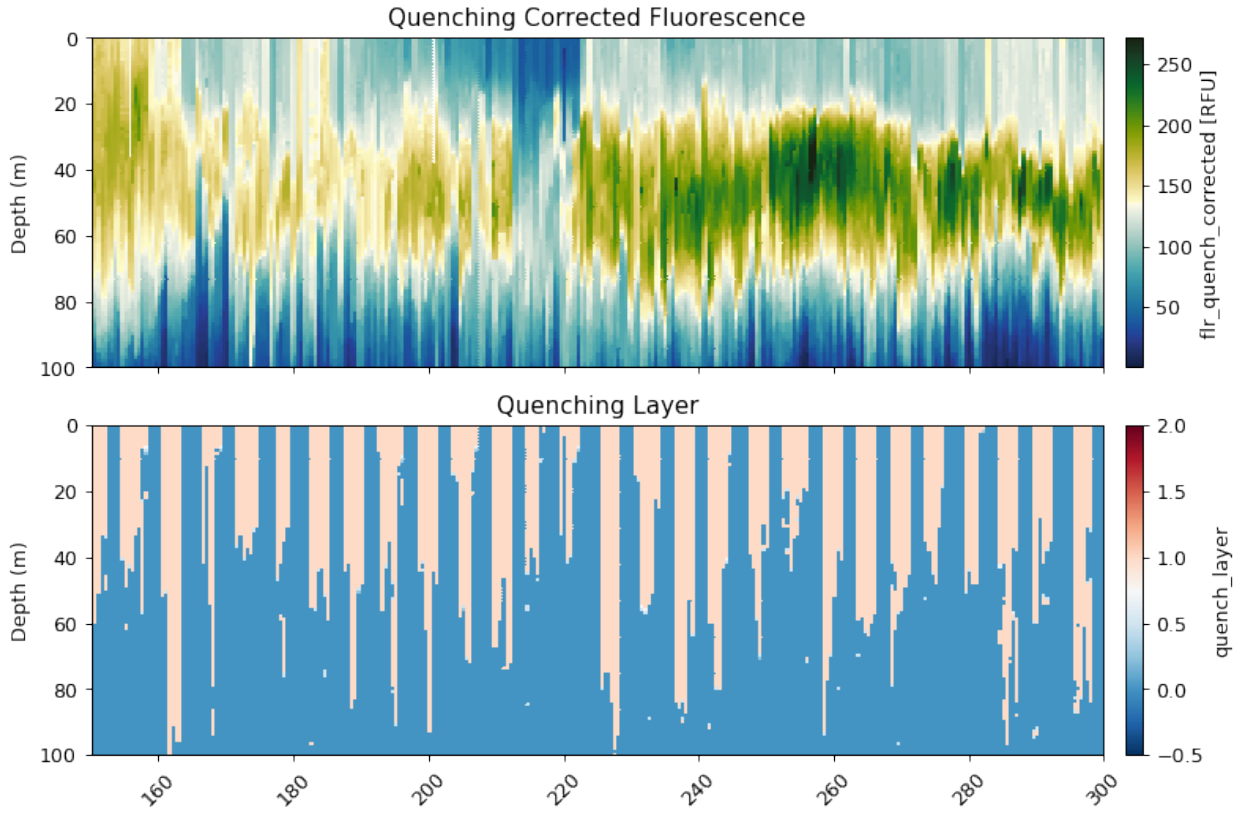
The function gets the backscatter/fluorescence ratio between from the quenching depth to the surface, and then calculates a mean nighttime ratio for each night. The quenching ratio is calculated from the nighttime ratio and the daytime ratio, which is then applied to fluorescence to correct for quenching. If the corrected value is less than raw, then the function will return the original raw data.

```
flr_qc, quench_layer = gt.optics.quenching_correction(
    flr_base, dat.bbp470, dives, depth, time, lats, lons,
    sunrise_sunset_offset=1, night_day_group=True)

fig, ax = plt.subplots(2, 1, figsize=[9, 6], sharex=True, dpi=90)

gt.plot(x, y, flr_qc, cmap=cmo.delta, ax=ax[0], robust=True)
gt.plot(x, y, quench_layer, cmap=cm.RdBu_r, ax=ax[1], vmin=-.5, vmax=2)

[a.set_xlabel('') for a in ax]
[a.set_xlim(150, 300) for a in ax]
```

(continues on next page)

(continued from previous page)

```
[a.set_ylim(100, 0) for a in ax]

ax[0].set_title('Quenching Corrected Fluorescence')
ax[1].set_title('Quenching Layer')

plt.show()
```



6.3.6 Wrapper function

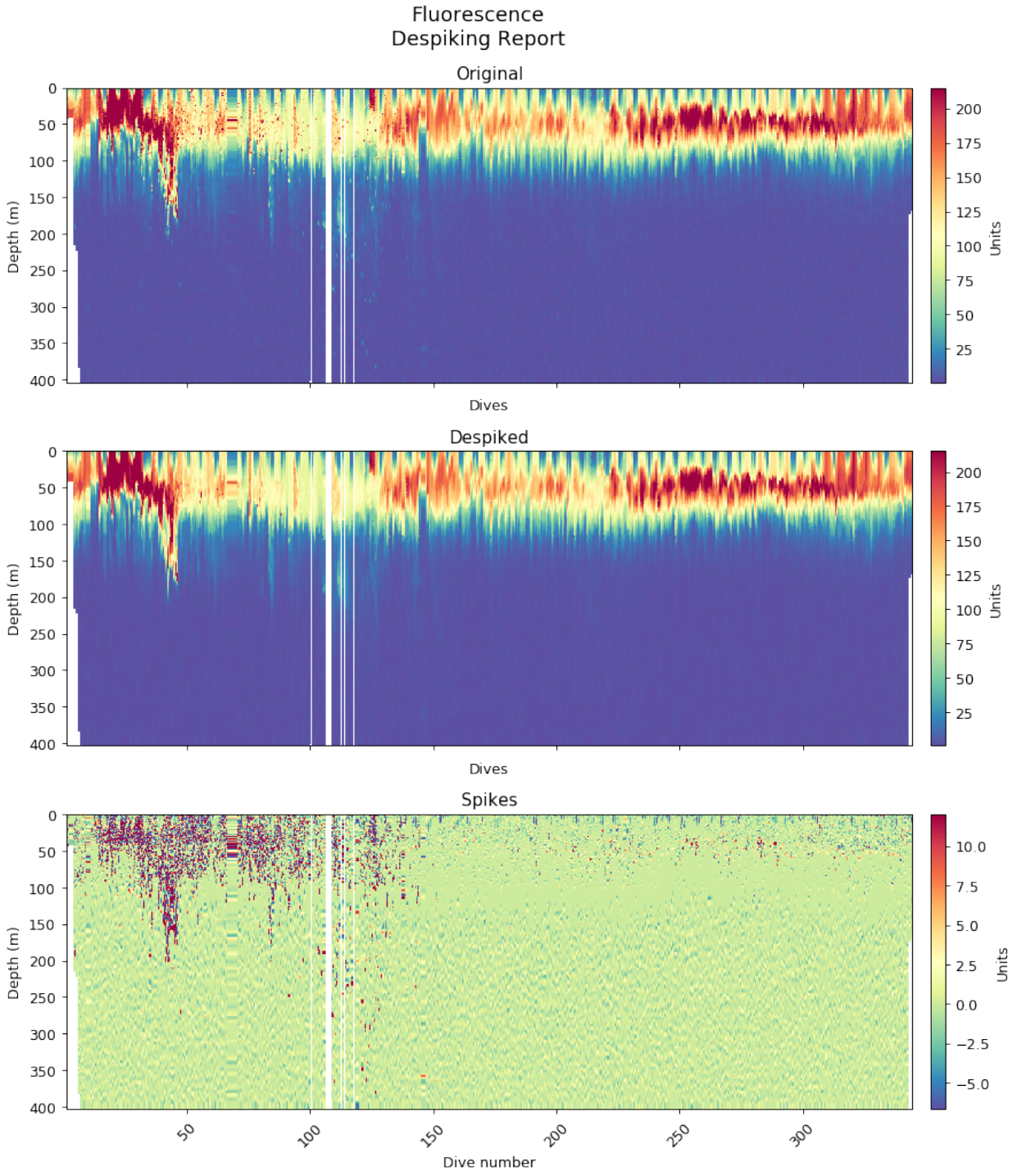
```
flr_qnch, flr, qnch_layer, [fig1, fig2] = gt.calc_fluorescence(
    fluor, dat.bbp700, dives, depth, time, lats, lons, 53, 0.0121,
    profiles_ref_depth=300, deep_method='mean', deep_multiplier=1,
    spike_window=11, spike_method='median', return_figure=True,
    night_day_group=False, sunrise_sunset_offset=2, verbose=True)

dat['flr_qc'] = flr
```

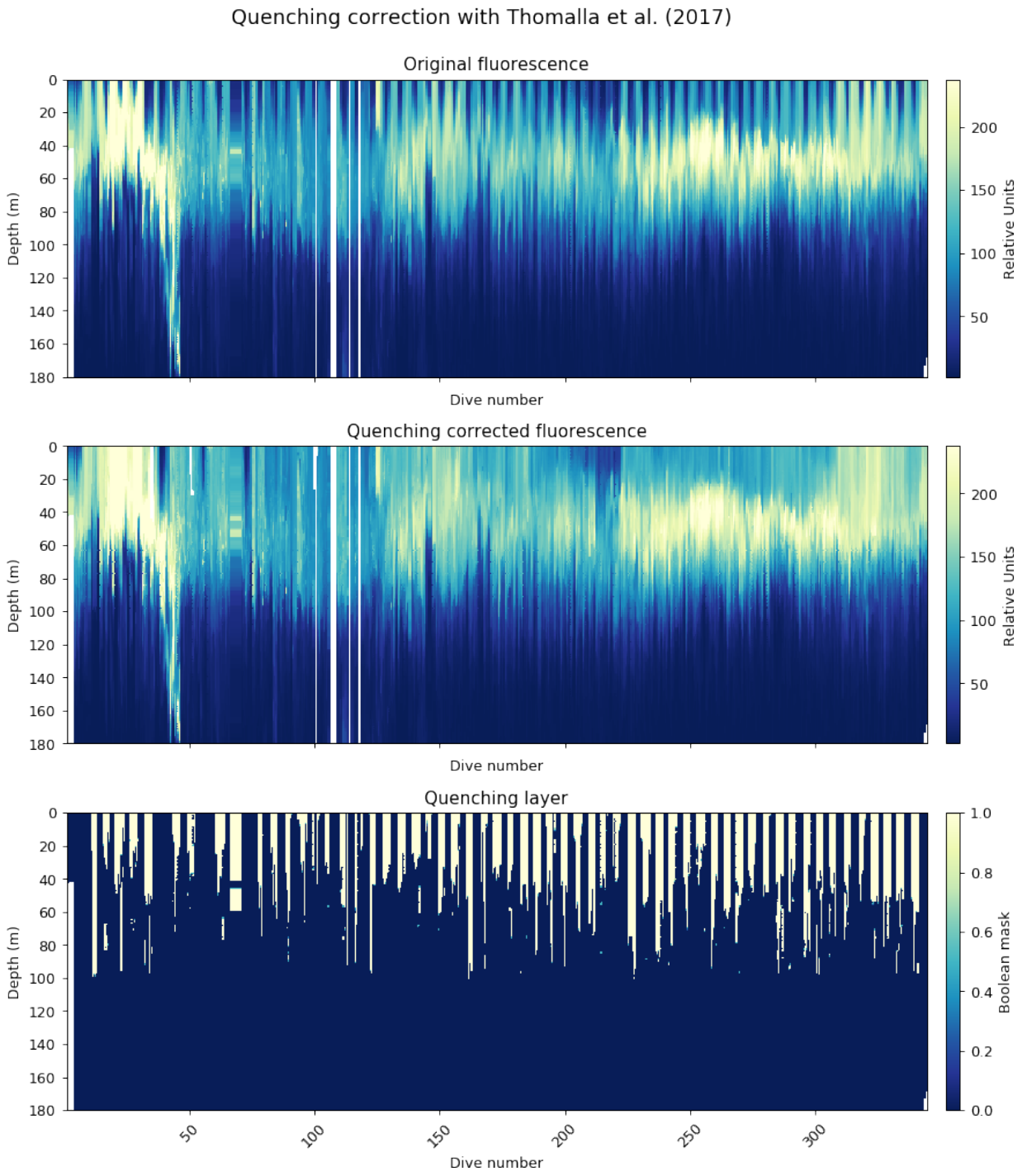
```
=================================================
Fluorescence
        Mask bad profiles based on deep values (ref depth=300m)
        Number of bad profiles = 19/672
        Dark count correction
        Quenching correction
```

(continues on next page)

(continued from previous page)

```
        Spike identification (spike window=11)
        Generating figures for despiking and quenching report
```




Quenching correction with Thomalla et al. (2017)
Original fluorescence
Depth (m)
Dive number
Relative Units
Quenching corrected fluorescence
Depth (m)
Dive number
Relative Units
Quenching layer
Depth (m)
Dive number
Boolean mask

CHAPTER

SEVEN

CALIBRATION WITH BOTTLE SAMPLES

Bottle calibration can also be done using the `calibration` module.

The bottle file needs to be in a specific format with dates (`datetime64` format), depth and the variable values. This can be imported with any method available. I recommend `pandas.read_csv` as shown in the example below. Note that latitude and longitude are not taken into account, thus the user needs to make sure that the CTD cast was in the correct location (and time, but this will be used to match the glider).

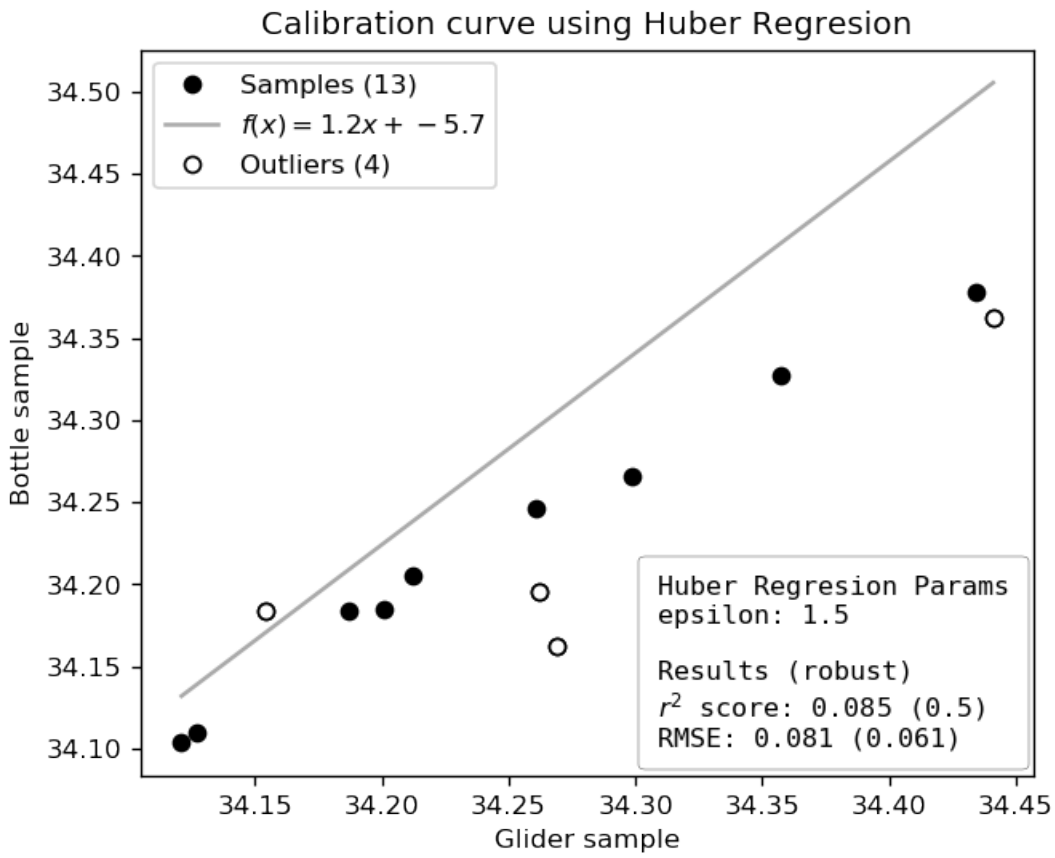
```
import pandas as pd

fname = '/Users/luke/Work/Publications/2019_Gregor_Front_glider/figures/SOSCEX 3 PS1.csv'
cal = pd.read_csv(fname, parse_dates=['datetime'], dayfirst=True)
```

The `calibration.bottle_matchup` function returns an array that matches the size of the ungridded glider data. The matching is done based on depth and time from both the glider and the CTD. The function will show how many samples have been matched and the smallest time difference between a CTD rosette cast and a dive (any time on the dive).

