CTD_dfo-eva035-20230915-NEW

November 21, 2025

1 CTD corrections applied to delayed-mode data

This notebook performs analysis and correction of GPCTD data from the following C-PROOF glider deployment:

[1]:
```
glider_name = 'dfo-eva035'
deploy_name = f'{glider_name}-20230915'
deploy_prefix = f'./glider/{glider_name}/{deploy_name}/'
filepath = f'deployments/{glider_name}/{deploy_name}/' # having this is␣
 ↪important later for functions that auto-load data
openfile = f'{filepath}/L0-timeseries/{deploy_name}_delayed.nc'
opengridfile = f'{filepath}/L0-gridfiles/{deploy_name}_grid_delayed.nc'
deployfile = f'{filepath}/deployment.yml'

description = 'Calvert'
initials = 'LT'

# CTD specs:
sensor = 'CTD_0256'

# For conductivity filter:
accuracy = 0.0003 #accuracy of the sensor is 0.0003 S/m, used as a cutoff on␣
 ↪the exclusion criterion

from datetime import date
processing_date = date.today().strftime('%Y%m%d')
processing_protocol = 'C-PROOF_SBE_CTDProcessingReport_v0.2.pdf'
processing_report = f'CTD_{deploy_name}'


# Import module for loading .md files
from IPython.display import Markdown, display
import os

os.chdir(f'/Users/Lauryn/Documents/processing/')
```

[2]:
```
# Summarize info for report:
print(f'** {description}:  glider {glider_name}**')
print(f'************')
print(f'* Deployment: {deploy_name}')
print(f'* Sensor: {sensor}')
print(f'')

# print(f'* Protocols are detailed in: {processing_protocol}')
print(f'* Processing steps will be saved in: CTD_{deploy_name}.html')
# print(f'* Files will be located in: {deploy_prefix}')
print(f'* Processed by {initials}, Ocean Sciences Division, Fisheries and␣
 ↪Oceans Canada')
print(f'* Processing date: {processing_date}')
```

```
** Calvert:  glider dfo-eva035**
************
* Deployment: dfo-eva035-20230915
* Sensor: CTD_0256

* Processing steps will be saved in: CTD_dfo-eva035-20230915.html
* Processed by LT, Ocean Sciences Division, Fisheries and Oceans Canada
* Processing date: 20251121
```

[3]:
```
display(Markdown("./docs/CTD_1_Preamble.md"))
```

2 1.0 Preamble

This document describes conductivity, temperature, and pressure data processing steps applied to delayed mode data collected using Sea-Bird Scientific Glider Payload Conductivity Temperature Depth (GPCTD) sensors mounted on C-PROOF Slocum and SeaExplorer autonomous ocean gliders. This sensor has a nominal sampling rate of 1 Hz and was designed specifically for Slocum gliders. This document covers the application of the sensor alignment correction and the thermal lag correction, as well as removal of questionable conductivity values and salinity profiles.

2.1 1.1 Set up the processing

The processing steps below are applied to delayed mode data stored in a single netcdf timeseries file created using the Pyglider data processing package (https://github.com/c-proof/pyglider).

The metadata and sensor calibration sheets are available via the deployment page on the C-PROOF website at: **https://cproof.uvic.ca/gliderdata/deployments/dfo-bb046/dfo-bb046-20220707/**

[4]:
```
import warnings
warnings.filterwarnings('ignore')

import xarray as xr
import numpy as np
```

```
import pathlib
import pyglidersensor as pgs
from pyglider.ncprocess import make_gridfiles

from datetime import datetime, date
%matplotlib ipympl


import scipy.stats as stats

import seawater
import gsw

%matplotlib notebook
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
import cmocean
import cartopy.crs as ccrs
import cartopy.feature as cfeature

import pandas as pd

%load_ext autoreload
%autoreload 2

from scipy import signal
import seawater as sw
```

