ctd-processing

January 24, 2024

1 Sea-Bird Scientific Community Toolkit - CTD Processing (0.0.2)

1.1 Introduction

This python notebook has been written to give Sea-Bird Scientific (SBS) customers and the scientific community at large a set of python code to be used to process and analyze data from the CTD family of SBS instruments. This toolkit was made in conjunction with the Fathom project. Great care has been taken to make sure that the python code that makes up this toolkit and its related packages gives the same results as the Fathom application.

Please contact SBS customer support for help or to request additional features.

1.2 Python Data Processing and Visualization

- Data Conversion
- Low Pass Filter
- Align CTD
- Cell Thermal Mass
- Loop Edit
- Derive TEOS-10
- Wild Edit
- Window Filter

1.3 Init

The initialization code section below is used to import required libraries and to make a helper function that is used to make scatter charts with the Plotly package.

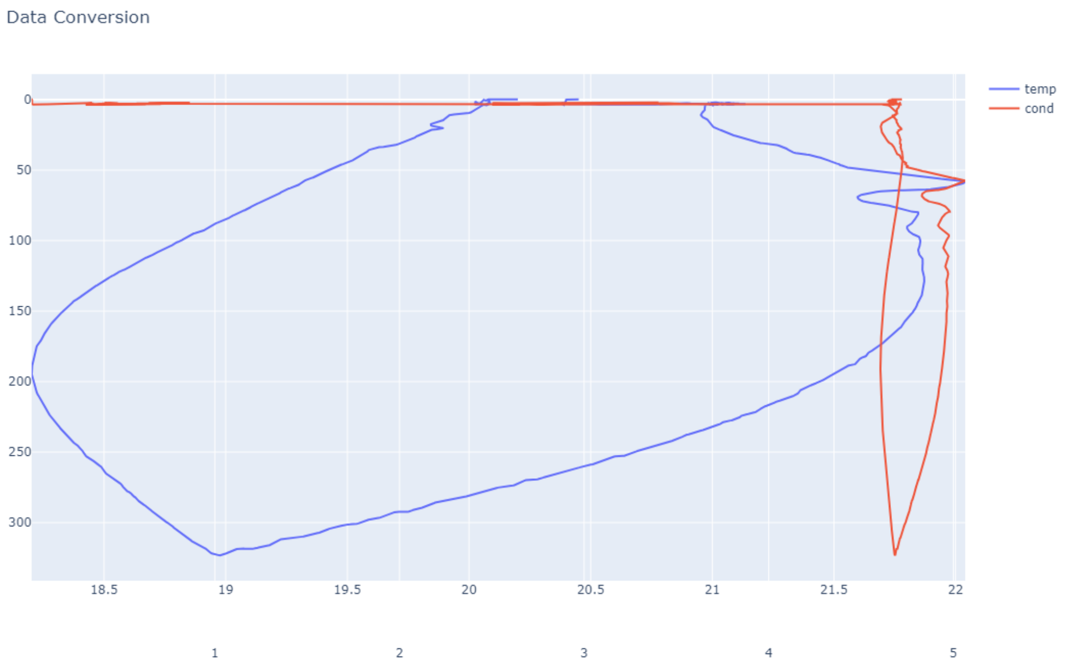
[11]:
```python
# Native imports
import os.path

# Third-party imports
import gsw
import numpy as np
import pandas as pd

# Sea-Bird imports
import sbs.process.cal_coefficients as cal
import sbs.process.conversion as conv
import sbs.process.processing as proc
import sbs.process.instrument_data as id
import sbs.visualize as viz


def plot(
    data, title, x_names, x_bounds={0: [17, 24], 1: [-1, 6], 2: [-1, 6], 3:␣
 ↪[-1, 6]}
):
    config = viz.ChartConfig(
        title=title,
        x_names=x_names,
        y_names=["press"],
        z_names=[],
        chart_type="overlay",
        bounds={
            "x": x_bounds,
            # 'y': { 0: [29, 32], 1: [29, 32], 2: [29, 32], 3: [29, 32] }
        },
        plot_loop_edit_flags=False,
        lift_pen_over_bad_data=True,
    )

    chart_data = viz.ChartData(data, config)
    fig = viz.plot_xy_chart(chart_data, config)
    fig["layout"]["yaxis"]["autorange"] = "reversed"
    fig.update_layout(height=800)
    fig.show()
```

1.4 Data Conversion

This section shows how to convert raw data contained in a .hex file into engineering units for the instruments that follow:
- SBE37SM - SBE37SMP - SBE37SMPODO - SBE37IM - SBE37IMP - SBE37IMPODO - SBE19Plus

Supported output units include temperature, conductivity, pressure, density, potential density, and depth.