7.1 Using depth

```
%autoreload 2

dat['bottle_sal'] = gt.calibration.bottle_matchup(
    dat.dives, dat.depth, dat.time,
    cal.depth, cal.datetime, cal.sal)

model = gt.calibration.robust_linear_fit(dat.salt_qc, dat.bottle_sal, fit_intercept=True,
↪ epsilon=1.5)
dat['salinity_qc'] = model.predict(dat.salt_qc)
```

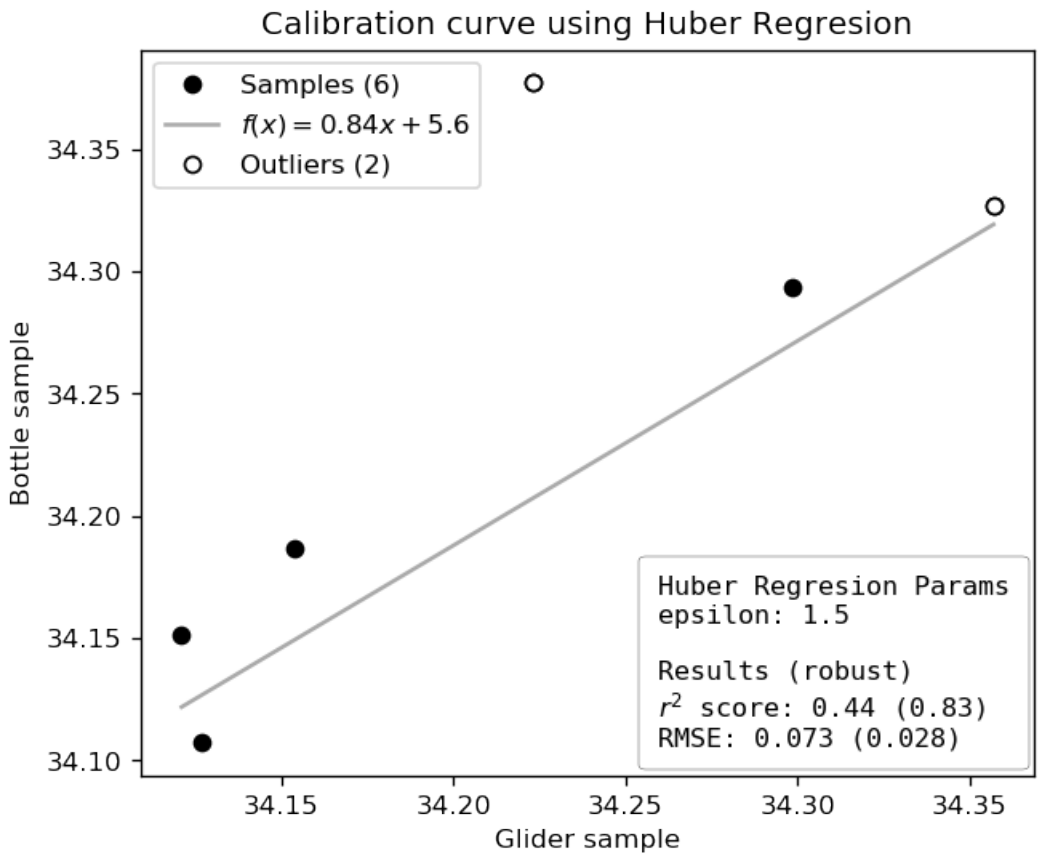
```
[stn 0/5]  FAILED: 2015-07-28 10:25 Couldn't find samples within constraints
[stn 1/5]  FAILED: 2015-07-28 16:15 Couldn't find samples within constraints
[stn 2/5]  FAILED: 2015-12-08 03:23 Couldn't find samples within constraints
[stn 3/5] SUCCESS: 2016-01-05 17:46 (15 of 15 samples) match-up within 0.0 minutes
[stn 4/5] SUCCESS: 2016-02-08 03:14 (12 of 17 samples) match-up within 0.0 minutes
(13, 1) (100, 1)
```



7.2 Using Density

```
%autoreload 2

dat['bottle_sal'] = gt.calibration.bottle_matchup(
    dat.dives, dat.density, dat.time,
    cal.density, cal.datetime, cal.sal)

model = gt.calibration.robust_linear_fit(dat.salt_qc, dat.bottle_sal, fit_intercept=True,
↪ epsilon=1.5)
dat['salinity_qc'] = model.predict(dat.salt_qc)
```

```
[stn 0/5]  FAILED: 2015-07-28 10:25 Couldn't find samples within constraints
[stn 1/5]  FAILED: 2015-07-28 16:15 Couldn't find samples within constraints
[stn 2/5]  FAILED: 2015-12-08 03:23 Couldn't find samples within constraints
[stn 3/5] SUCCESS: 2016-01-05 17:46 (15 of 15 samples) match-up within 0.0 minutes
[stn 4/5] SUCCESS: 2016-02-08 03:14 (16 of 17 samples) match-up within 0.0 minutes
(6, 1) (100, 1)
```


Calibration curve using Huber Regresion
Samples (6)
$f(x) = 0.84x + 5.6$
Outliers (2)
Huber Regresion Params
epsilon: 1.5
Results (robust)
r^2 score: 0.44 (0.83)
RMSE: 0.073 (0.028)
Bottle sample
Glider sample
34.10
34.15
34.20
34.25
34.30
34.35

CHAPTER

EIGHT

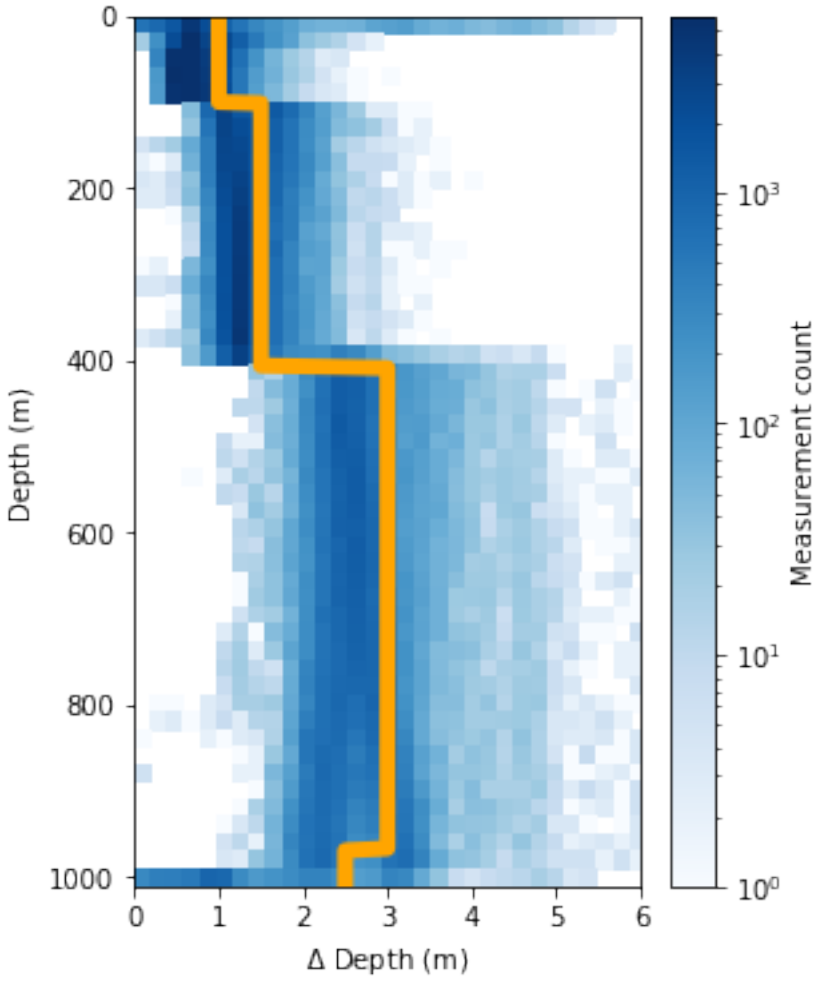
GRIDDING AND INTERPOLATION

8.1 Vertical gridding

It is often more convenient and computationally efficient to work with data that has been gridded to a standard vertical grid (i.e. depths have been binned). GliderTools offers very easy to use and efficient tools to grid data once all the processing has been completed.

The first task is to select the bin size of the data that will be gridded. GliderTools automatically selects bin sizes according to the sampling frequency of the dataset for every 50m. This is shown in the figure below, where the 2D histogram shows the sampling frequency (by depth) and the line shows the automatically selected bin size rounded up to the nearest 0.5m.

```
ax = gt.plot.bin_size(dat.depth, cmap=mpl.cm.Blues)
ax.set_xlim(0, 6)
line = ax.get_children()[1]
line.set_linewidth(6)
line.set_color('orange')

legend = ax.get_children()[-2]
legend.set_visible(False)
```



8.1.1 Gridding with automatic bin sizes

Gridding the data then becomes easy with automatic binning. But note that the x-coordinate has the be semi-discrete, e.g. dives number or dive time stamp average. You'll see that the gridding function also returns the mean bin size and then the average sampling frequency.

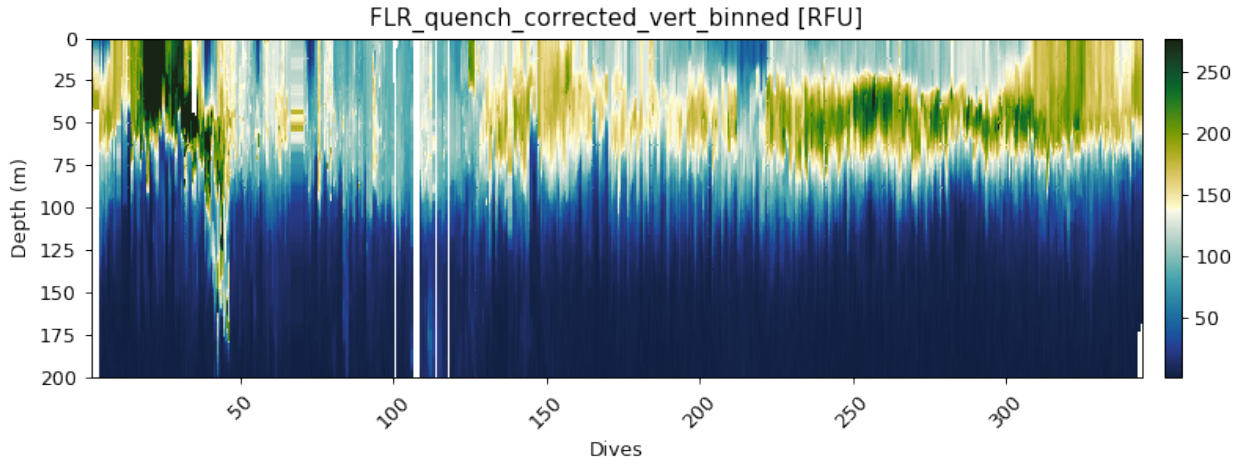
The function can return either an xr.DataArray or a pd.DataFrame. The DataArray is the default as metadata can be stored in these files (including coordinate information).

Gridded data can be passed to the plot function without x- and y-coordinates, as these are contained in the gridded data.

In fact, data is silently passed through the gridding function when x- and y-coordinates are included in the `gt.plot` function

```
flr_gridded = gt.grid_data(dives, depth, flr)

ax = gt.plot(flr_gridded, cmap=cmo.delta)
ax.set_ylim(200, 0)
```

```
Mean bin size = 1.99
Mean depth binned (50 m) vertical sampling frequency = 2.53
```

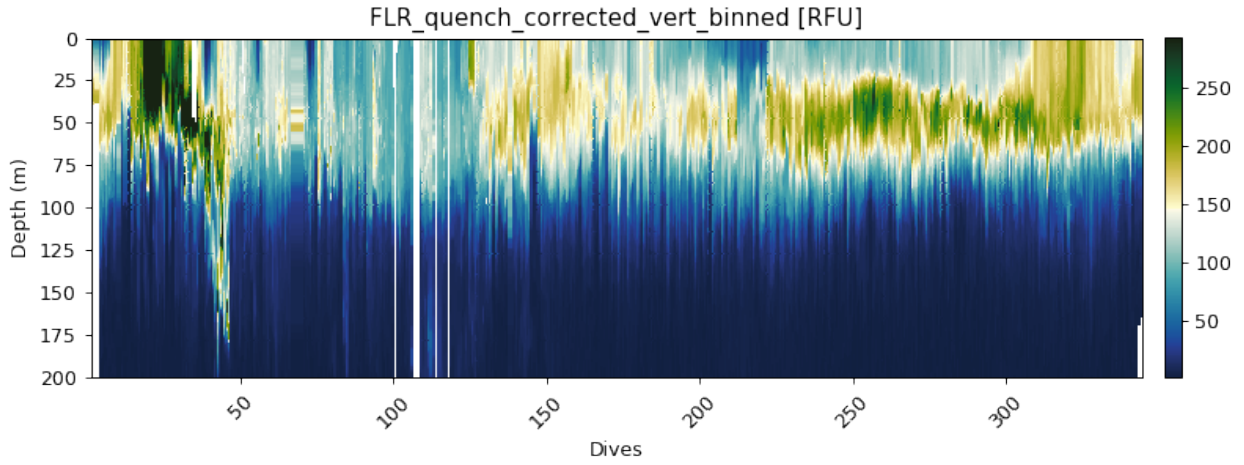


8.1.2 Gridding with manually defined bins

There is also the option to manuualy define your bins if you'd prefer. A custom bin array needs to be created. Use `np.arange` to create sections of the bins and combine them with `np.r_` as shown below:

```
custom_bin = np.r_[
    np.arange(0, 100, 0.5),
    np.arange(100, 400, 1.0),
    np.arange(400, 1000, 2.0)]

flr_gridded = gt.grid_data(x, y, flr, bins=custom_bin)

# The plot below is the standard plotting procedure for an xarray.DataArray
gt.plot(flr_gridded, cmap=cmo.delta)
ylim(200, 0)
```

```
Mean bin size = 1.25
Mean depth binned (50 m) vertical sampling frequency = 2.53




(200, 0)
```



8.2 2D interpolation with objective mapping (Kriging)

Users may want to interpolate data horizontally when working with finescale gradients. Several studies have used the `objmap` MATLAB function that uses objective mapping (a.k.a. Kriging). Kriging is an advanced form of inverse distance weighted interpolation, where points influence the interpolation based on the distance from an interpolation point, where the influence falls off with a Gauassian function. This is an expensive function when the dataset is large (due to a matrix inverse operation). The computational cost is reduced by breaking the problem into smaller pieces using a quadtree that iteratively breaks data into smaller problems.