[5]:
```
%reload_ext autoreload
%matplotlib ipympl
```

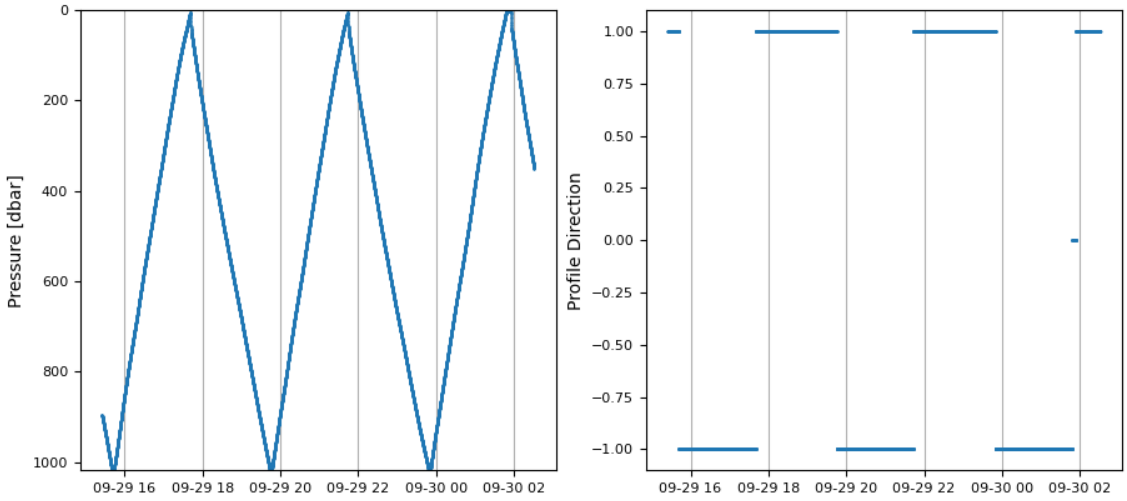
2.2 1.2 Profile Check

Check that upcasts and downcasts are being properly identified. **Negative values should be associated with upcasts**.

[6]:
```
print(f'Loading: {openfile}')

caption = ('Identifying upcasts and downcasts. The left panel shows '
           'pressure vs. time and the right panel shows profile direction vs. '
           'time for a small subset of the time series:')


fname = openfile
```

```
with xr.open_dataset(fname) as ds0:
    # SAVE SOME PARAMS FOR PLOTTING DOWN BELOW!
    N = len(ds0.time)
    MAX_DEPTH = np.nanmax(ds0.depth)
    NUM_PROFILES = np.nanmax(ds0.profile_index)

    print('************')
    print(f'* There are {N} data points in total, with {NUM_PROFILES} profiles')
    print(f'* Time period: {pd.to_datetime(np.nanmin(ds0.time)).
 ↪strftime("%Y-%m-%d")} to {pd.to_datetime(np.nanmax(ds0.time)).
 ↪strftime("%Y-%m-%d")}')
    print(f'* Depth range: {round(np.nanmin(ds0.depth))} - {round(MAX_DEPTH)}␣
 ↪metres')
    print('************')
    if N > 50000:
        todo = slice(int(N/2)-5000, int(N/2)+5000)
    else:
        todo = slice(int(N/3), int(2*N/3))

    fig, axs = plt.subplots(nrows=1, ncols=2,
                            constrained_layout=True,
                            figsize=(9, 4))

    ds = ds0.isel(time=todo)
    axs[0].plot(ds.time, ds.pressure, '.', markersize=1)
    axs[0].set_ylim([MAX_DEPTH, 0])
    axs[0].set_ylabel('Pressure [dbar]')
    axs[0].tick_params(axis='both', labelsize=8)
    axs[0].grid(axis='x')

    axs[1].plot(ds.time, ds.profile_direction, '.', markersize=1)
    axs[1].set_ylabel('Profile Direction')
    axs[1].tick_params(axis='both', labelsize=8)
    axs[1].grid(axis='x')
    print(caption)
```

```
Loading: deployments/dfo-eva035/dfo-eva035-20230915//L0-timeseries/dfo-
eva035-20230915_delayed.nc
************
* There are 607042 data points in total, with 1007.0 profiles
* Time period: 2023-09-15 to 2023-10-13
* Depth range: 0 - 1018 metres
************
Identifying upcasts and downcasts. The left panel shows pressure vs. time and
the right panel shows profile direction vs. time for a small subset of the time
series:
```