[12]:
```
hex_file = os.path.join(".", "example_data", "19plus_V2_CTD-processing_example.
 ↪hex")
raw_data = id.read_hex_file(
    filepath=hex_file,
    instrument_type=id.InstrumentType.SBE19Plus,
    enabled_sensors=[
        id.Sensors.Temperature,
        id.Sensors.Conductivity,
        id.Sensors.Pressure,
        id.Sensors.ExtVolt0,
        id.Sensors.ExtVolt1,
        id.Sensors.ExtVolt2,
        id.Sensors.ExtVolt3,
    ],
)
cal = cal.SN6130  # fake some cal coefficients for demonstration

temperature = conv.convert_temperature_array(
    temperature_counts=raw_data["temperature"].values,
    a0=cal.A0,
    a1=cal.A1,
    a2=cal.A2,
    a3=cal.A3,
    ITS90=False,
    celsius=True,
    use_MV_R=True,  # TODO: clearly define MV and R
)

pressure = conv.convert_pressure_array(
    raw_data["pressure"].values,
    raw_data["temperature compensation"],
    False,
    cal.PA0,
    cal.PA1,
    cal.PA2,
    cal.PTEMPA0,
    cal.PTEMPA1,
    cal.PTEMPA2,
    cal.PTCA0,
    cal.PTCA1,
    cal.PTCA2,
    cal.PTCB0,
    cal.PTCB1,
    cal.PTCB2,
)

conductivity = conv.convert_conductivity_array(
    raw_data["conductivity"].values,
    temperature,
    pressure,
    cal.G,
    cal.H,
    cal.I,
    cal.J,
    cal.CPCOR,
    cal.CTCOR,
    cal.WBOTC,
)

flag = np.zeros(len(temperature))

tcp_data = np.array([temperature, pressure, conductivity, flag]).transpose()
data = pd.DataFrame(data=tcp_data, columns=["temp", "press", "cond", "flag"])

plot(data=data, title="Data Conversion", x_names=["temp", "cond"], x_bounds={})
```



1.5 Low Pass Filter

This section shows how to use a low-pass filter on one or more columns of converted data. A low-pass filter removes high frequency data to make the data smoother. To make sure there is zero phase shift (no time shift), the filter is first applied forward through the data and then backward through the forward-filtered data. This removes any delays caused by the filter.

Pressure data is typically filtered with a time constant equal to four times the CTD scan rate. Conductivity and temperature are typically filtered for *some* CTDs. Typical time constants are:

Instrument	Temperature	Conductivity	Pressure
SBE 19plus or 19plus V2	0.5	0.5	1.0

1.5.1 Filter Formula

For a low-pass filter with time constant Γ:

$\Gamma = 1/2\pi f$
$T =$ sample interval (seconds)
$S_0 = 1/\Gamma$

Laplace transform of the transfer function of a low-pass filter (single pole) with a time constant of Γ seconds is:

$$H(s) = \frac{1}{1+(S/S_0)}$$

Using the bilinear transform:

$$S - f(z) \triangleq \frac{2(1-z^{-1})}{T(1+z^{-1})} = \frac{2(z-1)}{T(z+1)}$$

$$H(z) = \frac{1}{1+\frac{2(z-1)}{T(z+1)S_0}} = \frac{z^{-1}+1}{1+\frac{2}{TS_0}(1+\frac{1-2/TS_0}{1+2/TS_0}z^{-1})}$$

If:

$$A = \frac{1}{1+\frac{2}{TS_0}}$$

$$B = \frac{1-\frac{2}{TS_0}}{1+\frac{2}{TS_0}}$$

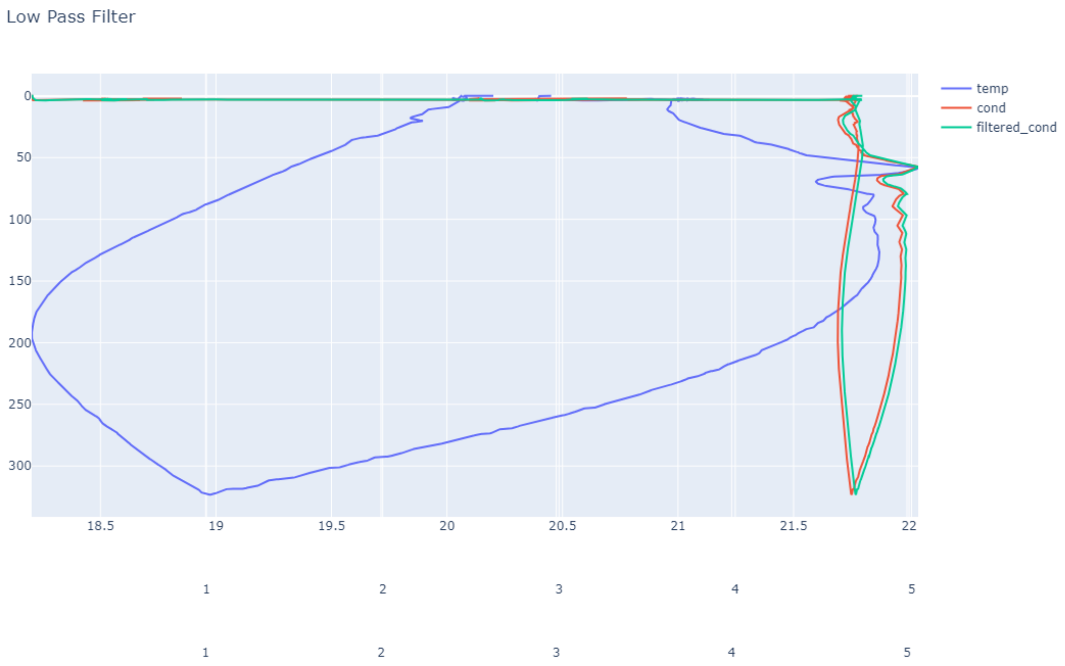
Then:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{A(z^{-1}+1)}{1+Bz^{-1}}$$

Where z^{-1} is the unit delay (one scan behind).