GliderTools provides a Python implementation of the MATLAB function. We have added parallel capability to speed the processing up, but this operation is still costly and could take several hours if an entire section is interpolated. We thus recommend that smaller sections are interpolated.

```
# first we select a subset of data (50k points)
subs = dat.isel(merged=slice(0, 50000))

# we then get time values - this makes creating the interpolation grid easier
var = subs.flr_qc
time = subs.time.values
depth = subs.depth
dives = subs.dives
dist = np.r_[0, gt.utils.distance(subs.longitude, subs.latitude).cumsum()]
```

8.2.1 Part 1: Semivariance

Interpolating any variable requires some knowlege about the spatial autocorrelation of that variable. A semivariogram allows one to get this information from the data. The basic idea of a semivariogram is to assess the similarity between data at different lengthscales (lags), where a low semivariance shows coherence and a large semivariance shows a mismatch. This information is required to interpolate data with sensible estimates and error estimates.

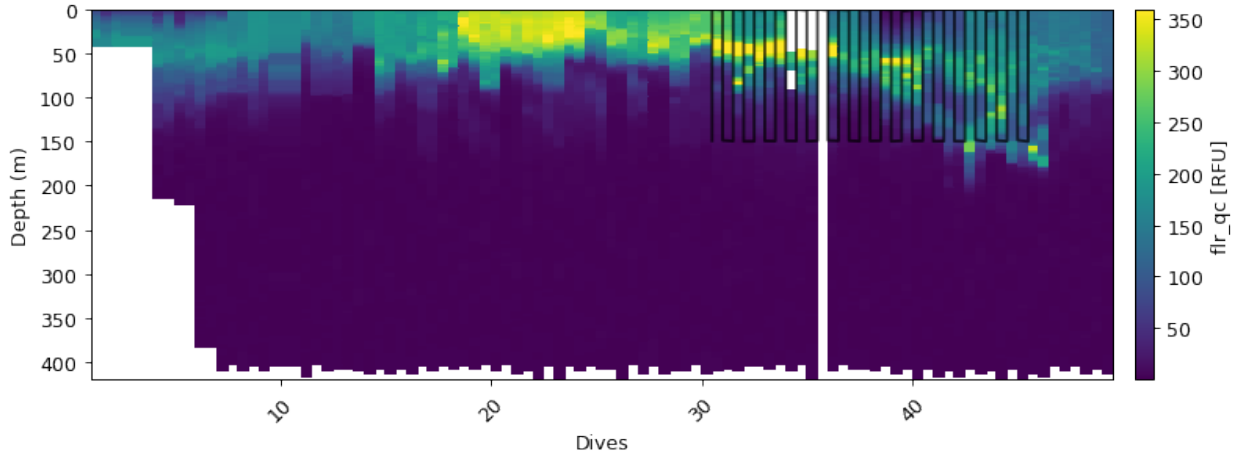
GliderTools offers a derivation of a variogram tool (`gt.mapping.variogram`) that makes the process of finding these parameters a little easier, though there is a fair deal of subjectivity, depending on the scale of the question at hand, and tinkering are required to make a sensible interpolation.

1.1. Choosing a subset of the data for semivariance estimation

The variogram function selects a number of dives (number depends on max_points) and performs the analysis on the subset of dives rathern than selecting random points. We thus recommend that a subset of the data is used to perform the analysis. In the example below, we take a subset of the data that as particularly high variability that we are interested in preserving. This subset is < 250m depth and limited to the first 20 dives. This should be tailored to the variable that you're interested in.

```
m = (depth<150) & (dives > 30) & (dives < 46)
ax = gt.plot(dives, depth, var)
ax.plot(dives[m], depth[m], '-m', ms=3, alpha=0.7)
```

```
[<matplotlib.lines.Line2D at 0x1c728526d8>]
```



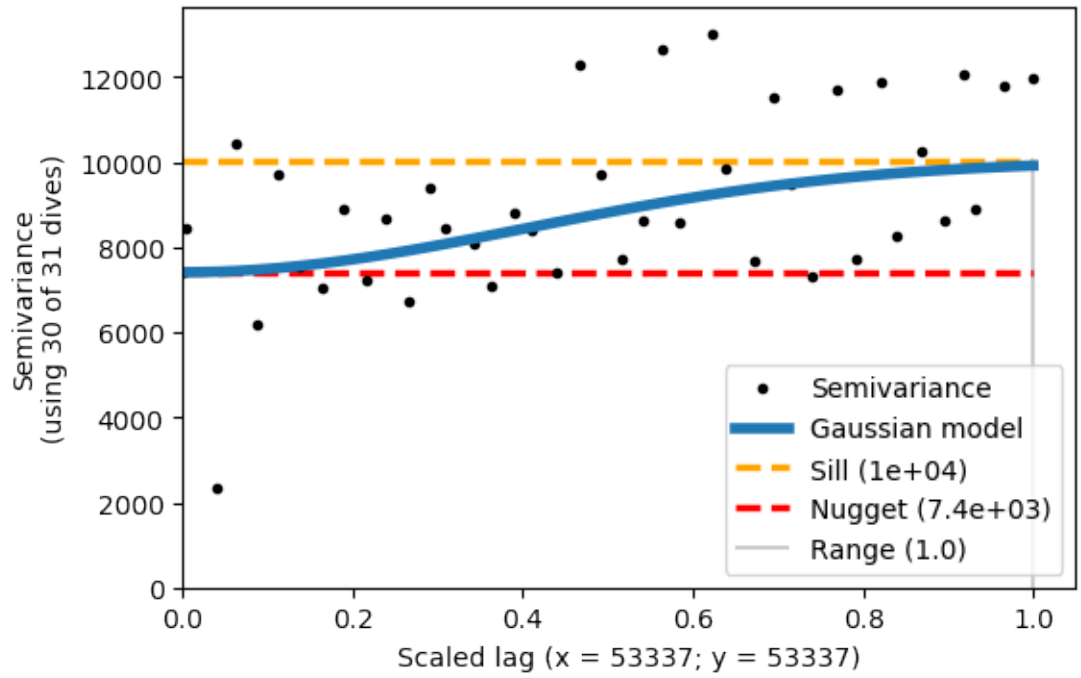
1.2. Initial estimate of semivariance

We can now find an initial estimate of the semivariance. This initial estimate will not scale the x/y coordinates for anisotropy (different scales of variability). The variogram function also accepts a boolean mask as an keyword argument. This will reduce the input data to the subset of data that you've chosen.

The example below shows this initial estimate. We're looking for an estimate where the Gaussian model fits the semivariance as well as possible, given that the variance paramters are acceptable. These variance parameters are: *sill, nugget, x and y length-scales*. The function automatically adjusts the range to be one and scales the x and y parameters accordingly.

The variogram function can take time (datetime64), but we use distance (in metres) to demonstrate the the anisotropic scaling.

```
vargram = gt.mapping.variogram(var, dist, depth, dives, mask=m)
```



The example above shows that x and y are scaled, but the Gaussian model does not fit the semivariance very well. The range is 1, because it is scaled accordingly. The sill and nugget are very similar - this is not a good result.

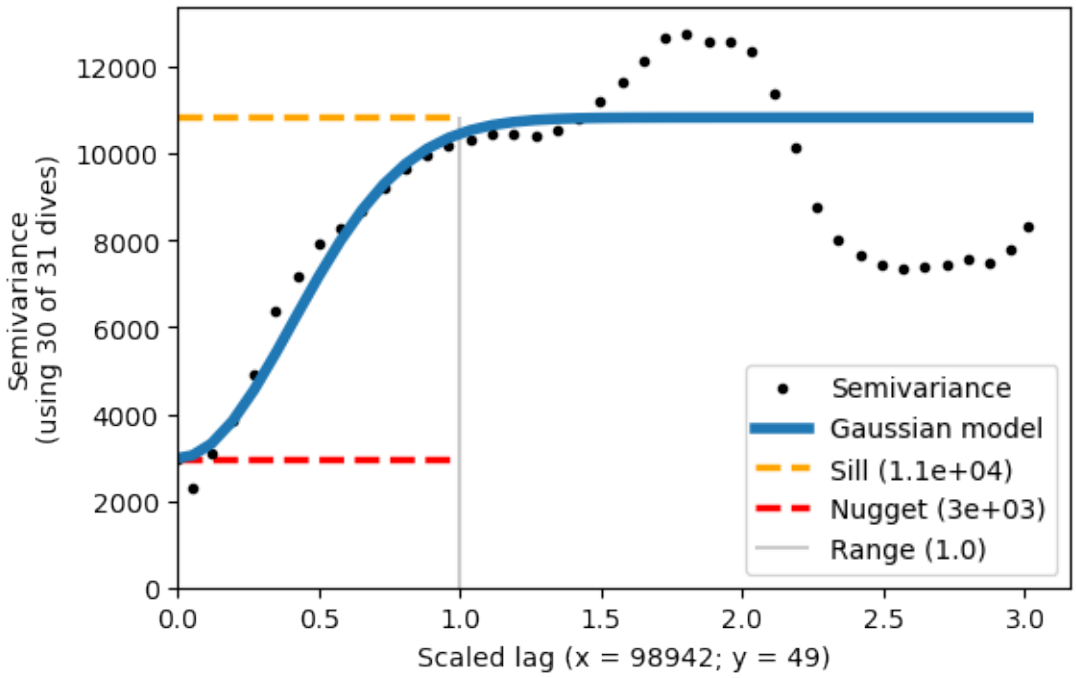
1.3. Finding the correct x and y length scales (anisotropy)

We can now scale the data with the xy_ratio. The ratio represents the scaling of x/y. For example, if x and y are both in metres (as in this case), we need to set a small xy_ratio as x has a much longer lengthscale. With some trial and error we choose a ratio of 0.0005, which fits the semivariogram relatively well and has a reasonably low *y* scaling estimate.

You'll see that the Gaussian model does not fit the semivariance exactly - this is OK. The important thing is that the first plateau matches the sill.

We can now use these values for interpolating.

```
vargram = gt.mapping.variogram(var, dist, depth, dives, mask=m, xy_ratio=0.0005)
```



8.2.2 2. Interpolation

2.1 Preparing the interpolation grid

To perform the interpolation we first need to create the grid onto which data will be interpolated. In the example below we use distance from the origin as the x-coordinate. Time can also be used and has to be in a `np.datetime64` format - we show a commented example of this. The y-coordinate is depth.

```
# creating the x- and y-interpolation coordinates
# and a 1m vertical grid and a horizontal grid with 500 points
xi = np.linspace(dist.min(), dist.max(), 500)
yi = np.arange(0, depth[var.notnull()].max(), 1, dtype=float)

# time can also be used. This is a commented example of how to create
# a time grid for interpolation.
# xi = np.arange(time.min(), time.max(), 30, dtype='datetime64[m]')
```

2.2 Interpolation with the semivariance parameters

The interpolation has a number of parameters that can be changed or adapted to the dataset at hand. The commented inputs below describe these inputs.

```
%autoreload 2

interpolated = gt.mapping.interp_obj(
    dist, depth, var, xi, yi,

    # Kriging interoplation arguments
```

(continues on next page)

(continued from previous page)

```
    partial_sill=1.1e4,  # taken from the semivariogram (sill - nugget)
    nugget=3e3,  # taken from the semivariogram
    lenscale_x=98942,  # in hours if x and xi are in datetime64
    lenscale_y=50,  # the vertical gridding influence
    detrend=True,  # if True use linear regression (z - z_hat), if False use average (z -
↪ z_mean)

    # Quadtree arguments
    max_points_per_quad=65,  # an optimsation setting ~100 is good
    min_points_per_quad=8,  # if neighbours have < N points, look at their neighbours

    # Parallel calculation inputs.
    n_cpus=3,  # the number of CPU's to use for the calculation - default is n-1
    parallel_chunk_size=512,  # when the dataset is very large, memory can become an␣
↪issue
                              # this prevents large buildup of parallel results
)
```

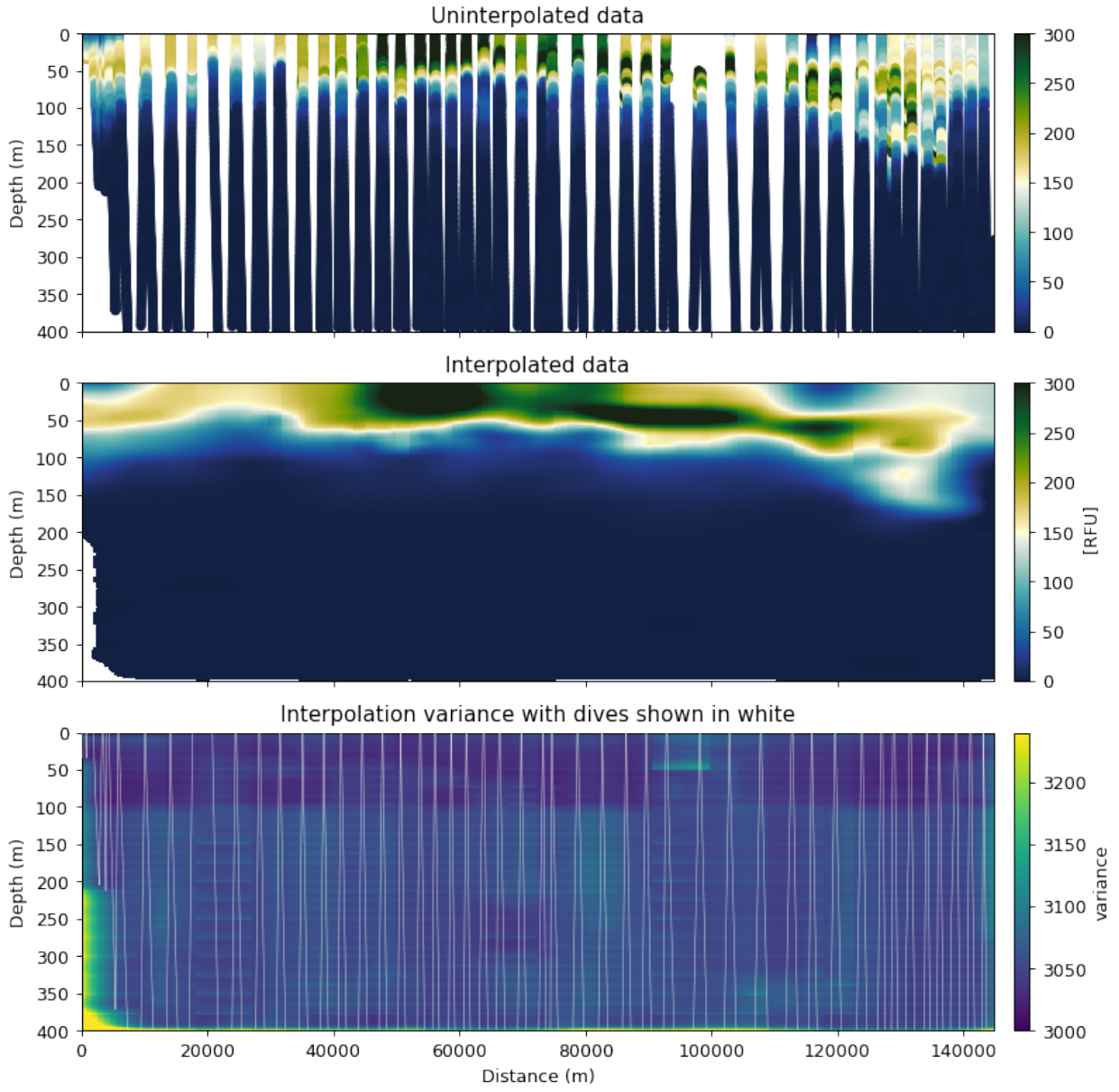
```
Starting Interpolation with quadtree optimal interpolation
------------------------------------------------------------

Preparing for interpolations:
        Finding and removing nans
        Removing data trend with linear regression
        Building QuadTree

Interpolation information:
        basis points:        25226
        interp grid:         500, 404
        max_points_per_quad: 65
        min_points_per_quad: 8
        number of quads:     952
        detrend_method:      linear_regression
        partial_sill:        11000.0
        nugget:              3000.0
        lengthscales:        X = 98942
                             Y = 50 m

Processing interpolation chunks in 2 parts over 3 CPUs:
        chunk 1/2 completed in 12s
        chunk 2/2 completed in 10s

Finishing off interoplation
        Adding back the trend
        Creating xarray dataset for output
```

```
fig, ax = plt.subplots(3, 1, figsize=[9, 9], sharex=True, dpi=90)

error_mask = (interpolated.variance / interpolated.nugget) < 1.05
interp_robust = interpolated.z.where(error_mask)
```

(continues on next page)

(continued from previous page)

```
props = dict(vmin=0, vmax=300, cmap=cmo.delta)
gt.plot.scatter(dist, depth, var, ax=ax[0], **props)
gt.plot.pcolormesh(interp_robust, ax=ax[1], **props)
gt.plot.pcolormesh(interpolated.variance, ax=ax[2], vmin=interpolated.nugget,␣
↪vmax=interpolated.nugget*1.08)

ax[2].plot(dist, depth, 'w-', zorder=40, alpha=0.8, lw=0.4)

[a.set_ylim(400, 0) for a in ax]
[a.set_xlabel('  ') for a in ax]

ax[0].get_children()[0].set_sizes([20])
ax[0].set_title('Uninterpolated data')
ax[1].set_title('Interpolated data')
ax[2].set_title('Interpolation variance with dives shown in white')
ax[2].set_xlabel('Distance (m)')

tks = xticks(rotation=0)
```


Uninterpolated data
Interpolated data
Interpolation variance with dives shown in white
Depth (m)
Distance (m)
[RFU]
variance
0
50
100
150
200
250
300
350
400
20000
40000
60000
80000
100000
120000
140000
3000
3050
3100
3150
3200

CHAPTER

NINE

SAVING DATA

We have not created an explicit way to save data in GliderTools. This is primarily due to the fact that the package is built on top of two packages that already do this very well: *pandas* and *xarray*. *pandas* is widely used and deals with tabular formatted data (2D). *xarray* widely used in earth sciences as it supports multi-dimensional indexing (3D+). We highly recommend that you read through the documentation for these packages as they are incredibly powerful and you would benefit from knowing these tools regardless of using GliderTools or not!