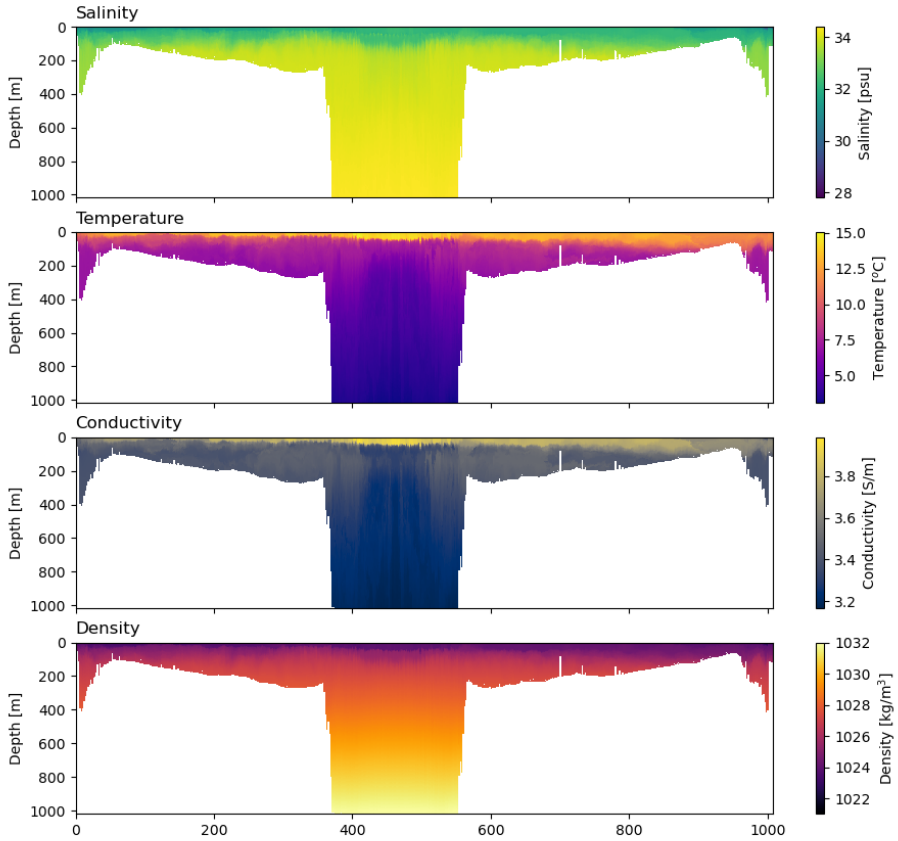


2.3 1.3 Delayed-mode data prior to corrections

Checking fields (temperature, salinity, conductivity and density) in the delayed-mode data, before any CTD corrections:

[7]:
```
tds = opengridfile
ds = xr.open_dataset(tds)
list(ds.keys())

fig, axs = plt.subplots(4, 1, figsize=(11, 10), sharey=True, sharex=True)

xlims = [0, NUM_PROFILES]
ylims=[MAX_DEPTH,0]
# ylims=[50,0]

pc = axs[0 ].pcolormesh(ds.profile, ds.depth, ds['salinity'],rasterized=True)
axs[0].set_ylim(ylims)
axs[1].set_xlim(xlims)
fig.colorbar(pc, ax=axs[0], label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

pc = axs[1].pcolormesh(ds.profile, ds.depth,␣
 ↪ds['temperature'],rasterized=True,cmap='plasma')

fig.colorbar(pc, ax=axs[1], label = 'Temperature [$^o$C]')
axs[1].set_title('Temperature',loc='left')
pc = axs[2].pcolormesh(ds.profile, ds.depth,␣
 ↪ds['conductivity'],rasterized=True,cmap='cividis')
fig.colorbar(pc, ax=axs[2], label = 'Conductivity [S/m]')
axs[2].set_title('Conductivity',loc='left')
```

```
# pc = axs[3].pcolormesh(ds.profile, ds.depth,␣
 ↪ds['oxygen_concentration'],rasterized=True,cmap='inferno')
# fig.colorbar(pc, ax=axs[3])
# axs[3].set_title('Oxygen Concentration',loc='left')

pc = axs[3].pcolormesh(ds.profile, ds.depth,␣
 ↪ds['density'],rasterized=True,cmap='inferno')
fig.colorbar(pc, ax=axs[3], label = 'Density [kg/m$^3$]')
axs[3].set_title('Density',loc='left')

axs[0].set_ylabel('Depth [m]');
axs[1].set_ylabel('Depth [m]');
axs[2].set_ylabel('Depth [m]');
axs[3].set_ylabel('Depth [m]');
```



[8]:
```
display(Markdown("./docs/CTD_2_Steps.md"))
```

3 2.0 Corrections applied to delayed mode data for this mission

Processing steps:

1. Identification of anomalous conductivity values
2. Identification of questionable salinity profiles
3. Sensor alignment correction
4. Thermal lag correction

3.1 Apply QC flags to data following Argo notation

1 : Good data
4 : Bad data that are potentially correctable
4 : Bad data
8 : Estimated value (interpolated, extrapolated or other estimation)