$Y(z)(1 + Bz - 1) = X(z)A(z - 1 + 1)$
$y[N] + By[N - 1] = Ax[N - 1] + Ax[N]$
$y[N] = A(x[N] + x[N - 1]) - By[N - 1]$

[13]:
```
data["filtered_temp"] = proc.low_pass_filter(
    x=data["temp"], time_constant=1, sample_interval=0.25
)

data["filtered_cond"] = proc.low_pass_filter(
    x=data["cond"], time_constant=1, sample_interval=0.25
)

plot(
    data=data,
    title="Low Pass Filter",
    x_names=["temp", "cond", "filtered_cond"],
    x_bounds={},
)
```



1.6 Align CTD

This section shows how the Align CTD function aligns parameter data in time, relative to pressure, so that the calculated salinity, dissolved oxygen concentration, and other parameters are made with measurements from the same parcel of water. Typically, Align CTD is used to align temperature, conductivity, and oxygen measurements relative to pressure.

There are three principal causes of misalignment of CTD measurements: - physical misalignment of the sensors in depth - inherent time delay (time constants) of the sensor responses - water transit time delay in the pumped plumbing line - the time it takes the parcel of water to go through the plumbing to each sensor (or, for free-flushing sensors, the related flushing delay, which depends on profiling speed)

When measurements are correctly aligned, salinity spikes (and density) errors are minimized, and oxygen data agrees with the correct pressure (e.g., temperature vs. oxygen plots agree between down and up profiles).

1.6.1 Conductivity and Temperature

Temperature and conductivity are often misaligned with respect to pressure. It can help to move the temperature and conductivity relative to pressure.

The best diagnostic of correct alignment is the removal of salinity spikes that align with very sharp temperature steps. To determine the best alignment, make a plot of 10 meters of temperature and

salinity data at a depth that contains a very sharp temperature step. For the downcast, when temperature and salinity decrease with increased pressure:
- A negative salinity spike at the conductivity step means that conductivity leads temperature (conductivity sensor sees step before temperature sensor does). Move conductivity relative to temperature a negative number of seconds.
- If the salinity spike is positive, move conductivity relative to temperature a positive number of seconds.

The best alignment of conductivity and temperature occurs when the salinity spikes are minimized. It may be necessary to try different intervals to find the best alignment.

Typical Temperature Alignment The SBE 19, 19plus, and 19plus V2 use a temperature sensor with a relatively slow response time.

Instrument	Advance of temperature relative to pressure
19plus or 19plus V2	+0.5 seconds

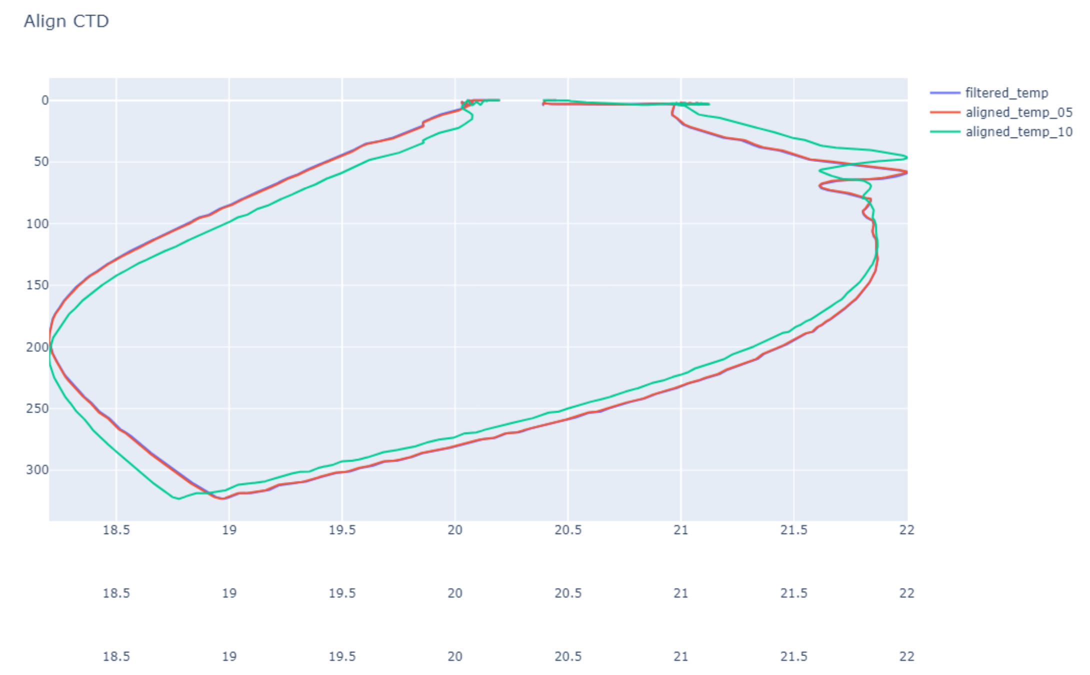
Typical Conductivity Alignment For an SBE 19plus or 19plus V2 with a standard 2000-rpm pump, do not move conductivity.
However, if temperature is moved relative to pressure and you do not want to change the relative timing of temperature and conductivity, you must add the same interval to conductivity.