We have written GliderTools primarily with *xarray* as the backend, due to the ability to store attributes (or metadata) alongside the data - something that *pandas* does not yet do. Moreover, we have also created the tool so that metadata is passed to the output of each function, while appending the function call to the *history* attribute. This ensures that the user of the data knows when and what functions (and arguements) were called and for which version of GliderTools this was done.

9.1 Examples

First we give an example of how to save and read files to netCDF (which we recommend).

```
import xarray as xr

# xds is an xarray.DataFrame with record of dimensions, coordinates and variables
xds.to_netcdf('data_with_meta.nc')

# this file can simply be loaded in the same way, without using GliderTools
# all the information that was attached to the data is still in the netCDF
xds = xr.open_dataset('data_with_meta.nc')
```

In this second example we show how to save the data to a CSV. While this is a common and widely used format, we do not recommend this as the go to format, as all metadata is lost when the file is saved.

```
import pandas as pd

# If you prefer to save your data as a text file, you can easily do this with Pandas
# note that converting the file to a dataframe discards all the metadata
df = xds.to_dataframe()
df.to_csv('data_without_meta.csv')

# this file can simply be loaded in the same way, without using GliderTools
# there will be no more metadata attached to each variable
df = pd.read_csv('data_without_meta.csv')
```

(continues on next page)

(continued from previous page)

```
# finally, you can also convert the file back to an xarray.Dataset
# however, the data will still be lost
xds = df.to_xarray()
```

CHAPTER

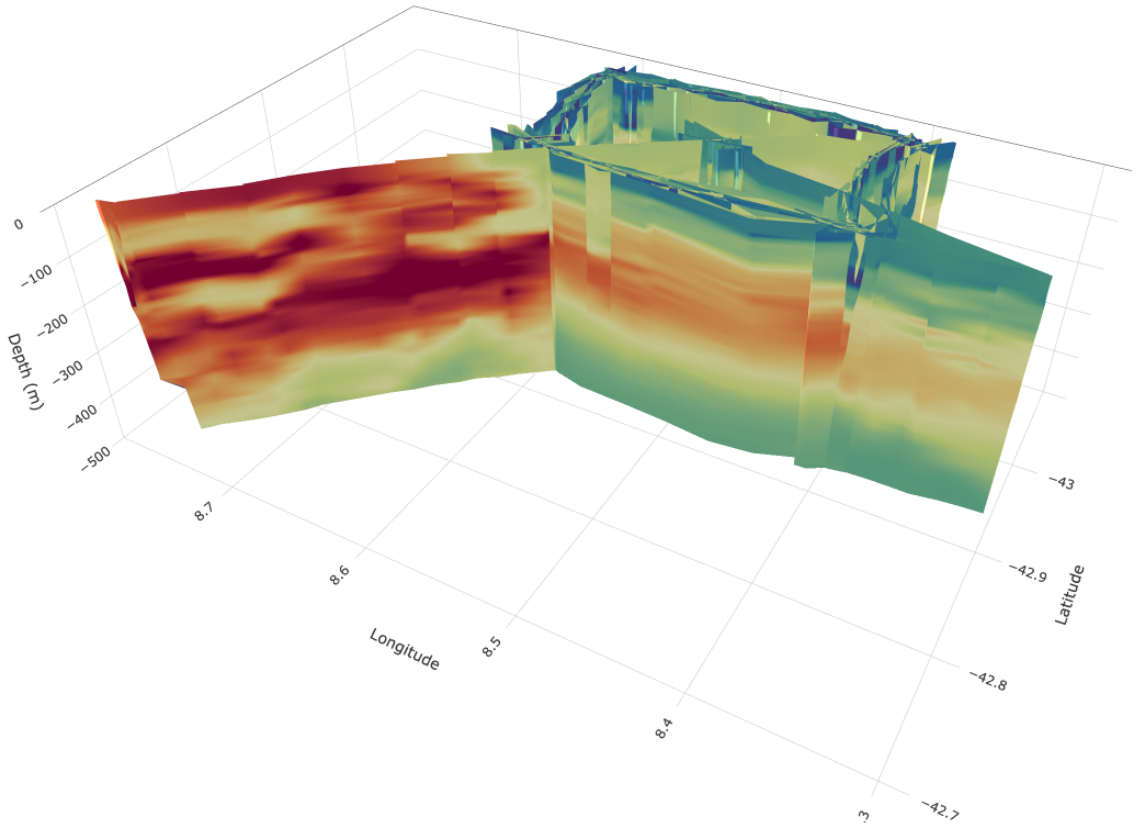
TEN

OTHER TOOLS AND UTILITIES

10.1 3D interactive plot

This is purely for investigative purposes, but provides a good way to interact with the data.

```
plotly_figure = gt.plot.section3D(
    dat.dives, dat.depth, dat.longitude, dat.latitude, dat.salt_qc,
    zmin=-500, vmax=.999, vmin=.005
)
```



CHAPTER

ELEVEN

API REFERENCE

The API reference is automatically generated from the function docstrings in the GliderTools package. Refer to the examples in the sidebar for reference on how to use the functions.

11.1 Loading Data

load.seaglider_basestation_netCDFs(files, ...)	Load a list of variables from the SeaGlider object as a `pandas.DataFrame`.
load.seaglider_show_variables(files)	
load.ego_mission_netCDF(filename)	Loads an EGO formatted glider mission file.
load.slocum_geomar_matfile(filename[, verbose])	Load .mat file generated with the geomar MATLAB script for Slocum data.
load.voto_seaexplorer_nc(filename)	Load .nc file downloaded from https://observations.voiceoftheocean.org/.
load.voto_seaexplorer_dataset(ds)	Adapts a VOTO xarray dataset, for example downloaded from the VOTO ERDAP server (https://erddap.observations.voiceoftheocean.org/erddap/index.html) to be used in GliderTools
load.voto_concat_datasets(datasets)	Concatenates multiple datasets along the time dimensions, profile_num and dives variable(s) are adapted so that they start counting from one for the first dataset and monotonically increase.

11.1.1 glidertools.load.seaglider_basestation_netCDFs

glidertools.load.**seaglider_basestation_netCDFs**(*files*, *variable_names*, *return_merged=False*, *verbose=True*, *keep_global_attrs=False*, *netcdf_attrs={}*, *keep_variable_attrs=True*)

Load a list of variables from the SeaGlider object as a `pandas.DataFrame`.

Parameters

variable_names (*list*) – a list of strings representing the keys you would like to load.

Returns

Will always have coordinate dimensions loaded (even if not specified). These can then be accessed either by the variable objects or by .data[<dimension_name>].

Return type
pandas.DataFrame

Note: Using this method resets all previously loaded and stored data (data is stored under `SeaGlider.data={dim: pandas.DataFrame}`). This is done to avoid erroneous coordinate matchup with sometimes missing data).

11.1.2 glidertools.load.seaglider_show_variables

`glidertools.load.`**`seaglider_show_variables`**(*files*)

11.1.3 glidertools.load.ego_mission_netCDF

`glidertools.load.`**`ego_mission_netCDF`**(*filename*)

Loads an EGO formatted glider mission file.

Parameters
`filename` (*`str`*) – path and filename of the EGO netCDF file.

Return type
an xarray.Dataset object with all netCDF info attached

ego_data : xr.Dataset

11.1.4 glidertools.load.slocum_geomar_matfile

`glidertools.load.`**`slocum_geomar_matfile`**(*filename*, *verbose=True*)

Load .mat file generated with the geomar MATLAB script for Slocum data.

A dive column is generated on importing the data. When single values per dive (e.g. u/v), the value is set for the entire dive.

Parameters

- **`filename`** (*`str`*) – path of .mat file.
- **`verbose`** (*`bool, optional`*) – defaults to True

Returns
DataFrame containing the all columns in the *.mat* file

Return type
pandas.DataFrame

11.1.5 glidertools.load.voto_seaexplorer_nc

glidertools.load.**voto_seaexplorer_nc**(*filename*)

Load .nc file downloaded from https://observations.voiceoftheocean.org/. A dives column is generated on importing the data.

Parameters
filename (*str*) – path of .nc file.

Returns
Dataset containing the all columns in the source file and dives column

Return type
xarray.Dataset

11.1.6 glidertools.load.voto_seaexplorer_dataset

glidertools.load.**voto_seaexplorer_dataset**(*ds*)

Adapts a VOTO xarray dataset, for example downloaded from the VOTO ERDAP server (https://erddap.observations.voiceoftheocean.org/erddap/index.html) to be used in GliderTools

Parameters
ds (*xarray.Dataset*)

Returns
Dataset containing the all columns in the source file and dives column

Return type
xarray.Dataset

11.1.7 glidertools.load.voto_concat_datasets

glidertools.load.**voto_concat_datasets**(*datasets*)

Concatenates multiple datasets along the time dimensions, profile_num and dives variable(s) are adapted so that they start counting from one for the first dataset and monotonically increase.

Parameters
datasets (*list of xarray.Datasets*)

Returns
concatenated Dataset containing all the data from the list of datasets

Return type
xarray.Dataset

11.2 High level processing

processing.calc_physics(variable, dives, depth)	A standard setup for processing physics variables (temperature, salinity).
processing.calc_oxygen(o2raw, pressure, ...)	This function processes oxygen.
processing.calc_backscatter(bb_raw, tempC, ...)	The function processes the raw backscattering data in counts into total backscatter (bbp) in metres.
processing.calc_fluorescence(flr_raw, bbp, ...)	This function processes raw fluorescence and corrects for quenching using the Thomalla et al. (2018) approach [1]_.
processing.calc_par(par_raw, dives, depth, ...)	Calculates the theoretical PAR based on an exponential curve fit.

11.2.1 glidertools.processing.calc_physics

glidertools.processing.**calc_physics**(*variable*, *dives*, *depth*, *spike_window=3*, *spike_method='minmax'*, *iqr=1.5*, *depth_threshold=400*, *mask_frac=0.2*, *savitzky_golay_window=11*, *savitzky_golay_order=2*, *verbose=True*, *name='Physics Variable'*)

A standard setup for processing physics variables (temperature, salinity).

The function applies a neighbourhood interquartile range (IQR) outlier filter, the Briggs et al. (2011) spike filter followed by a Savitzky-Golay smoothing function.

The Savitzky-Golay filter is demonstrated well on wikipedia: https://en.wikipedia.org/wiki/Savitzky-Golay_filter

11.2.2 glidertools.processing.calc_oxygen

glidertools.processing.**calc_oxygen**(*o2raw*, *pressure*, *salinity*, *temperature*, *lat*, *lon*)

This function processes oxygen.

It is assumed umol/kg are passed as input. The units are automatically detected by looking at the mean ratio. Below are some conversions to help with the Oxygen units: If your oxygen values have units ml/L use the conversion function oxygen_ml_per_l_to_umol_per_kg

Parameters

- **o2raw** (*array, dtype=float, shape=[n,]*) – raw oxygen in umol/kg
- **pressure** (*array, dtype=float, shape=[n,]*)
- **salinity** (*array, dtype=float, shape=[n,]*)
- **temperature** (*array, dtype=float, shape=[n,]*)
- **lat** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The latitude of the glider position.
- **lon** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The longitude of the glider position.

Returns

- **o2mll** (*array, dtype=float, shape=[n,]*) – oxygen concentration in mL/L

- **o2pct** (*array, dtype=float, shape=[n,]*) – theoretical oxygen saturation percentage
- **o2aou** (*array, dtype=float, shape=[n,]*) – apparent oxygen utilisation based on measured oxygen and oxygen saturation.

11.2.3 glidertools.processing.calc_backscatter

glidertools.processing.**calc_backscatter**(*bb_raw*, *tempC*, *salt*, *dives*, *depth*, *wavelength*, *dark_count*, *scale_factor*, *spike_window=7*, *spike_method='median'*, *iqr=3*, *profiles_ref_depth=300*, *deep_multiplier=1*, *deep_method='median'*, *return_figure=False*, *verbose=True*)

The function processes the raw backscattering data in counts into total backscatter (bbp) in metres.

The function uses a series of steps to clean the data before applying the Zhang et al. (2009) functions to convert the data into total backscatter (bbp/m)). The function uses functions from the flo_functions toolkit[1]. The theta angle of sensors (124deg) and xfactor for theta 124 (1.076) are set values that should be updated if you are not using a WetLabs ECO BB2FL

The following standard sequence is applied:

1. find IQR outliers (i.e. data values outside of the lower and upper limits calculated by cleaning.outlier_bounds_iqr)
2. find_bad_profiles (e.g. high values below 300 m are counted as bad profiles)
3. flo_scale_and_offset (factory scale and offset)
4. flo_bback_total (total backscatter based on Zhang et al. 2009)[2]
5. backscatter_dark_count (based on Briggs et al. 2011)[3]
6. despike (using Briggs et al. 2011 - rolling min–max)[3]

Parameters

- **bb_raw** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The raw output from the backscatter channel in counts.
- **tempC** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The QC'd temperature data in degC.
- **salt** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The QC'd salinity in PSU.
- **dives** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The depth array in metres.
- **wavelength** (*int*) – The wavelength of the backscatter channel, e.g. 700 nm.
- **dark_count** (*float*) – The dark count factory values from the calibration sheet.
- **scale_factor** (*float*) – The scale factor factory values from the calibration sheet.