3.2 2.1.1 Identification and removal of anomalous conductivity values

We identify and remove any conductivity values that are obviously unphysical, which is typically caused by air bubbles in the conductivity cell. We use a simple criterion applied to the raw conductivity data. The criterion temporarily flags any data points that are more than **5 standard deviations** away from the overall time series mean for a given depth bin and profile bin, then recomputes the mean and standard deviation, excluding the temporarily flagged values. Conductivity values that still differ from the mean by more than **3 standard deviations** are flagged as 'bad' (QC 4). If the difference between the 'bad' values and the mean is less than the accuracy of the sensor, which is 0.0003 S/m for the GPCTD, then those points are not excluded.

This criterion is applied to data binned first by profile index, in increments of **50 profiles**, then binned by depth, in increments of **5 m**. The use of profile index bins rather than time or temperature bins is designed to allow for the removal of unphysical values.

Adjustments to this correction are based on examining the data and making a judgment call about which conductivity values are undeniably 'bad'. In this case, we want to exclude the **extremely low values occurring at the surface** consistent with air bubbles in the cell. Some unphysical values are missed by this correction, and may be caught during the removal of unphysical salinity profiles in further stepsbelow.

Note that for this mission:

[16]:
```
srate = stats.mode((np.diff(ds0.time)).astype('timedelta64[s]')).mode
fs = 1/srate.astype(float) #the sampling frequency = 1/(delta t)
print('************')
print(f'The mode of the sampling rate for the GPCTD is one sample every {srate}.
  ↪')
print('************')
```

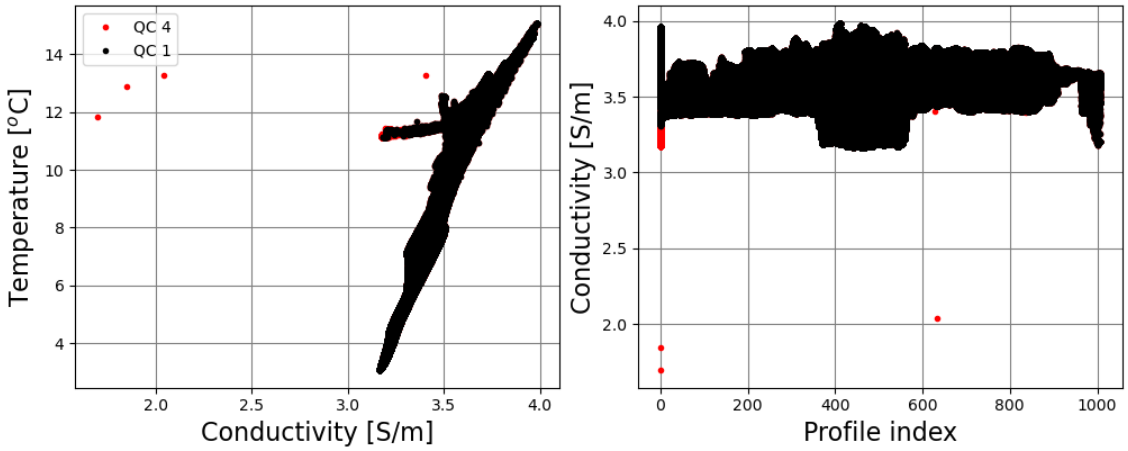
```
************
The mode of the sampling rate for the GPCTD is one sample every 4 seconds.
```