1.6.2 Oxygen

Oxygen data is also systematically delayed with respect to pressure. The two primary causes are the long time constant of the oxygen sensor (for the SBE 43, ranging from 2 seconds at 25 ºC to approximately 5 seconds at 0 ºC) and an additional delay from the transit time of water in the pumped plumbing line. As with temperature and conductivity, you can move oxygen data relative to pressure to adjust for this delay.

Instrument	Advance of oxygen relative to pressure
19plus or 19plus V2	+3 to +7 seconds

```
[14]: data["aligned_temp_05"] = proc.align_ctd(
          x=data["filtered_temp"], offset=0.5, sample_interval=0.25
      )

      data["aligned_temp_10"] = proc.align_ctd(
          x=data["filtered_temp"], offset=10, sample_interval=0.25
      )

      plot(
          data=data,
          title="Align CTD",
          x_names=["filtered_temp", "aligned_temp_05", "aligned_temp_10"],
          x_bounds={},
)
```



1.7 Cell Thermal Mass

This section shows the use of the Cell Thermal Mass function which uses a recursive filter to remove conductivity cell thermal mass effects from the measured conductivity. Typical values for alpha and 1/beta are:

Instrument	alpha	1/beta
SBE 19plus or 19plus V2 with TC duct and 2000 rpm pump	0.04	8.0

1.7.1 Cell Thermal Mass Formulas

$$a = \frac{2 * alpha}{(sample\ interval * beta + 2)}$$

$$b = 1 - \frac{2a}{alpha}$$

$$\frac{c}{dT} = 0.1 + 6e\text{-}4 * (temperature - 20)$$

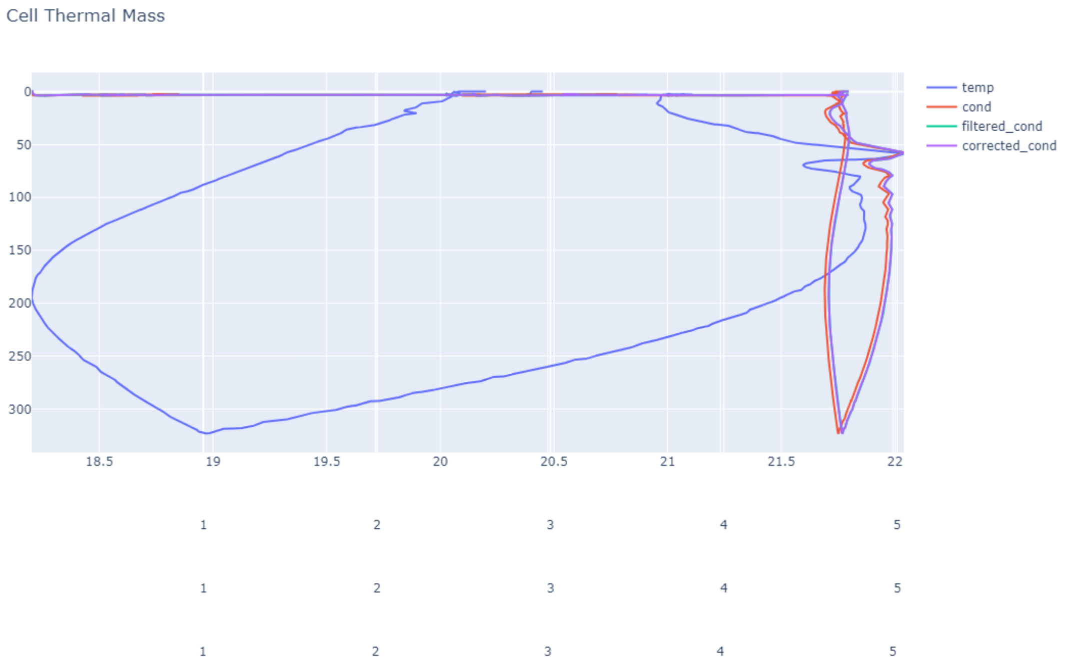
$$dT = temperature - previous\ temperature$$

$$ctm = -b * previous\ ctm + a * \frac{c}{dT} * dT$$

Where sample interval is in seconds, temperature is in °C, and ctm is in S/m.