[1] https://github.com/ooici/ion-functions Copyright (c) 2010, 2011 The Regents of the University of California

[2] Zhang, X., Hu, L., & He, M. (2009). Scattering by pure seawater: Effect of salinity. Optics Express, 17(7), 5698. https://doi.org/10.1364/OE.17.005698

[3] Briggs, N., Perry, M. J., Cetinic, I., Lee, C., D'Asaro, E., Gray, A. M., & Rehm, E. (2011). High-resolution observations of aggregate flux during a sub-polar North Atlantic spring bloom. Deep-Sea Research Part I: Oceanographic Research Papers, 58(10), 1031–1039. https://doi.org/10.1016/j.dsr.2011.07.007

- **spike_window** (*int*) – The window size over which to run the despiking method.
- **spike_method** (*str*) – Whether to use a rolling median or combination of min+max filter as the despiking method.
- **iqr** (*int*) – Multiplier to determine the lower and upper limits of the interquartile range for outlier detection.
- **profiles_ref_depth** (*int*) – The depth threshold for optics.find_bad_profiles below which the median or mean is calculated for identifying outliers.
- **deep_multiplier** (*int=1*) – The standard deviation multiplier for calculating outliers, i.e. $\mu \pm \sigma \cdot [1]$.
- **deep_method** (*str*) – Whether to use the deep median or deep mean to determine bad profiles for optics.find_bad_profiles.
- **return_figure** (*bool*) – If True, will return a figure object that shows before and after the quenching correction was applied.
- **verbose** (*bool*) – If True, will print the progress of the processing function.

Returns

- **baseline** (*numpy.ma.masked_array*) – The despiked + bad profile identified backscatter with the mask denoting the filtered values of the backscatter baseline as defined in Briggs et al. (2011).
- **quench_corrected** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The backscatter spikes as defined in Briggs et al. (2011).
- **figs** (*object*) – The figures reporting the despiking, bad profiles and quenching correction.

References

11.2.4 glidertools.processing.calc_fluorescence

glidertools.processing.**calc_fluorescence**(*flr_raw*, *bbp*, *dives*, *depth*, *time*, *lat*, *lon*, *dark_count*, *scale_factor*, *spike_window=7*, *spike_method='median'*, *night_day_group=True*, *sunrise_sunset_offset=1*, *profiles_ref_depth=300*, *deep_multiplier=1*, *deep_method='median'*, *return_figure=False*, *verbose=True*)

This function processes raw fluorescence and corrects for quenching using the Thomalla et al. (2018) approach[1].

The following standard sequence is applied:

1. find_bad_profiles (e.g. high Fluorescence in > 300 m water signals bad profile)
2. fluorescence_dark_count & scale factor (i.e. factory correction)
3. despike (using Briggs et al. 2011 - rolling min–max)
4. quenching_correction (corrects for quenching with Thomalla et al. 2017)

Parameters

- **flr_raw** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The raw output of fluorescence data in instrument counts.

[1] Thomalla, S. J., Moutier, W., Ryan-Keogh, T. J., Gregor, L., & Schutt, J. (2018). An optimized method for correcting fluorescence quenching using optical backscattering on autonomous platforms. Limnology and Oceanography: Methods, 16(2), 132–144. https://doi.org/10.1002/lom3.10234

- **bbp** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The processed backscatter data from the less noisy channel, i.e. the one dataset with less spikes or bad profiles.
- **dives** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The depth array in metres.
- **time** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The date & time array in a numpy.datetime64 format.
- **lat** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The latitude of the glider position.
- **lon** (*np.array / pd.Series, dtype=float, shape=[n,]*) – The longitude of the glider position.
- **dark_count** (*float*) – The dark count factory values from the calibration sheet.
- **scale_factor** (*float*) – The scale factor factory values from the calibration sheet.
- **spike_window** (*int=7*) – The window size over which to run the despiking method.
- **spike_method** (*str=median*) – Whether to use a rolling median or combination of min+max filter as the despiking method.
- **night_day_group** (*bool=True*) – If True, use preceding night otherwise use following night for calculating the flr:bbp ratio.
- **sunrise_sunset_offset** (*int=1*) – The delayed onset and recovery of quenching in hours [1] (assumes symmetrical).
- **profiles_ref_depth** (*int=300*) – The depth threshold for optics.find_bad_profiles below which the median or mean is calculated for identifying outliers.
- **deep_multiplier** (*int=1*) – The standard deviation multiplier for calculating outliers, i.e. mean $\pm$ std x [1].
- **deep_method** (*str='median'*) – Whether to use the deep median or deep mean to determine bad profiles for optics.find_bad_profiles.
- **return_figure** (*bool=False*) – If True, will return a figure object that shows before and after the quenching correction was applied.
- **verbose** (*bool=True*) – If True, will print the progress of the processing function.

Returns

- **baseline** (*array, dtype=float, shape=[n,]*) – The despiked + bad profile identified fluorescence that has not had the quenching correction applied.
- **quench_corrected** (*array, dtype=float, shape=[n,]*) – The fluorescence data corrected for quenching.
- **quench_layer** (*array, dtype=bool, shape=[n,]*) – The quenching layer as a mask.
- **figs** (*object*) – The figures reporting the despiking, bad profiles and quenching correction.

References

11.2.5 glidertools.processing.calc_par

glidertools.processing.**calc_par**(*par_raw*, *dives*, *depth*, *time*, *scale_factor_wet_uEm2s*, *sensor_output_mV*, *curve_max_depth=80*, *verbose=True*)

Calculates the theoretical PAR based on an exponential curve fit.

The processing steps are:

1. par_scaling (factory cal sheet scaling)
2. par_dark_count (correct deep par values to 0 using 5th %)
3. par_fill_surface (return the theoretical curve of par based exponential fit)

Parameters

- **data** (*All inputs must be ungridded np.ndarray or pd.Series*)
- **par_raw** (*array, dtype=float, shape=[n,]*) – raw PAR
- **dives** (*array, dtype=float, shape=[n,]*) – the dive count (round is down dives, 0.5 up dives)
- **depth** (*array, dtype=float, shape=[n,]*) – in metres
- **time** (*array, dtype=float, shape=[n,]*) – as a np.datetime64 array

Returns
par_filled – PAR with filled surface values.

Return type
array, dtype=float, shape=[n,]

11.3 Cleaning

cleaning.outlier_bounds_std(arr[, multiplier])	Mask values outside the upper and lower outlier limits by standard deviation
cleaning.outlier_bounds_iqr(arr[, multiplier])	Mask values outside the upper/lower outlier limits by interquartile range:
cleaning.horizontal_diff_outliers(dives, ...)	Find Z-score outliers (> 3) on the horizontal.
cleaning.mask_bad_dive_fraction(mask, dives, var)	Find bad dives - where more than a fraction of the dive is masked
cleaning.data_density_filter(x, y[, ...])	Use the 2D density cloud of observations to find outliers for any variables
cleaning.despike(var, window_size[, ...])	Return a smooth baseline of data and the anomalous spikes
cleaning.despiking_report(dives, depth, raw, ...)	A report for the results of cleaning.despike.
cleaning.rolling_window(var, func, window)	A rolling window function that is nan-resiliant
cleaning.savitzky_golay(var, window_size, order)	Smooth (and optionally differentiate) data with a Savitzky-Golay filter.

11.3.1 glidertools.cleaning.outlier_bounds_std

glidertools.cleaning.**outlier_bounds_std**(*arr*, *multiplier=3*)

Mask values outside the upper and lower outlier limits by standard deviation

$$\mu \pm 3\sigma$$

the multiplier [3] can be adjusted by the user returns the lower_limit, upper_limit

Parameters

- **arr** (*np.array|xr.DataArray, dtype=float, shape=[n,]*) – the full timeseries of the entire dataset
- **multiplier** (*float=1.5*) – sets the standard deviation multiplier

Returns
arr – A data object where values outside the limits are masked. Metdata will be preserved if the original input array is xr.DataArray

Return type
array | xarray.DataArray

11.3.2 glidertools.cleaning.outlier_bounds_iqr

glidertools.cleaning.**outlier_bounds_iqr**(*arr*, *multiplier=1.5*)

Mask values outside the upper/lower outlier limits by interquartile range:

$$lim_{low} = Q_1 - 1.5 \cdot (Q_3 - Q_1)$$
$$lim_{up} = Q_3 + 1.5 \cdot (Q_3 - Q_1)$$

the multiplier [1.5] can be adjusted by the user returns the lower_limit, upper_limit

Parameters

- **arr** (*np.array|xr.DataArray, dtype=float, shape=[n,]*) – the full timeseries of the entire dataset
- **multiplier** (*float=1.5*) – sets the interquartile range

Returns
arr – A data object where values outside the limits are masked. Metdata will be preserved if the original input array is xr.DataArray

Return type
array | xarray.DataArray

11.3.3 glidertools.cleaning.horizontal_diff_outliers

glidertools.cleaning.**horizontal_diff_outliers**(*dives*, *depth*, *arr*, *multiplier=1.5*, *depth_threshold=450*, *mask_frac=0.2*)

Find Z-score outliers (> 3) on the horizontal. Can be limited below a certain depth.

The function uses the horizontal gradient as a threshold, below a defined depth threshold to find outliers. Useful to identify when a variable at depth is not the same as neighbouring values.

Parameters

- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives)
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres
- **arr** (*numpy.ndarray or pandas.Series*) – Array of data variable for cleaning to be performed on.
- **multiplier** (*float*) – A z-score threshold
- **depth_threshold** (*int*) – Outliers will be identified below this depth value to the max depth value of the dive.
- **mask_frac** (*float*) – When the ratio of bad values per dive is greater than this value, then the dive will be masked.

Returns
A mask of dives where the bad values per dive ratio is greater than mask_frac.

Return type
mask

11.3.4 glidertools.cleaning.mask_bad_dive_fraction

glidertools.cleaning.**mask_bad_dive_fraction**(*mask*, *dives*, *var*, *mask_frac=0.2*)

Find bad dives - where more than a fraction of the dive is masked

Parameters

- **mask** (*array, dtype=bool, shape=[n,]*) – boolean 1D array with masked values
- **dives** (*array, dtype=float, shape=[n,]*) – discrete dive numbers (down round, up n.5)
- **var** (*array, dtype=float, shape=[n,]*) – series or array containing data that will be masked with NaNs
- **mask_frac** (*int=0.2*) – fraction of the dive that is masked for the whole dive to be bad

Returns

- **var** (*array, dtype=float, shape=[n,]*) – the same as the input, but has been masked
- **mask_dives** (*array, dtype=bool*) – a mask array that has full dives that are deemed "bad" masked out

11.3.5 glidertools.cleaning.data_density_filter

glidertools.cleaning.**data_density_filter**(*x*, *y*, *conv_matrix=None*, *min_count=5*, *return_figures=True*)

Use the 2D density cloud of observations to find outliers for any variables

The data density filter needs tuning to work well. This uses convolution to create the density cloud - you can specify the exact convolution matrix, or its shape

Parameters

- **x** (*np.array / pd.Series, shape=[n,]*) – e.g. temperature
- **y** (*np.array / pd.Series, shape=[n,]*) – e.g. salinity

- **conv_matrix** (*int, list, np.array, optional*) – int = size of the isotropic round convolution window. [int, int] = anisotropic (oval) convoltion window. 2d array is a weighted convolution window; rectangle = np.ones([int, int]); more advanced anisotropic windows can also be created
- **min_count** (*int, default=5, optional*) – masks the 2d histogram counts smaller than this limit when performing the convolution
- **return_figures** (*bool, default=True, optional*) – returns figures of the data plotted for blob detection…

Returns

- **mask** (*np.array, shape=[n,]*) – a mask that returns only values
- *figure* – only returned if return_figure is True

11.3.6 glidertools.cleaning.despike

glidertools.cleaning.**despike**(*var*, *window_size*, *spike_method='median'*)

Return a smooth baseline of data and the anomalous spikes

This script is copied from Nathan Briggs' MATLAB script as described in Briggs et al (2011). It returns the baseline of the data using either a rolling window method and the residuals of [measurements - baseline].

Parameters

- **arr** (*numpy.ndarray or pandas.Series*) – Array of data variable for cleaning to be performed on.
- **window_size** (*int*) – the length of the rolling window size
- **method** (*str*) – A string with *minmax* or *median*. 'minmax' first applies a rolling minimum to the dataset thereafter a rolling maximum is applied. This forms the baseline, where the spikes are the difference from the baseline. 'median' first applies a rolling median to the dataset, which forms the baseline. The spikes are the difference between median and baseline, and thus are more likely to be negative.

Returns

- **baseline** (*numpy.ndarray or pandas.Series*) – The baseline from which outliers are determined.
- **spikes** (*numpy.ndarray or pandas.Series*) – Spikes are the residual of [measurements - baseline].

11.3.7 glidertools.cleaning.despiking_report

glidertools.cleaning.**despiking_report**(*dives*, *depth*, *raw*, *baseline*, *spikes*, *name=None*, *pcolor_kwargs={}*)

A report for the results of cleaning.despike.

The function creates a figure object of 3 subplots containing a pcolormesh of the original data, the despiked data and the spikes calculated from the cleaning.despike function.

Parameters

- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives)

- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres
- **raw** (*numpy.ndarray or pandas.Series*) – Array of data variable for cleaning to be performed on.
- **baseline** (*numpy.ndarray or pandas.Series*) – The baseline from which outliers are determined.
- **spikes** (*numpy.ndarray or pandas.Series*) – Spikes are the residual of [measurements - baseline].
- **name** (*str*) – String name for the figure header, if None will try to get a pandas. Series.name from raw.
- **pcolor_kwargs** (*dict*) – A dictionary of keyword arguements passed to pcolormesh.

Returns

figure – Creates a figure object with 3 subplots containing a pcolormesh of the original data, the despiked data and the spikes calculated from the cleaning.despike function.

Return type

object

11.3.8 glidertools.cleaning.rolling_window

glidertools.cleaning.**rolling_window**(*var*, *func*, *window*)

A rolling window function that is nan-resiliant

Parameters

- **arr** (*array, dtype=float, shape=[n,]*) – array that you want to pass the rolling window over
- **func** (*callable*) – an aggregating function. e.g. mean, std, median
- **window** (*int*) – the size of the rolling window that will be applied

Returns

arr – the same as the input array, but the rolling window has been applied

Return type

array, dtype=float, shape=[n,]

11.3.9 glidertools.cleaning.savitzky_golay

glidertools.cleaning.**savitzky_golay**(*var*, *window_size*, *order*, *deriv=0*, *rate=1*, *interpolate=True*)

Smooth (and optionally differentiate) data with a Savitzky-Golay filter.

The Savitzky-Golay filter removes high frequency noise from data[1]. It has the advantage of preserving the original shape and features of the signal better than other types of filtering approaches, such as moving averages techniques. By default, nans in the array are interpolated with a limit set to the window size of the dataset before smoothing. The nans are inserted back into the dataset after the convolution. This limits the loss of data over blocks where there are nans. This can be switched off with the *interpolate* keyword arguement.

Parameters

- **var** (*array, dtype=float, shape=[n,]*) – the values of the time history of the signal.

[1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

- **window_size** (*int*) – the length of the window. Must be an odd integer number.
- **order** (*int*) – the order of the polynomial used in the filtering. Must be less then *window_size* - 1.
- **deriv** (*int*) – the order of the derivative to compute (default = 0 means only smoothing)
- **interpolate** (*bool=True*) – By default, nans in the array are interpolated with a limit set to the window size of the dataset before smoothing. The nans are inserted back into the dataset after the convolution. This limits the loss of data over blocks where there are nans. This can be switched off with the *interpolate* keyword arguement.

Returns
ys – the smoothed signal (or it's n-th derivative).

Return type
ndarray, shape (N)

Notes

The Savitzky-Golay is a type of low-pass filter, particularly suited for smoothing noisy data. The main idea behind this approach is to make for each point a least-square fit with a polynomial of high order over a odd-sized window centered at the point[2].

Examples

```
>>> t = linspace(-4, 4, 500)
    y = exp( -t**2 ) + random.normal(0, 0.05, t.shape)
    ysg = savitzky_golay(y, window_size=31, order=4)
    import matplotlib.pyplot as plt
    plt.plot(t, y, label='Noisy signal')
    plt.plot(t, exp(-t**2), 'k', lw=1.5, label='Original signal')
    plt.plot(t, ysg, 'r', label='Filtered signal')
    plt.legend()
    plt.show()
```

References

11.4 Physics

physics.mixed_layer_depth(ds, variable[, ...])	Calculates the MLD for ungridded glider array.
physics.potential_density(salt_PSU, temp_C, ...)	Calculate density from glider measurements of salinity and temperature.
physics.brunt_vaisala(salt, temp, pres[, lat])	Calculate the square of the buoyancy frequency.