****

[17]:
```
# Identify the questionable conductivity values
flag_stdev = 5 #number of standard deviations to temporarily flag bad salinity␣
 ↪values
clean_stdev = 3 #number of standard deviations to flag bad conductivity values,␣
 ↪after removing the temporary bad values from the calc
dT = 50 #size of the profile bins
dz = 5 #size of the depth bins

ts0 = ds0.copy()
ts0.conductivity[ts0.conductivity<0.1] = np.nan
ts = pgs.get_conductivity_clean(ts0, dT, dz, flag_stdev, clean_stdev, accuracy)


##### make new variable for conductivity QC

ts = ts.assign(conductivity_QC = np.nan*ts.conductivity)
ts['conductivity_QC'] = xr.where(np.isfinite(ts.conductivityClean), 1, 4)

#####

# Figures to look at the comparison
fig, ax = plt.subplots(1,2,figsize=(10,4), constrained_layout=True)

ax[0].plot(ts.conductivity, ts.temperature, color='r', marker='.',␣
 ↪linestyle='none', label='QC 4')
ax[0].plot(ts.conductivityClean, ts.temperature, color='k', marker='.',␣
 ↪linestyle='none', label='QC 1')
ax[0].set_ylabel('Temperature [$^o$C]', fontsize=16)
ax[0].set_xlabel('Conductivity [S/m]', fontsize=16)
ax[0].grid(axis='both', color='0.5')

ax[1].plot(ts.profile_index, ts.conductivity, color='r', marker='.',␣
 ↪linestyle='none')
ax[1].plot(ts.profile_index, ts.conductivityClean, color='k', marker='.',␣
 ↪linestyle='none')
ax[1].set_xlabel('Profile index', fontsize=16)
ax[1].set_ylabel('Conductivity [S/m]', fontsize=16)
ax[1].grid(axis='both', color='0.5')

ax[0].legend()

print('Fig 2: Temperature vs. conductivity (left), depth vs. conductivity␣
 ↪(middle), '
      'and conductivity vs. profile index (right), '
```

```
    'with the red dots showing the unphysical values flagged as bad and are␣
↪flagged as QC 4:')
```

Fig 2: Temperature vs. conductivity (left), depth vs. conductivity (middle), and conductivity vs. profile index (right), with the red dots showing the unphysical values flagged as bad and are flagged as QC 4:



Adjustments to this correction are based on examining the data and making a judgment call about which conductivity values are undeniably 'bad'. In this case, we want to flag the **extremely low values occurring at the surface** (Fig. 2) consistent with air bubbles in the cell. Some unphysical values are missed by this correction, and may be caught during the removal of unphysical salinity profiles below.

[18]:
```
###### grid to make finding bad profiles easier
ts.to_netcdf(f'{filepath}/{deploy_name}_QC.nc')
# Save a gridded version as well
outfile = make_gridfiles(f'{filepath}/{deploy_name}_QC.nc', f'{filepath}',␣
↪deployfile, fnamesuffix='QC')
```

3.2.1 2.2 Identifying questionable salinity profiles

Here, potentially suspicious salinity profiles are identified in order to prevent them from being used in the thermal lag correction. While these questionable salinity profiles are not included in the following steps, these profiles **are not removed** from the final corrected salinity product.

We identify any salinity profiles that are obviously unphysical, which is typically caused by something (usually biology) getting caught in the conductivity cell, and set all values within those profiles to NaN. We use a simple criterion applied to the salinity data, binned by temperature, with bin sizes based on the time series mean temperature profile. The criterion temporarily flags any data points that are more than **4 standard deviations** away from the overall mean for the salinity time series within a given temperature bin, then recomputes the mean and standard deviation, excluding the temporarily flagged values. Salinity values that still differ from the mean by more than **4 standard deviations** are flagged as 'bad'. Finally, any profile where more than **10%**

of the salinity values have been flagged as 'bad' using this criterion is removed. The number of standard deviations used and the percent of flagged required to flag a profile as 'bad' can be adjusted.