To determine the values for alpha and beta, see:
Lueck, R.G., 1990: Thermal Inertia of Conductivity Cells: Theory., American Meteorological Society Oct 1990, 741-755.

```
[15]: data["corrected_cond"] = proc.cell_thermal_mass(
          temperature_C=data["filtered_temp"],
          conductivity_Sm=data["filtered_cond"],
          amplitude=0.03,
          time_constant=7,
          sample_interval=0.25,
      )

      plot(
          data=data,
          title="Cell Thermal Mass",
          x_names=["temp", "cond", "filtered_cond", "corrected_cond"],
          x_bounds={},
      )
```

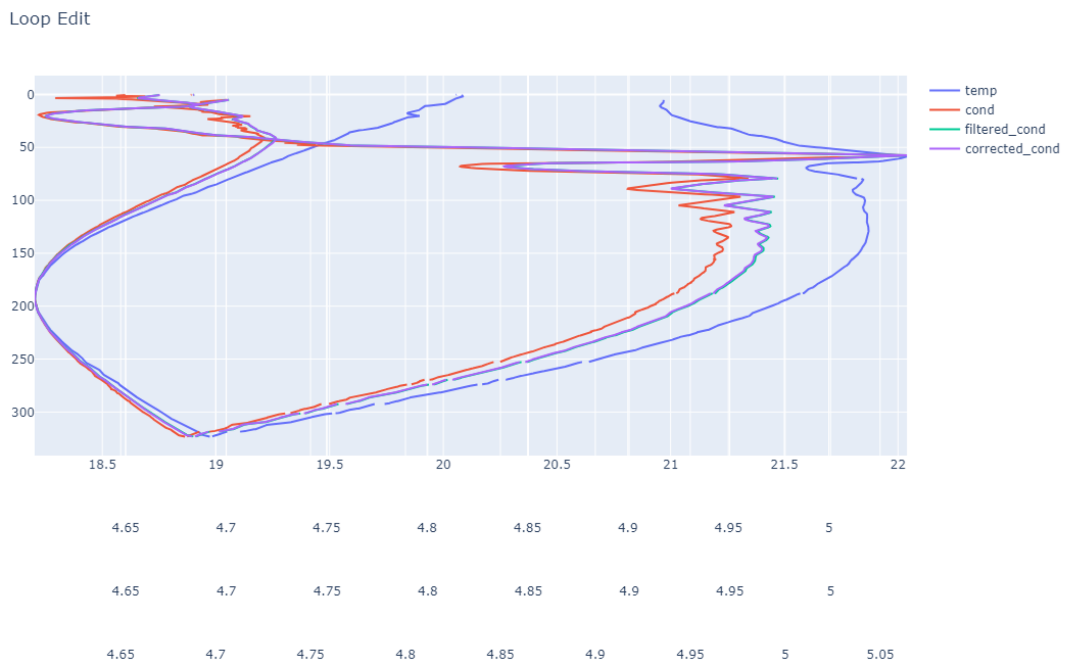


1.8 Loop Edit

This section shows how the Loop Edit function identifies scans as bad. The flag value associated with the scan is setto badflag in input .cnv files that have pressure slowdowns or reversals (typically caused by ship heave). Optionally, Loop Edit can also identify scans related to an initial surface soak with badflag. The badflag value defaults to -9.99e-29.

Loop Edit operates on three successive scans to determine velocity. This is such a fine scale that noise in the pressure channel from counting jitter or other unknown sources can cause Loop Edit to mark scans with badflag in error. Therefore, you must do the Filter on the pressure data to reduce noise before you do the Loop Edit. See Filter for pressure filter recommendations for each instrument.

[16]:
```
proc.loop_edit_pressure(
    pressure=data["press"].values,
    latitude=0,
    flag=data["flag"].values,
    sample_interval=0.25,
    min_velocity_type=proc.MinVelocityType.FIXED,
    min_velocity=0.1,
    window_size=3,
    mean_speed_percent=20,
    remove_surface_soak=True,
    min_soak_depth=5,
    max_soak_depth=20,
    use_deck_pressure_offset=False,
    exclude_flags=True,
    flag_value=-9.99e-29,
)

plot(
    data=data,
    title="Loop Edit",
    x_names=["temp", "cond", "filtered_cond", "corrected_cond"],
    x_bounds={},
)
```



1.9 Derive TEOS-10

This section shows how the Derive TEOS-10 function uses temperature, conductivity, practical salinity, pressure, latitude, and/or longitude to compute the following thermodynamic parameters using TEOS-10 equations:

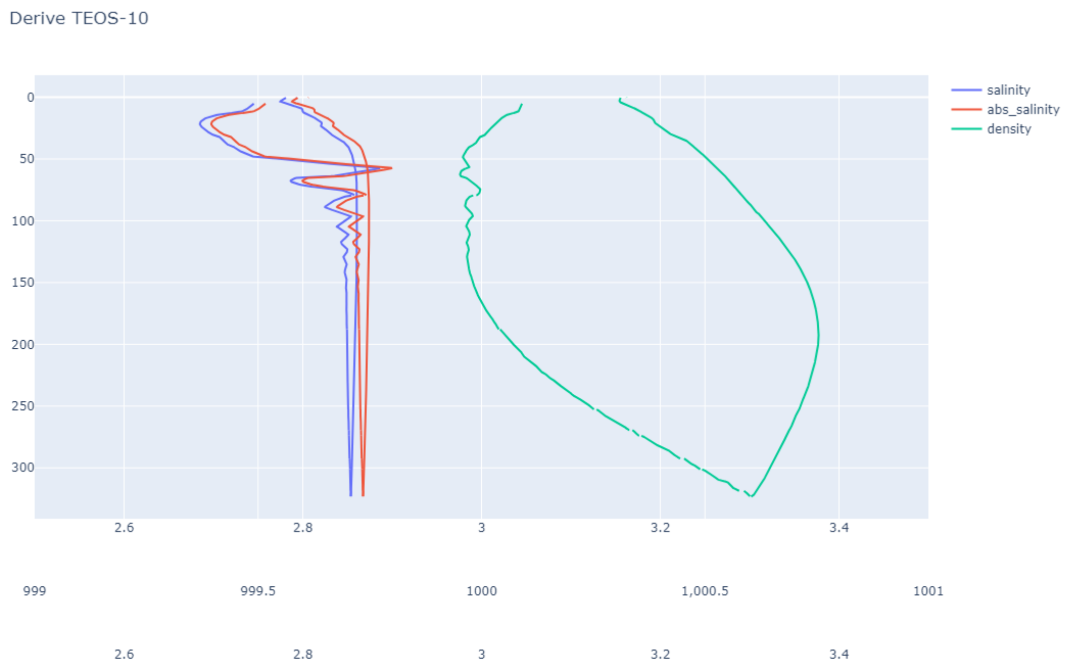
The table below references python functions from TEOS-10 that are the same functions in SBE Data Processing. > Note: Results may differ slightly from SBE Data Processing due to GSW version differences.