[2] Numerical Recipes 3rd Edition: The Art of Scientific Computing W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery Cambridge University Press ISBN-13: 9780521880688

11.4.1 glidertools.physics.mixed_layer_depth

glidertools.physics.**mixed_layer_depth**(*ds*, *variable*, *thresh=0.01*, *ref_depth=10*, *verbose=True*)

Calculates the MLD for ungridded glider array.

You can provide density or temperature. The default threshold is set for density (0.01).

Parameters

- **ds** (*xarray.Dataset Glider dataset*)
- **variable** (*str*) – variable that will be used for the threshold criteria
- **thresh** (*float=0.01 threshold for difference of variable*)
- **ref_depth** (*float=10 reference depth for difference*)
- **return_as_mask** (*bool, optional*)
- **verbose** (*bool, optional*)

Returns

mld – will be an array of depths the length of the number of unique dives.

Return type

array

11.4.2 glidertools.physics.potential_density

glidertools.physics.**potential_density**(*salt_PSU*, *temp_C*, *pres_db*, *lat*, *lon*, *pres_ref=0*)

Calculate density from glider measurements of salinity and temperature.

The Basestation calculates density from absolute salinity and potential temperature. This function is a wrapper for this functionality, where potential temperature and absolute salinity are calculated first. Note that a reference pressure of 0 is used by default.

Parameters

- **salt_PSU** (*array, dtype=float, shape=[n,]*) – practical salinty
- **temp_C** (*array, dtype=float, shape=[n,]*)
- **C** (*temperature in deg*)
- **pres_db** (*array, dtype=float, shape=[n,]*) – pressure in decibar
- **lat** (*array, dtype=float, shape=[n,]*) – latitude in degrees north
- **lon** (*array, dtype=float, shape=[n,]*) – longitude in degrees east

Returns

potential_density

Return type

array, dtype=float, shape=[n,]

11.4.3 glidertools.physics.brunt_vaisala

glidertools.physics.**brunt_vaisala**(*salt*, *temp*, *pres*, *lat=None*)

Calculate the square of the buoyancy frequency.

This is a copy from GSW package, with the exception that the array maintains the same shape as the input. Note that it only works on ungridded data at the moment.

N^{2} = frac{-g}{sigma_{theta}} frac{dsigma_{theta}}{dz}

Parameters

- **SA** (*array-like*) – Absolute Salinity, g/kg
- **CT** (*array-like*) – Conservative Temperature (ITS-90), degrees C
- **p** (*array-like*) – Sea pressure (absolute pressure minus 10.1325 dbar), dbar
- **lat** (*array-like, 1-D, optional*) – Latitude, degrees.
- **axis** (*int, optional*) – The dimension along which pressure increases.

Returns

N2 – Buoyancy frequency-squared at pressure midpoints, 1/s. The shape along the pressure axis dimension is one less than that of the inputs.

Return type

array

11.5 Optics

optics.find_bad_profiles(dives, depth, var)	Find profiles that exceed a threshold at a reference depth.
optics.par_dark_count(par, depth, time[, ...])	Calculates an in situ dark count from the PAR sensor.
optics.backscatter_dark_count(bbp, depth[, ...])	Calculates an in situ dark count from the backscatter sensor.
optics.fluorescence_dark_count(flr, depth[, ...])	Calculates an in situ dark count from the fluorescence sensor.
optics.par_scaling(par_uV, ...)	Scaling correction for par with factory calibration coefficients.
optics.par_fill_surface(par, dives, depth[, ...])	Algebraically calculates the top 5 metres of the par profile.
optics.photic_depth(par, dives, depth[, ...])	Algebraically calculates the euphotic depth.
optics.sunset_sunrise(time, lat, lon)	Calculates the local sunrise/sunset of the glider location.
optics.quenching_correction(flr, bbp, dives, ...)	Corrects the fluorescence data based upon Thomalla et al. (2017).
optics.quenching_report(flr, flr_corrected, ...)	A report for the results of optics.quenching_correction.

11.5.1 glidertools.optics.find_bad_profiles

glidertools.optics.**find_bad_profiles**(*dives*, *depth*, *var*, *ref_depth=None*, *stdev_multiplier=1*, *method='median'*)

Find profiles that exceed a threshold at a reference depth.

This function masks bad dives based on
mean + std x [1] or median + std x [1] at a reference depth.

Function is typically used to clean raw fluorescence and backscatter data.

Parameters

- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.
- **var** (*numpy.ndarray or pandas.Series*) – Array of data variable for function to be performed on.
- **ref_depth** (*int*) – The depth threshold for optics.find_bad_profiles below which the median or mean is calculated for identifying outliers.
- **stdev_multiplier** (*int*) – The standard deviation multiplier for calculating outliers, i.e. mean +- std x [1].
- **method** (*str*) – Whether to use the deep median or deep mean to determine bad profiles for optics.find_bad_profiles.

Returns

- **bad_dive_idx** (*numpy.ndarray or pandas.Series*) – The index of dives where the deep mean/median is greater than the limit
- **bad_dive** (*mask*) – If True, the dive deep mean/median is greater than the limit.

11.5.2 glidertools.optics.par_dark_count

glidertools.optics.**par_dark_count**(*par*, *depth*, *time*, *depth_percentile=90*)

Calculates an in situ dark count from the PAR sensor.

The in situ dark count for the PAR sensor is calculated from the median, selecting only observations in the nighttime and in the 90th percentile of the depth sampled (i.e. the deepest depths measured)

Parameters

- **par** (*numpy.ndarray or pandas.Series*) – The par array after factory calibration in units uE/m2/sec.
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.
- **time** (*numpy.ndarray or pandas.Series*) – The date & time array in a numpy.datetime64 format.
- **depth_percentile** (*int*) – User defined percentile for minimum dark depth. Defaults to 90 so that samples from deepest 10 % of profile are used in correction

Returns
par_dark – The par data corrected for the in situ dark value in units uE/m2/sec.

Return type
numpy.ndarray or pandas.Series

11.5.3 glidertools.optics.backscatter_dark_count

glidertools.optics.**backscatter_dark_count**(*bbp*, *depth*, *percentile=5*)

Calculates an in situ dark count from the backscatter sensor.

The in situ dark count for the backscatter sensor is calculated from the user-defined percentile between 200 and 400m.

Parameters

- **bbp** (*numpy.ndarray or pandas.Series*) – The total backscatter array after factory calibration in m-1.
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.

Returns
bbp – The total backscatter data corrected for the in situ dark value.

Return type
numpy.ndarray or pandas.Series

11.5.4 glidertools.optics.fluorescence_dark_count

glidertools.optics.**fluorescence_dark_count**(*flr*, *depth*, *percentile=5*)

Calculates an in situ dark count from the fluorescence sensor.

The in situ dark count for the fluorescence sensor is calculated from the user-defined percentile between 300 and 400m.

Parameters

- **flr** (*numpy.ndarray or pandas.Series*) – The fluorescence array after factory calibration.
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.

Returns
flr – The fluorescence data corrected for the in situ dark value.

Return type
numpy.ndarray or pandas.Series

11.5.5 glidertools.optics.par_scaling

glidertools.optics.**par_scaling**(*par_uV*, *scale_factor_wet_uEm2s*, *sensor_output_mV*)

Scaling correction for par with factory calibration coefficients.

The function subtracts the sensor output from the raw counts and divides with the scale factor. The factory calibrations are unique for each deployment and should be taken from the calibration file for that deployment.

Parameters

- **par_uV** (*numpy.ndarray or pandas.Series*) – The raw par data with units uV.

- **scale_factor_wet_uEm2s** (*float*) – The scale factor from the factory calibration file in units uE/m2/sec.
- **sensor_output_mV** (*float*) – The sensor output in the dark from the factory calibration file in units mV.
- **Returns**
- **par_uEm2s** (*numpy.ndarray or pandas.Series*) – The par data corrected for the sensor output and scale factor from the factory calibration file in units uE/m2/sec.

11.5.6 glidertools.optics.par_fill_surface

glidertools.optics.**par_fill_surface**(*par*, *dives*, *depth*, *max_curve_depth=100*)

Algebraically calculates the top 5 metres of the par profile.

The function removes the top 5 metres of par data, and then using an exponential equation calculates the complete profile.

Parameters

- **par** (*numpy.ndarray or pandas.Series*) – The par data with units uE/m2/sec.
- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.
- **max_curve_depth** (*int*) – The maximum depth of which to fit the exponential function.

Returns
par_filled – The par data with the algebraically calculated top 5 metres.

Return type
numpy.ndarray or pandas.Series

11.5.7 glidertools.optics.photic_depth

glidertools.optics.**photic_depth**(*par*, *dives*, *depth*, *return_mask=False*, *ref_percentage=1*)

Algebraically calculates the euphotic depth.

The function calculates the euphotic depth and attenuation coefficient (Kd) based upon the linear fit of the natural log of par with depth.

Parameters

- **par** (*numpy.ndarray or pandas.Series*) – The par data with units uE/m2/sec.
- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.
- **return_mask** (*bool*) – If True, will return a mask for the photic layer (depth < euphotic depth).
- **ref_percentage** (*int*) – The percentage light depth to calculate the euphotic layer, typically assumed to be 1% of surface par.

Returns

- **light_depths** (*numpy.ndarray*) – An array of the euphotic depths in metres.
- **slopes** (*numpy.ndarray*) – An array of the par attenuation coefficient (Kd).

11.5.8 glidertools.optics.sunset_sunrise

glidertools.optics.**sunset_sunrise**(*time*, *lat*, *lon*)

Calculates the local sunrise/sunset of the glider location.

The function uses the Skyfield package to calculate the sunrise and sunset times using the date, latitude and longitude. The times are returned rather than day or night indices, as it is more flexible for the quenching correction.

Parameters

- **time** (*numpy.ndarray or pandas.Series*) – The date & time array in a numpy.datetime64 format.
- **lat** (*numpy.ndarray or pandas.Series*) – The latitude of the glider position.
- **lon** (*numpy.ndarray or pandas.Series*) – The longitude of the glider position.

Returns

- **sunrise** (*numpy.ndarray*) – An array of the sunrise times.
- **sunset** (*numpy.ndarray*) – An array of the sunset times.

11.5.9 glidertools.optics.quenching_correction

glidertools.optics.**quenching_correction**(*flr*, *bbp*, *dives*, *depth*, *time*, *lat*, *lon*, *max_photic_depth=100*, *night_day_group=True*, *surface_layer=5*, *sunrise_sunset_offset=1*)

Corrects the fluorescence data based upon Thomalla et al. (2017).

The function calculates the quenching depth and performs the quenching correction based on the fluorescence to backscatter ratio. The quenching depth is calculated based upon the different between night and daytime fluorescence. The default setting is for the preceding night to be used to correct the following day's quenching (*night_day_group=True*). This can be changed so that the following night is used to correct the preceding day. The quenching depth is then found from the difference between the night and daytime fluorescence, using the steepest gradient of the {5 minimum differences and the points the difference changes sign (+ve/-ve)}. The function gets the backscatter/fluorescence ratio between from the quenching depth to the surface, and then calculates a mean nighttime ratio for each night. The quenching ratio is calculated from the nighttime ratio and the daytime ratio, which is then applied to fluorescence to correct for quenching. If the corrected value is less than raw, then the function will return the original raw data.

Parameters

- **flr** (*numpy.ndarray or pandas.Series*) – fluorescence data after cleaning and factory calibration conversion
- **bbp** (*numpy.ndarray or pandas.Series*) – Total backscatter after cleaning and factory calibration conversion
- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.

- **time** (*numpy.ndarray or pandas.Series*) – The date & time array in a numpy.datetime64 format.
- **lat** (*numpy.ndarray or pandas.Series*) – The latitude of the glider position.
- **lon** (*numpy.ndarray or pandas.Series*) – The longitude of the glider position.
- **max_photic_depth** (*int*) – Limit the quenching correction to depth less than a given value [100].
- **night_day_group** (*bool*) – If True, use preceding night otherwise use following night for calculating the flr:bbp ratio.
- **surface_layer** (*int*) – The surface depth that is omitted from the correction calculations (metres)
- **sunrise_sunset_offset** (*int*) – The delayed onset and recovery of quenching in hours [1] (assumes symmetrical).

Returns

- **flr_corrected** (*numpy.ndarray or pandas.Series*) – The fluorescence data corrected for quenching.
- **quenching layer** (*bool*) – A boolean mask of where the fluorescence is quenched.

11.5.10 glidertools.optics.quenching_report

glidertools.optics.**quenching_report**(*flr*, *flr_corrected*, *quenching_layer*, *dives*, *depth*, *pcolor_kwargs={}*)

A report for the results of optics.quenching_correction.

The function creates a figure object of 3 subplots containing a pcolormesh of the original fluorescence data, the quenching corrected fluorescence data and the quenching layer calculated from the optics.quenching_correction function.

Parameters

- **flr** (*numpy.ndarray or pandas.Series*) – Fluorescence data after cleaning and factory calibration conversion.
- **flr_corrected** (*numpy.ndarray or pandas.Series*) – The fluorescence data corrected for quenching.
- **layer** (*quenching*) – A boolean mask of where the fluorescence is quenched.
- **dives** (*numpy.ndarray or pandas.Series*) – The dive count (round is down dives, 0.5 is up dives).
- **depth** (*numpy.ndarray or pandas.Series*) – The depth array in metres.
- **pcolor_kwargs** (*dict*) – A dictionary of keyword arguements passed to pcolormesh.

Returns

figure – Creates a figure object with 3 subplots containing a pcolormesh of the original fluorescence data, the quenching corrected fluorescence data and the quenching layer calculated from the optics.quenching_correction function.

Return type

object

11.6 Calibration

calibration.bottle_matchup(gld_dives, ...[, ...])	Performs a matchup between glider and bottle samples based on time and depth (or density).
calibration.model_figs(bottle_data, ...[, ax])	Creates the figure for a linear model fit.
calibration.robust_linear_fit(gld_var, ...)	Perform a robust linear regression using a Huber Loss Function to remove outliers.

11.6.1 glidertools.calibration.bottle_matchup

glidertools.calibration.**bottle_matchup**(*gld_dives*, *gld_depth*, *gld_time*, *btl_depth*, *btl_time*, *btl_values*, *min_depth_diff_metres=5*, *min_time_diff_minutes=120*)

Performs a matchup between glider and bottle samples based on time and depth (or density).

Parameters

- **gld_depth** (*np.array, dtype=float*) – glider depth at time of measurement
- **gld_dives** (*np.array, dtype=float*) – dive index of the glider (given by glider toolbox)
- **gld_time** (*np.array, dtype=datetime64*) – glider time that will be used as primary indexing variable
- **btl_time** (*np.array, dtype=datetime64*) – in-situ bottle sample's time
- **btl_depth** (*np.array, dtype=float*) – depth of in-situ sample
- **btl_values** (*np.array, dtype=float*) – the value that will be interpolated onto the glider time and depth coordinates (time, depth/dens)
- **min_depth_diff_metres** (*float, default=5*) – the minimum allowable depth difference
- **min_time_diff_minutes** (*float, default=120*) – the minimum allowable time difference between bottles and glider

Returns

array – Returns the bottle values in the format of the glider i.e. the length of the output will be the same as gld_*

Return type

float

11.6.2 glidertools.calibration.model_figs

glidertools.calibration.**model_figs**(*bottle_data*, *glider_data*, *model*, *ax=None*)

Creates the figure for a linear model fit.

Parameters

- **bottle_data** (*np.array, shape=[m,]*) – bottle data with the number of matched bottle/glider samples
- **glider_data** (*np.array, shape[m,]*) – glider data with the number of matched bottle/glider samples
- **model** (*sklearn.linear_model object*) – a fitted model that you want to test.