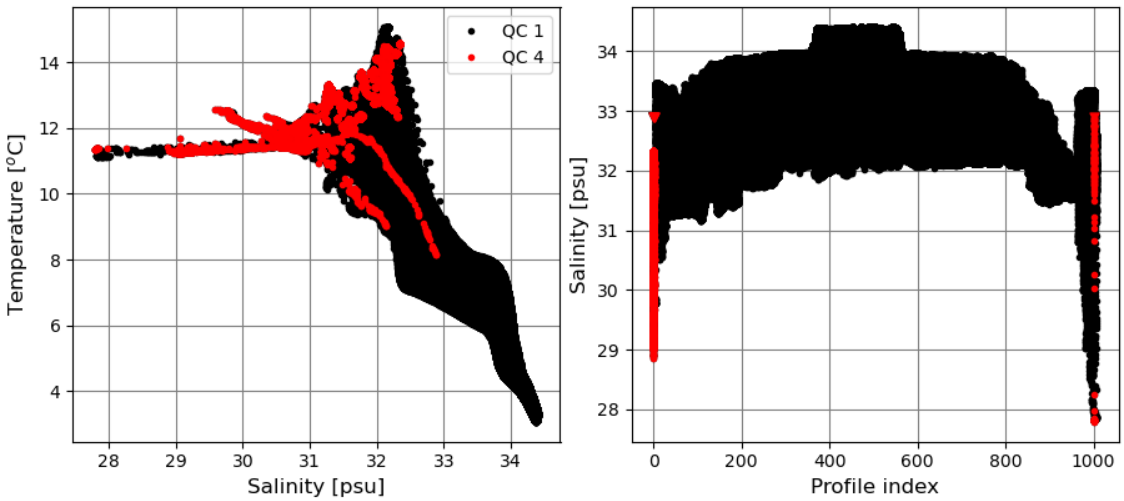
[19]:
```
fname = f'{filepath}/{deploy_name}_QC.nc'
gridfname = f'{filepath}/{deploy_name}_gridQC.nc'

ds=xr.open_dataset(gridfname)
ts = xr.open_dataset(fname) ##timeseries


####### find mean temperature profile of the timeseries
Tmean = ds['temperature'].mean(dim='time')
Tmean = Tmean.sortby(Tmean, ascending=True).where(np.isfinite(Tmean), drop=True)

# Identify the questionable salinity values
clean_profs = 0 #number of profiles to exclude from the start and end of the␣
 ↪time series
flag_stdev = 4 #number of standard deviations to temporarily flag bad salinity␣
 ↪values
clean_stdev = 4 #number of standard deviations to flag bad salinity values,␣
 ↪after removing the temporary bad values
clean_cutoff = 0.1 #fraction of bad salinity values required to label a profile␣
 ↪as bad
dtbin = 10 #number of temperature bins

sal = pgs.get_salinity_grid(ts, Tmean, clean_profs, flag_stdev, clean_stdev,␣
 ↪clean_cutoff, dtbin)

sal.to_netcdf(f'{filepath}/SalinityGrid.nc')

bad_profiles = sal.profiles.where(sal.bad >= clean_cutoff, drop=True)

print('Number of flagged profiles: '+ str(len(bad_profiles)))
print('Profiles flagged as bad due to questionable salinity values:',␣
 ↪bad_profiles.values)

#####
ts = ts.assign(salinity_QC = np.nan*ts.salinity)
ts['salinity_QC'] = xr.where(ts.profile_index.isin(bad_profiles), 4, 1)
```

```
Number of flagged profiles: 4
Profiles flagged as bad due to questionable salinity values: [0.000e+00
1.000e+00 2.000e+00 1.002e+03]
```

[20]:
```
fig, ax = plt.subplots(1,2,figsize=(9,4),
                           constrained_layout=True)
```

```
sal4 = ts.where(ts.salinity_QC == 4, drop=True)

ax[0].plot(ts.salinity, ts.temperature, color='k', marker='.',␣
 ↪linestyle='none', label='QC 1')
ax[0].plot(sal4.salinity, sal4.temperature, color='r', marker='.',␣
 ↪linestyle='none', label='QC 4')

ax[0].set_ylabel('Temperature [$^o$C]', fontsize=12)
ax[0].set_xlabel('Salinity [psu]', fontsize=12)
ax[0].grid(axis='both', color='0.5')
ax[0].legend()


#####
ax[1].plot(ts.profile_index,ts.salinity,marker='.', linestyle='none',  c='k')
ax[1].plot(sal4.profile_index,sal4.salinity,marker='.', linestyle='none', c='r')

ax[1].set_ylabel('Salinity [psu]', fontsize=12)
ax[1].set_xlabel('Profile index', fontsize=12)
ax[1].grid(axis='both', color='0.5')
x = bad_profiles
y = np.nanmax(sal4.salinity.values) + np.zeros_like(bad_profiles)
ax[1].scatter(x,y,30,marker='v',color='k',zorder=1)
ax[1].scatter(x,y,25,marker='v',color='r',zorder=2)

print ('Salinity plotted as a function of temperature '
'(left) and vs. profile index (right), \n with the salinity profiles '
'flagged as QC 4 shown in red and indicated by '
'the red arrows at the top of the panel on the right:')
```

```
Salinity plotted as a function of temperature (left) and vs. profile index
(right),
 with the salinity profiles flagged as QC 4 shown in red and indicated by the
red arrows at the top of the panel on the right:
```



[21]:
```
###### grid to make finding bad profiles easier
ts.to_netcdf(f'{filepath}/{deploy_name}_QC2.nc')
# Save a gridded version as well
outfile = make_gridfiles(f'{filepath}/{deploy_name}_QC2.nc', f'{filepath}',␣
↪deployfile, fnamesuffix='QC2')
```