Variable Name	GSW-Python function
Absolute Salinity	gsw.SA_from_SP
Absolute Salinity Anomaly	gsw.deltaSA_from_SP
adiabatic lapse rate	gsw.adiabatic_lapse_rate_from_CT
Conservative Temperature	gsw.CT_from_t
Conservative Temperature, freezing	gsw.CT_freezing
density, TEOS-10	gsw.rho1
dynamic enthalpy	gsw.dynamic_enthalpy
enthalpy	gsw.enthalpy
entropy	gsw.entropy_from_t
gravity	gsw.grav
internal energy	gsw.internal_energy

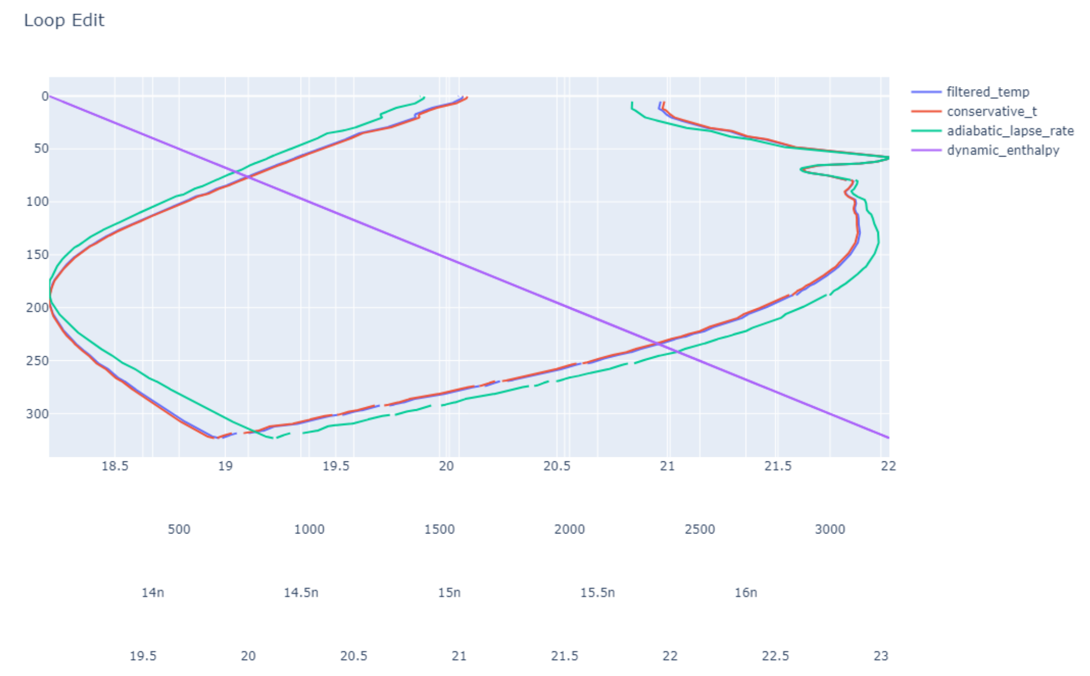
Variable Name	GSW-Python function
isentropic compressibility	gsw.kappa
latent heat of evaporation	gsw.latentheat_evap_CT
latent heat of melting	gsw.latentheat_melting
potential temperature	gsw.pt0_from_t
Preformed Salinity	gsw.Sstar_from_SA
Reference Salinity	gsw.SR_from_SP
saline contraction coefficient	gsw.beta
sound speed	gsw.sound_speed
specific volume	gsw.specvol
specific volume anomaly	gsw.specvol_anom_standard
temperature freezing	gsw.t_freezing
thermal expansion coefficient	gsw.alpha