Returns
figure axes – A figure showing the fit of the

Return type
matplotlib.Axes

11.6.3 glidertools.calibration.robust_linear_fit

glidertools.calibration.**robust_linear_fit**(*gld_var*, *gld_var_cal*, *interpolate_limit=3*, *return_figures=True*, ***kwargs*)

Perform a robust linear regression using a Huber Loss Function to remove outliers. Returns a model object that behaves like a scikit-learn model object with a model.predict method.

Parameters

- **gld_var** (*np.array, shape=[n,]*) – glider variable
- **gld_var_cal** (*np.array, shape=[n,]*) – bottle variable on glider indicies
- **fit_intercept** (*bool, default=False*) – forces 0 intercept if False
- **return_figures** (*bool, default=True*) – create figure with metrics
- **interpolate_limit** (*int, default=3*) – glider data may have missing points. The glider data is thus interpolated to ensure that as many bottle samples as possible have a match-up with the glider.
- ****kwargs** (*keyword=value pairs*) – will be passed to the Huber Loss regression to adjust regression

Returns
model – A fitted model. Use model.predict(glider_var) to create the calibrated output.

Return type
sklearn.linear_model

11.7 Gridding and Interpolation

mapping.interp_obj(x, y, z, xi, yi[, ...])	Performs objective interpolation (or Kriging) of a 2D field.
mapping.grid_data(x, y, var[, bins, how, ...])	Grids the input variable to bins for depth/dens (y) and time/dive (x).
mapping.variogram(variable, horz, vert, dives)	Find the interpolation parameters and x and y scaling of the horizontal and vertical coordinate paramters for objective interpolation (Kriging).

11.7.1 glidertools.mapping.interp_obj

glidertools.mapping.**interp_obj**(*x*, *y*, *z*, *xi*, *yi*, *partial_sill=0.1*, *nugget=0.01*, *lenscale_x=20*, *lenscale_y=20*, *detrend=True*, *max_points_per_quad=55*, *min_points_per_quad=8*, *return_error=False*, *n_cpus=None*, *verbose=True*, *parallel_chunk_size=512*)

Performs objective interpolation (or Kriging) of a 2D field.

The objective interpolation breaks the problem into smaller fields by iteratively breaking the problem into quadrants. Each quadrant is then interpolated (also using intformation from its neighbours). The interpolation is inverse distance weighted using a gaussian kernel (or radial basis function). The kernel has a width of 12 hours if the x-dimension is time, otherwise scaled by the x-variable unit. The kernel is in meters assuming that depth is the y-coord. This can be changed with keyword arguements. An error estimate can also be calculated if requested.

The following link provides good background on the Kriging procedure: http://desktop.arcgis.com/en/arcmap/10.3/tools/3d-analyst-toolbox/how-kriging-works.htm

Parameters

- **x** (*np.array | pd.series*) – horizontal coordinates of the input data (same length as y, z) can be types float or datetime64
- **y** (*np.array | pd.series*) – vertical coordinates of the input data (same length as x, z)
- **z** (*np.array | pd.series*) – values to be interoplated (same length as x, y)
- **xi** (*np.array*) – horizontal coordinates of the interpolation grid (must be 1D) can be types float or datetime64
- **yi** (*np.array | pd.series*) – vertical coordinates of the interpolation grid (must be 1D)
- **nugget** (*float [0.01]*) – the error estimate due to sampling inaccuracy also known as the nugget in Kriging literature. This should be taken from the semivariogram
- **partial_sill** (*float [0.1]*) – represents the spatial covariance of the variable being interpolated. Should be estimated from the semivariogram. See Kriging literature for more information
- **lenscale_x** (*float [20]*) – horizontal length scale horizontal coordinate variable If dtype(x) is np.datetime64 (any format) then lenscale units is in hours. Otherwise if type(x).
- **lenscale_y** (*float [20]*) – horizontal length scale horizontal coordinate variable.
- **max_points_per_quad** (*int [55]*) – the data is divided into quadrants using a quadtree approach - iteratively dividing data into smaller quadrants using x and y coordinates. The algorithm stops splitting the data into quadrants when there are no quadrants exceeding the limit set with max_points_per_quad is. This is done to reduce the computational cost of the function.
- **min_points_per_quad** (*int [8]*) – sets the minimum number of points allowed in a neighbouring quadrant when creating the interpolation function for a particular quadrant. If the number of points is less than specified, the algorithm looks for neighbours of the neighbours to include more points in the interpolation.
- **n_cpus** (*int [n - 1]*) – use parallel computing. The quadrant calculations are spread across CPUs. Must be positive and > 0
- **parallel_chunk_size** (*int [512]*) – the number of leaves that will be processed in parallel in one go. This is a memory saving feature. If your dataset is very large, parallel processing will use up a lot of memmory. Increasing the chunk size increases the memory requirements.

- **verbose** (*bool [True]*) – will print out information about the interpolation

Returns
Contains the following arrays: - z: interpolated values - variance: error estimate of the interpolation - weights: the quadtree weighting used to calculate the estimates - nugget: the nugget used in the interpolation - partial_sill: value used for the interpolation

Return type
xr.Dataset

Note: The data may have semi-discrete artifacts. This is also present in the MATLAB output.

Example

```
>>> xi = np.arange(time.values.min(), time.values.max(), 30,
                   dtype='datetime64[m]')
>>> yi = np.linspace(depth.min(), depth.max(), 1.)
>>> interpolated = gt.mapping.interp_obj(
        time, depth, var, xi, yi,
        nugget=.0035, partial_sill=0.02,
        lenscale_x=80, lenscale_y=80,
        detrend=True)
```

11.7.2 glidertools.mapping.grid_data

glidertools.mapping.**grid_data**(*x*, *y*, *var*, *bins=None*, *how='mean'*, *interp_lim=6*, *verbose=True*, *return_xarray=True*)

Grids the input variable to bins for depth/dens (y) and time/dive (x). The bins can be specified to be non-uniform to adapt to variable sampling intervals of the profile. It is useful to use the `gt.plot.bin_size` function to identify the sampling intervals. The bins are averaged (mean) by default but can also be the `median, std, count`,

Parameters

- **x** (*np.array, dtype=float, shape=[n,]*) – The horizontal values by which to bin need to be in a psudeo discrete format already. Dive number or `time_average_per_dive` are the standard inputs for this variable. Has `p` unique values.
- **y** (*np.array, dtype=float, shape=[n,]*) – The vertical values that will be binned; typically depth, but can also be density or any other variable.
- **bins** (*np.array, dtype=float; shape=[q,], default=[0 : 1 : max_depth]*) – Define the bin edges for y with this function. If not defined, defaults to one meter bins.
- **how** (*str, defualt='mean'*) – the string form of a function that can be applied to pandas.Groupby objects. These include `mean, median, std, count`.
- **interp_lim** (*int, default=6*) – sets the maximum extent to which NaNs will be filled.

Returns
glider_section – A 2D section in the format specified by `ax_xarray` input.

Return type
xarray.DataArray, shape=[p, q]

Raises

Userwarning – Triggers when x does not have discrete values.

11.7.3 glidertools.mapping.variogram

glidertools.mapping.**variogram**(*variable*, *horz*, *vert*, *dives*, *mask=None*, *xy_ratio=1*, *max_points=5000*, *ax=True*)

Find the interpolation parameters and x and y scaling of the horizontal and vertical coordinate paramters for objective interpolation (Kriging).

The range of the variogram will automatically be scaled to 1 and the x and y length scales will be given in the output. This can be used in the gt.mapping.interp_obj function.

Parameters

- **variable** (*array-like*) – target variable for interpolation as a flat array
- **horz** (*array-like*) – the horizontal coordinate variable as a flat array. Can be time in np.datetime64 format, or any other float value
- **vert** (*array-like*) – the vertical coordinate variable as a flat array. Usually depth, but can be pessure or any other variable.
- **dives** (*array-like*) – the dive numbers as a flat array
- **mask** (*array-like*) – a boolean array that can be used to retain certain regions, e.g. depth < 250
- **xy_ratio** (*float*) – determines the anisotropy of the coordinate variables. The value can be changed iteritively to improve the shape of the semivariance curve. The range of the variogram will automatically be scaled to 1. The x and y length scales are given in the output.
- **max_points** (*int*) – maximum number of points that will be used in the variogram. The function selects a subset of dives rather than random points to be consistent in the vertical. Increasing this number will increase the accuracy of the length scales.
- **ax** (*bool or mpl.Axes*) – If True, will automatically create an axes, if an axes object is given, returns the plot in those axes.

Returns

- **variogram_params** (*dict*) – a dictionary containing the information required by the gt.mapping.interp_obj function.
- **plot** (*axes*) – a axes object containing the plot of the semivariogram

Example

```
>>> gt.mapping.variogram(var, time, depth, dives, mask=depth<350,
                         xy_ratio=0.5, max_points=6000)
```

11.8 Plotting

plot.plot_functions(**kwargs)	Plot data (gridded or not) as a section and more.

11.8.1 glidertools.plot.plot_functions

class glidertools.plot.**plot_functions**(***kwargs*)

Plot data (gridded or not) as a section and more.

This function provides several options to plot data as a section. The default action when called is to plot data as a `pcolormesh` section.

See the individual method help for more information about each plotting method.

Parameters

- **args** (*array_like*) –
  - same length x, y, z. Will be gridded with depth of 1 meter.
  - x(m), y(n), z(n, m) arrays
  - z DataFrame where indicies are depth and columns are dives
  - **z DataArray where dim0 is dives and dim1 is depth, or** contains information about time and depth axes
- **kwargs** (*key-value pairs*) –
  - ax - give an axes to the plotting function
  - robust - use the 0.5 and 99.5 percentile to set color limits
  - gridding_dz - gridding depth [default 1]

__init__()

Methods

__init__()	
bin_size(depth[, bins, ax, add_colorbar])	Plots a 2D histogram of the depth sampling frequency.
contourf(*args, **kwargs)	Plot a section plot of the dives with x-time and y-depth and z-variable.
pcolormesh(*args, **kwargs)	Plot a section plot of the dives with x-time and y-depth and z-variable.
save_figures_to_pdf(fig_list, pdf_name, ...)	Saves a list of figure objects to a pdf.
scatter(x, y, z[, ax, robust])	Plot a scatter section plot of a small dataset (< 10 000 obs)
section3D(dives, depth, x, y, variable[, ...])	Returns an interactive 3D plot in an HTML page.

11.9 General Utilities

utils.time_average_per_dive(dives, time)	Gets the average time stamp per dive.
utils.mask_above_depth(ds, depths)	Masks all data above depths.
utils.mask_below_depth(ds, depths)	Masks all data below depths.
utils.mask_profile_depth(df, mask_depth, above)	Masks either above or below mask_depth.
utils.merge_dimensions(df1, df2[, interp_lim])	Merges variables measured at different time intervals.
utils.calc_glider_vert_velocity(time, depth)	Calculate glider vertical velocity in cm/s
utils.calc_dive_phase(time, depth[, ...])	Determine the glider dive phase
utils.calc_dive_number(time, depth[, ...])	Determine the glider dive number (based on dive phase)
utils.dive_phase_to_number(phase)	
utils.distance(lon, lat[, ref_idx])	Great-circle distance in m between lon, lat points.
utils.group_by_profiles(ds[, variables])	Group profiles by dives column.

11.9.1 glidertools.utils.time_average_per_dive

glidertools.utils.**time_average_per_dive**(*dives*, *time*)

Gets the average time stamp per dive. This is used to create psuedo discrete time steps per dive for plotting data (using time as x-axis variable).

Parameters

- **dives** (*np.array, dtype=float, shape=[n,]*) – discrete dive numbers (down = d.0; up = d.5) that matches time length
- **time** (*np.array, dtype=datetime64, shape=[n,]*) – time stamp for each observed measurement

Returns

time_average_per_dive – each dive will have the average time stamp of that dive. Can be used for plotting where time_average_per_dive is set as the x-axis.

Return type

np.array, dtype=datetime64, shape=[n,]

11.9.2 glidertools.utils.mask_above_depth

glidertools.utils.**mask_above_depth**(*ds*, *depths*)

Masks all data above depths.

Parameters

- **df** (*xarray.Dataframe or pandas.Dataframe*)
- **mask_depths** (*float (for constant depth masking) or pandas.Series as*) – returned e.g. by the mixed_layer_depth function

11.9.3 glidertools.utils.mask_below_depth

glidertools.utils.**mask_below_depth**(*ds*, *depths*)

Masks all data below depths.

Parameters

- **df** (*xarray.Dataframe or pandas.Dataframe*)
- **mask_depths** (*float (for constant depth masking) or pandas.Series as*) – returned e.g. by the mixed_layer_depth function

11.9.4 glidertools.utils.mask_profile_depth

glidertools.utils.**mask_profile_depth**(*df*, *mask_depth*, *above*)

Masks either above or below mask_depth. If type(mask_depth)=np.nan, the whole profile will be masked. Warning: This function is for a SINGLE profile only, for masking a complete Glider Dataset please look for utils.mask_above_depth and/or utils.mask_below_depth.

Parameters

- **df** (*xarray.Dataframe or pandas.Dataframe*)
- **mask_depths** (*float (for constant depth masking) or pandas.Series as*) – returned e.g. by the mixed_layer_depth function
- **above** (*boolean*) – Mask either above mask_depth (True) or below (False)

11.9.5 glidertools.utils.merge_dimensions

glidertools.utils.**merge_dimensions**(*df1*, *df2*, *interp_lim=3*)

Merges variables measured at different time intervals. Glider data may be sampled at different time intervals, as is the case for primary CTD and SciCon data.

Parameters

- **df1** (*pandas.DataFrame*) – A dataframe indexed by datetime64 sampling times. Can have multiple columns. The index of this first dataframe will be preserved.
- **df2** (*pandas.DataFrame*) – A dataframe indexed by datetime64 sampling times. Can have multiple columns. This second dataframe will be interpolated linearly onto the first dataframe.

Returns

merged_df – The combined arrays interpolated onto the index of the first axis

Return type

pandas.DataFrame

Raises

Userwarning – If either one of the indicies are not datetime64 dtypes

Example

You can use the following code and alter it if you want more control

```
>>> df = pd.concat([df1, df2], sort=True, join='outer')
>>> df = (df
          .sort_index()
          .interpolate(limit=interp_lim)
          .bfill(limit=interp_lim)
          .loc[df1.index]
      )
```

11.9.6 glidertools.utils.calc_glider_vert_velocity

glidertools.utils.**calc_glider_vert_velocity**(*time*, *depth*)

Calculate glider vertical velocity in cm/s

Parameters

- **time** (*np.array [datetime64]*) – glider time dimension
- **depth** (*np.array [float]*) – depth (m) or pressure (dbar) if depth not avail

Returns

velocity – vertical velocity in cm/s

Return type

np.array

11.9.7 glidertools.utils.calc_dive_phase

glidertools.utils.**calc_dive_phase**(*time*, *depth*, *dive_depth_threshold=15*)

Determine the glider dive phase

Parameters

- **time** (*np.array [datetime64]*) – glider time dimension
- **depth** (*np.array [float]*) – depth (m) or pressure (dbar) if depth not avail
- **dive_depth_threshold** (*[float]*) – minimum dive depth (m or dbar), should be less than your most shallow dive

Returns

phase – phase according to the EGO dive phases

Return type

np.array [int]

11.9.8 glidertools.utils.calc_dive_number

glidertools.utils.**calc_dive_number**(*time*, *depth*, *dive_depth_threshold=15*)

Determine the glider dive number (based on dive phase)

Parameters

- **time** (*np.array [datetime64]*) – glider time dimension
- **depth** (*np.array [float]*) – depth (m) or pressure (dbar) if depth not avail
- **dive_depth_threshold** (*[float]*) – minimum dive depth (m or dbar), should be less than your most shallow dive

Returns

dive_number – the dive number where down dives are x.0 and up dives are x.5

Return type

np.ndarray [float]

11.9.9 glidertools.utils.dive_phase_to_number

glidertools.utils.**dive_phase_to_number**(*phase*)

11.9.10 glidertools.utils.distance

glidertools.utils.**distance**(*lon*, *lat*, *ref_idx=None*)

Great-circle distance in m between lon, lat points.