3.2.2 2.1.2 Manually flag profiles with spikes from biofouling

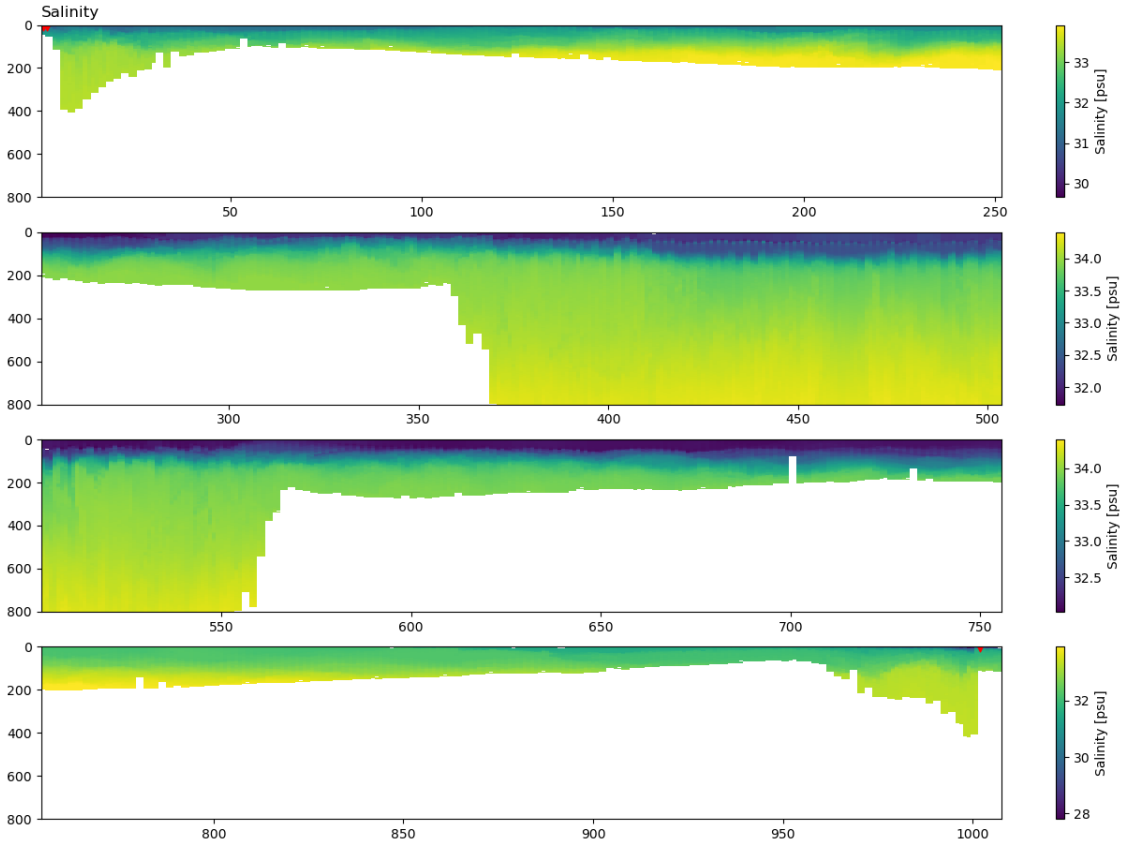
Manually remove spikes in the data. Sometimes **biofouling** occurs causing unphysical values.

[22]:
```
fname = f'{filepath}/{deploy_name}_QC2.nc'
gridfname = f'{filepath}/{deploy_name}_gridQC2.nc'

ds=xr.open_dataset(gridfname)
ts = xr.open_dataset(fname) ##timeseries


##################
sal4 = ds.where(ds.salinity_QC == 4, drop=True)


# Now adding zoomed plot
xlim_1 = [0, int(NUM_PROFILES/4)]
xlim_2 = [int(NUM_PROFILES/4), int(NUM_PROFILES/4*2)]
xlim_3 = [int(NUM_PROFILES/4*2), int(NUM_PROFILES/4*3)]
xlim_4 = [int(NUM_PROFILES/4*3), NUM_PROFILES]

Y_LIMS = [800, 0]

fig, axs = plt.subplots(4, 1, #height_ratios=[1, 4],
```

```
                        figsize = [12,9],
                        layout='constrained', sharex=False)

profile_lims = xlim_1
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[0]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
axs[0].set_ylim(Y_LIMS)

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
axs[0].scatter(x,y,60,marker='v',color='r')


########
profile_lims = xlim_2
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[1]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
ax.set_ylim(Y_LIMS)

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')


x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')


######
profile_lims = xlim_3
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[2]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
```

```
ax.set_ylim(Y_LIMS)

x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

#########
profile_lims = xlim_4
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[3]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
ax.set_ylim(Y_LIMS)
x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

print('Zooming in along the glider deployment to visualize salinity spikes. Red␣
 ↪arrows identify profiles already flagged as QC4.')
```

```
Zooming in along the glider deployment to visualize salinity spikes. Red arrows
identify profiles already flagged as QC4.
```