1 use gsw.rho with reference pressure for the sigmas

[17]:
```
data['salinity'] = gsw.SP_from_C(
    C=data['corrected_cond'].values,
    t=data['filtered_temp'].values,
    p=data['press'].values
)

data["abs_salinity"] = gsw.SA_from_SP(
    SP=data["salinity"].values, p=data["press"].values, lon=0, lat=0
)

data['conservative_t'] = gsw.CT_from_t(
    SA=data['abs_salinity'].values,
    t=data['filtered_temp'].values,
    p=data['press'].values
)

data['conservative_t'] = gsw.CT_from_t(
    SA=data['abs_salinity'].values,
    t=data['filtered_temp'].values,
    p=data['press'].values
)

data['density'] = gsw.rho(
    SA=data['abs_salinity'].values,
    CT=data['conservative_t'].values,
    p=[0]
)

plot(
    data=data,
    title='Derive TEOS-10',
    x_names=['salinity', 'abs_salinity', 'density'],
    x_bounds={0: [2.5, 3.5], 1: [2.5, 3.5], 2: [999, 1001]}
)
```



[18]:
```
data['adiabatic_lapse_rate'] = gsw.adiabatic_lapse_rate_from_CT(
    SA=data['abs_salinity'].values,
    CT=data['conservative_t'].values,
    p=data['press'].values
)

data["dynamic_enthalpy"] = gsw.dynamic_enthalpy(
    SA=data["abs_salinity"].values,
    CT=data["conservative_t"].values,
    p=data["press"].values,
)

plot(
    data=data,
    title="Loop Edit",
    x_names=[
        "filtered_temp",
        "conservative_t",
        "adiabatic_lapse_rate",
        "dynamic_enthalpy",
    ],
    x_bounds={},
)
```



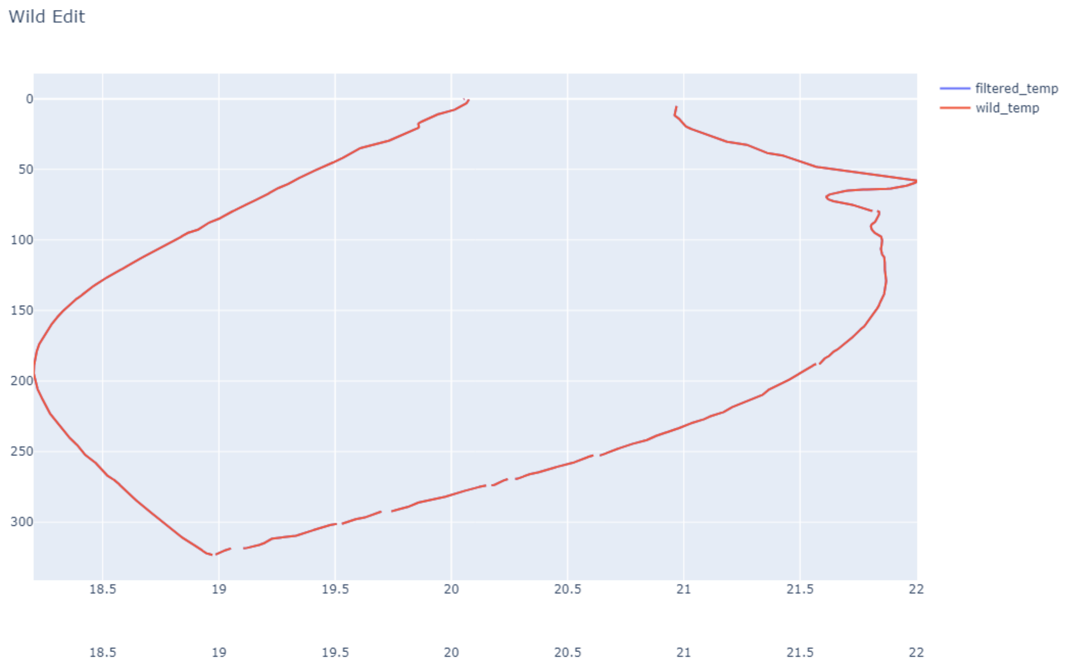
1.10 Wild Edit

This section shows how the Wild Edit function identifies wild points in the data. The data value is replaced with a badflag value, by default -9.99e-29. Wild Edit's algorithm requires two passes through the data. The first pass gives an accurate estimate of the data's true standard deviation, while the second pass replaces the appropriate data with badflag.

Wild Edit operates as follows: 1. Compute the mean and standard deviation of data in block (specified by Scans per Block) for each selected variable. Temporarily flag values that differ from mean by more than standard deviations specified for pass 1. 2. Recompute mean and standard deviation, and do not include temporarily flagged values. Identify the values that differ from the mean by more than standard deviations specified for pass 2 and replace the data value with badflag. 3. Repeat Steps 1 and 2 for next block of scans. If the last block of data in the input file has less than specified number of scans, use data from previous block to fill in the block.

If the data file is particularly corrupted, you may need to do Wild Edit more than once, with different block sizes and number of standard deviations. If the input file has some variables with large values and some with relatively smaller values, it may be necessary to run Wild Edit more than once, and change the value for Keep data within this distance of mean so that it is meaningful for each variable. Increase Scans per block from 100 to approximately 500 to get better results.