Parameters

- **lon** (*array-like, 1-D (size must match)*) – Longitude, latitude, in degrees.
- **lat** (*array-like, 1-D (size must match)*) – Longitude, latitude, in degrees.
- **ref_idx** (*None, int*) – Defaults to None, which gives adjacent distances. If set to positive or negative integer, distances will be calculated from that point

Returns

distance – distance in meters between adjacent points or distance from reference point

Return type

array-like

11.9.11 glidertools.utils.group_by_profiles

glidertools.utils.**group_by_profiles**(*ds*, *variables=None*)

Group profiles by dives column. Each group member is one dive. The returned profiles can be evaluated statistically, e.g. by pandas.DataFrame.mean or other aggregating methods. To filter out one specific profile, use xarray.Dataset.where instead.

Parameters

- **ds** (*xarray.Dataset*) – 1-dimensional Glider dataset
- **variables** (*list of strings, optional*) – specify variables if only a subset of the dataset should be grouped into profiles. Grouping only a subset is considerably faster and more memory-effective.

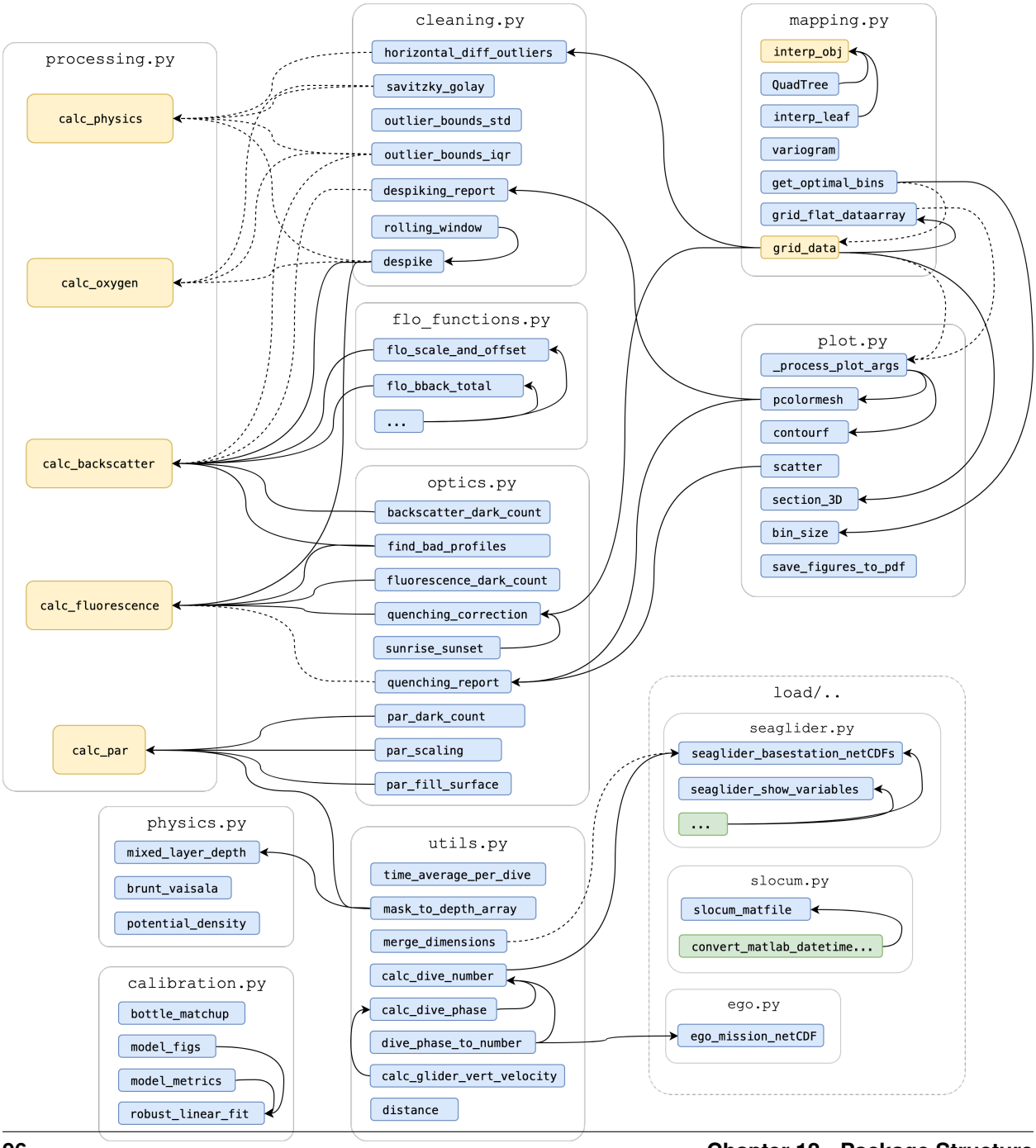
Returns

- *profiles*
- *dataset grouped by profiles (dives variable), as created by the*
- *pandas.groupby methods.*

CHAPTER

TWELVE

PACKAGE STRUCTURE



CHAPTER

THIRTEEN

WHAT'S NEW

13.1 v2023.07.25 (2023/07/25)

13.1.1 New Features

- added import for VOTO seaexplorer data (GH#170) By Martin Mohrmann.
- added versatile, depth dependent masking (GH#172) and per profile grouping (GH#175). By Martin Mohrmann.
- add concatenation of two or more datasets (GH#173), even with different set of variables (GH#183). By Martin Mohrmann.

13.1.2 Breaking changes

- Changed the behavior of *find_dive_phase* and *calc_dive_number* to use a smaller depth threshold when determining a valid dive (15 dbar down from 200 dbar). this is also now adjusteable. (GH#134) By Tom Hull.
- GliderTools defaults for Figure creation were changed. Automatic application of plt.tight_layout was dropped in favour of more flexible embedding of GliderTools plots into existing layouts/subplots. (GH#185). By Martin Mohrmann.
- The mixed layer depth algorithm was corrected. (GH#169, GH#168). By Martin Mohrmann. API change! Existing mixed layer computation code must be adapted.

13.1.3 Internal changes

- Removed outdated python-seawater dependency (GH#186). By Callum Rollo.
- Update documentation of required dependencies (GH#174). By Sören Thomsen.
- Some cleanup of old python2 dependencies (GH#166). By Martin Mohrmann.
- Replace deprecated pkg_resources with importlib.metadata (GH#187). By Martin Mohrmann.
- Add release guide to documentation (GH#186). By Martin Mohrmann.
- Cleanup of unused imports (GH#174). By Martin Mohrmann.

13.1.4 Bug fixes

- Adapt demo notebook to updated Glider Tools (GH#179). By Callum Rollo.
- Fix netCDF attribute handling for non-string attributes (GH#194). By Martin Mohrmann.
- Adapt quenching_report to modern numpy versions (GH#191) By Martin Mohrmann.
- Improve error handling for MLD computation (GH#190). By Martin Mohrmann.

Thanks also to Julius Busecke for help with the github CI, Sam Woodman for detailed bug reports and everyone else who has contributed.

13.2 v2022.12.13 (2022/12/13)

13.2.1 Internal changes

- Refactoring and update of testing and development framework, update of flake, black and almost all python dependencies

13.2.2 Breaking changes

- Fixed processing/calc_oxygen (:pull: *116*, :issue: *112*) By Callum Rollo.

13.2.3 Internal Changes

- Implemented code linting as part of the CI (GH#100) By Julius Busecke.

13.2.4 Documentation

- Added conda installation instructions + badge. (GH#94) By Julius Busecke.

13.2.5 Bug fixes

- Replaced *skyfield* dependency with *astral*, fixing sunrise/sunset problems at high latitudes. By Isabelle Sindiswa Giddy.

13.3 v2021.03 (2021/3/30)

13.3.1 Documentation

- Updated contributor guide for conda based workflow. (GH#81) By Julius Busecke.

13.3.2 Internal Changes

- Migration of CI to conda based workflow with multiple python versions. (GH#54) By Julius Busecke.
- Revamp distribution actions. (GH#82) By Julius Busecke.
- Migrate from astral to skyfield (:pull:'121') By 'Isabelle Giddy <https://github.com/isgiddy>'_.

CHAPTER

FOURTEEN

CITING GLIDERTOOLS

If you would like to cite or reference Glider Tools, please use:

Gregor, L., Ryan-Keogh, T. J., Nicholson, S.-A., du Plessis, M., Giddy, I., & Swart, S. (2019). GliderTools: A Python Toolbox for Processing Underwater Glider Data. Frontiers in Marine Science, 6(December), 1–13. https://doi.org/10.3389/fmars.2019.00738

14.1 Project Contributors

The following people have made contributions to the project (in alphabetical order by last name) and are considered "The GliderTools Developers". These contributors will be added as authors upon the next major release of GliderTools (i.e. new DOI release).

- Dhruv Balwada - University of Washington, USA. (ORCID: 0000-0001-6632-0187)
- Julius Busecke - Columbia University, USA. (ORCID: 0000-0001-8571-865X)
- Isabelle Giddy - University of Cape Town: Cape Town, Western Cape, South Africa. (ORCID: 0000-0002-8926-3311)
- Luke Gregor - Environmental Physics, ETH Zuerich: Zurich, Switzerland. (ORCID: 0000-0001-6071-1857)
- Tom Hull - Centre for Environment Fisheries and Aquaculture Science: Lowestoft, UK. (ORCID: 0000-0002-1714-9317)
- Martin Mohrmann - Voice of the Ocean Foundation, Gothenburg, Sweden. (ORCID: 0000-0001-8056-4866)
- Sarah-Anne Nicholson - Council for Scientific and Industrial Research: Cape Town, South Africa. (ORCID: 0000-0002-1226-1828)
- Marcel du Plessis - University of Cape Town: Cape Town, Western Cape, South Africa. (ORCID: 0000-0003-2759-2467)
- Callum Rollo - Voice of the Ocean Foundation, Gothenburg, Sweden. (ORCID: 0000-0002-5134-7886)
- Tommy Ryan-Keogh - Council for Scientific and Industrial Research: Cape Town, South Africa. (ORCID: 0000-0001-5144-171X)
- Sebastiaan Swart - University of Gothenburg: Gothenburg, Sweden. (ORCID: 0000-0002-2251-8826)
- Soeren Thomsen - LOCEAN/IPSL/CNRS/Sorbonne University: Paris, France. (ORCID: 0000-0002-0598-8340)

CHAPTER

FIFTEEN

CONTRIBUTION GUIDE

Contributions are highly welcomed and appreciated. Every little help counts, so do not hesitate! You can make a high impact on `glidertools` just by using it, being involved in discussions

and reporting issues.

The following sections cover some general guidelines regarding development in `glidertools` for maintainers and contributors.

Nothing here is set in stone and can't be changed. Feel free to suggest improvements or changes in the workflow.

Contribution links

15.1 Feature requests and feedback

We are eager to hear about your requests for new features and any suggestions about the API, infrastructure, and so on. Feel free to start a discussion about these on the discussions tab on github under the "ideas" section.

After discussion with a few community members, and agreement that the feature should be added and who will work on it, a new issue should be opened. In the issue, please make sure to explain in detail how the feature should work and keep the scope as narrow as possible. This will make it easier to implement in small PRs.

15.2 Report bugs

Report bugs for `glidertools` in the issue tracker with the label "bug".

If you can write a demonstration test that currently fails but should pass that is a very useful commit to make as well, even if you cannot fix the bug itself.

15.3 Fix bugs

Look through the GitHub issues for bugs.

Talk to developers to find out how you can fix specific bugs.

15.4 Preparing Pull Requests

1. Fork the glidertools GitHub repository. It's fine to use `glidertools` as your fork repository name because it will live under your username.
2. Clone your fork locally using git, connect your repository to the upstream (main project), and create a branch:

```
$ git clone git@github.com:YOUR_GITHUB_USERNAME/glidertools.git # clone to local␣
↪machine
$ cd glidertools
$ git remote add upstream git@github.com:GliderToolsCommunity/GliderTools.git #␣
↪connect to upstream remote

# now, to fix a bug or add feature create your own branch off "master":

$ git checkout -b your-bugfix-feature-branch-name master # Create a new branch␣
↪where you will make changes
```

   If you need some help with Git, follow this quick start guide: https://git.wiki.kernel.org/index.php/QuickStart

3. Set up a [conda](environment) with all necessary dependencies:

```
$ conda env create -f ci/environment-py3.8.yml
```

4. Activate your environment:

```
$ conda activate test_env_glidertools
```

   Make sure you are in this environment when working on changes in the future too.

5. Install the GliderTools package:

```
$ pip install -e . --no-deps
```

6. Before you modify anything, ensure that the setup works by executing all tests:

```
$ pytest
```

   You want to see an output indicating no failures, like this:

```
$ ========================== n passed, j warnings in 17.07s␣
↪============================
```

7. Install pre-commit and its hook on the `glidertools` repo:

```
$ pip install --user pre-commit
$ pre-commit install
```

Afterwards `pre-commit` will run whenever you commit. If some errors are reported by pre-commit you should format the code by running:

```
$ pre-commit run --all-files
```

and then try to commit again.

https://pre-commit.com/ is a framework for managing and maintaining multi-language pre-commit hooks to ensure code-style and code formatting is consistent.

> You can now edit your local working copy and run/add tests as necessary. Please follow PEP-8 for naming. When committing, `pre-commit` will modify the files as needed, or will generally be quite clear about what you need to do to pass the commit test.

8. Break your edits up into reasonably sized commits:

```
$ git commit -a -m "<commit message>"
$ git push -u
```

Committing will run the pre-commit hooks (isort, black and flake8). Pushing will run the pre-push hooks (pytest and coverage)

We highly recommend using test driven development, but our coverage requirement is low at the moment due to lack of tests. If you are able to write tests, please stick to xarray's testing recommendations.

9. **Add yourself to the**
   Project Contributors list via `./docs/authors.md`.

10. Finally, submit a pull request (PR) through the GitHub website using this data:

```
head-fork: YOUR_GITHUB_USERNAME/glidertools
compare: your-branch-name

base-fork: GliderToolsCommunity/GliderTools
base: master
```

The merged pull request will undergo the same testing that your local branch had to pass when pushing.

11. After your pull request is merged into the *GliderTools/master*, you will need to fetch those changes and rebase your master so that your master reflects the latest version of GliderTools. The changes should be fetched and incorporated (rebase) also right before you are planning to introduce changes.:

```
$ git checkout master # switch back to master branch
$ git fetch upstream  # Download all changes from central upstream repo
$ git rebase upstream/master  # Apply the changes that have been made to central␣
↪repo,
$ # since your last fetch, onto you master.
$ git branch -d your-bugfix-feature-branch-name  # to delete the branch after PR is␣
↪approved
```

15.5 Release Instructions

This is a documentation repo for people in the group on how to do the integrated deployment.

NB RULE! Never commit to master.

1. Change the version in the setup.py file. Must be format YYYY.<release number>
2. Create a release with a tag that has the same format as the version above.
3. The distribution will be built automatically and pushed to PyPi
4. The DOI will also be updated on Zenodo. (untested, see #165)

CHAPTER

SIXTEEN

WISHLIST

A list of things we'd love to add to GliderTools and the work involved.

1. Support for raw files from Slocum gliders and Seagliders with the following additional functionality
    - Thermal lag correction for each of the gliders supported in the suggestion above.
    - Support for hadware modules by model and manufacturer
2. Make final data output compatible with www.OceanGliders.org data format, https://www.oceangliders.org/taskteams/data-management/

INDEX

Symbols

B

C

D

E

F

G

H

I

M

O

P

Q