[19]:
```
data["wild_temp"] = proc.wild_edit(
    data=data["filtered_temp"],
    flags=data["flag"],
    std_pass_1=2,
    std_pass_2=20,  # TODO: this breaks when set to 3 or lower
    scans_per_block=100,
    distance_to_mean=0,
    exclude_bad_flags=True,
)

plot(data=data, title="Wild Edit", x_names=["filtered_temp", "wild_temp"],␣
 ↪x_bounds={})
```



1.11 Window Filter

The Window Filter function gives four types of window filters and a median filter that can be used to make the data smoother:
- Window filters calculate a weighted average of data values about a center point and replace the data value at the center point with this average.
- The median filter calculates a median for data values around a center point and replaces the data value at the center point with the median

1.11.1 Descriptions and Formulas

Shape and length define filter windows:
- Window Filter gives four window shapes: boxcar, cosine, triangle, and Gaussian.
- The minimum window length is 1 scan, and the maximum is 511 scans. Window length must be an odd number, so that the window has a center point. If a window length is specified as an even number, Window Filter automatically adds 1 to make the length odd.

The window filter calculates a weighted average of data values around a center point, using the transfer function below:

$$y(n) = \sum_{k=-L/2}^{L/2} w(k)x(n-k)$$

The window filter process is similar for all filter types:
1. Filter weights are calculated (see the equations below).
2. Filter weights are normalized to sum to 1. - When the data includes scans that have been identified as bad, and the `exclude_flags` variable is set to true, the weights are renormalized, and do not include the filter elements that would operate on the bad data point.

In the equations below:

$L = window\ length\ in\ scans\ (must\ be\ odd)$
$n = window\ index,\ -\frac{L-1}{2}..\frac{L-1}{2}$
$w(n) = set\ of\ window\ weights$

Boxcar Filter $w(n) = \frac{1}{L}$

Cosine Filter $w(n) = cos(\frac{n\pi}{L+1})$

Triangle Filter $w(n) = 1 - \frac{|2n|}{L+1}$

Gaussian Filter $phase = \frac{offset}{sample\ interval}$

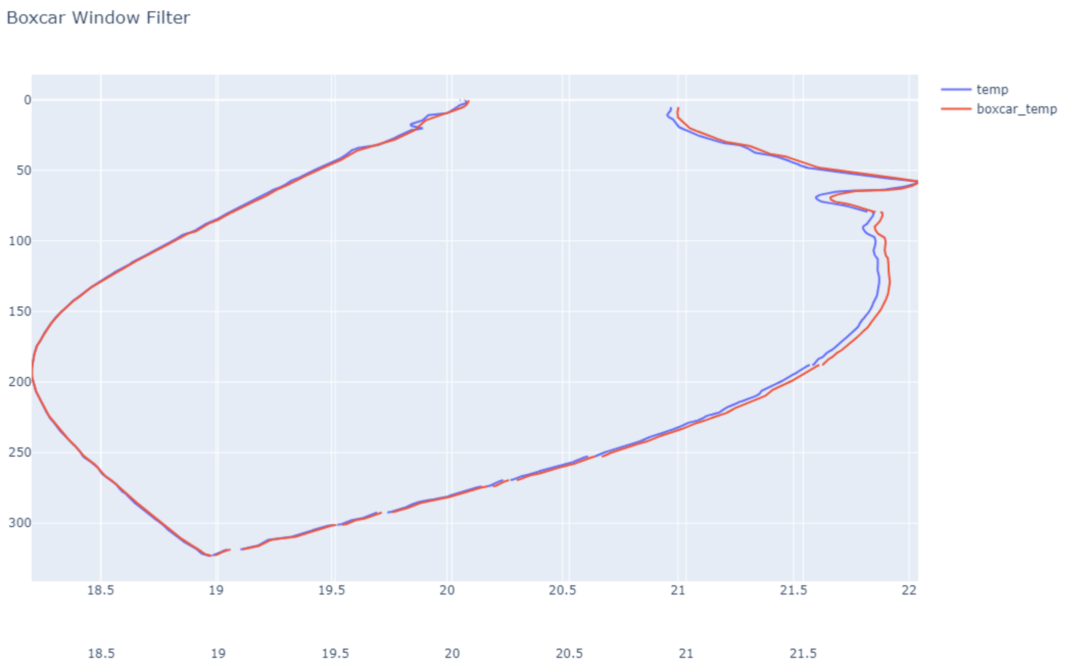
$scale = log(2)(2\frac{sample\ rate}{half\ width\ (scans)})^2$

$w(n) = e^{(n-phase)^2 * scale}$

The Gaussian window has parameters of halfwidth (in scans) and offset (in time), in addition to window length (in scans). These extra parameters filter and move data in time in one operation. Halfwidth determines the width of the Gaussian curve. A window length of 9 and halfwidth of 4 gives a set of filter weights that fills the window. A window length of 17 and halfwidth of 4 gives a set of filter weights that fills only half the window. If the filter weights do not fill the window, the offset parameter may be used to move the weights within the window so that the edge of the Gaussian curve is not cut.

Median Filter The median filter does not make data smoother in the same sense as the window filters described above. The median filter is most useful in spike removal. A median value is determined for a specified window, and the data value at the window's center point is replaced by the median value.

[20]:
```python
data["boxcar_temp"] = proc.window_filter(
    data_in=data["temp"],
    flags=data["flag"],
    window_type=proc.WindowFilterType.BOXCAR,
    window_width=21,
    sample_interval=0.25,
)

plot(
    data=data,
    title="Boxcar Window Filter",
    x_names=["temp", "boxcar_temp"],
    x_bounds={},
)
```


Boxcar Window Filter
temp
boxcar_temp