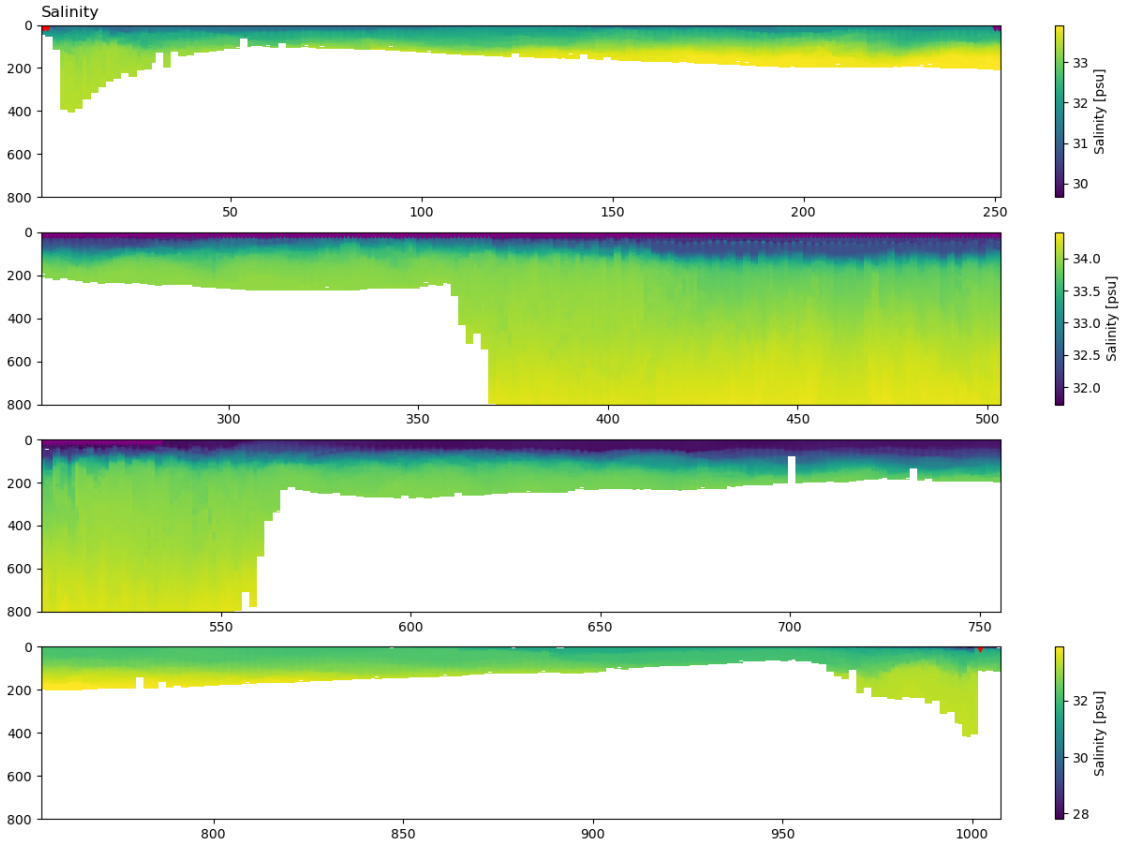


[24]:
```
###### Manually identify questionable profiles
bad_profiles2 = xr.DataArray(np.append(np.arange(250,535),0))


# Now adding zoomed plot
xlim_1 = [0, int(NUM_PROFILES/4)]
xlim_2 = [int(NUM_PROFILES/4), int(NUM_PROFILES/4*2)]
xlim_3 = [int(NUM_PROFILES/4*2), int(NUM_PROFILES/4*3)]
xlim_4 = [int(NUM_PROFILES/4*3), NUM_PROFILES]

Y_LIMS = [800, 0]

fig, axs = plt.subplots(4, 1, #height_ratios=[1, 4],
                        figsize = [12,9],
                        layout='constrained', sharex=False)


profile_lims = xlim_1
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)
```

```
ax = axs[0]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
axs[0].set_ylim(Y_LIMS)

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

x = bad_profiles2.where(bad_profiles2.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='purple')

########
profile_lims = xlim_2
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[1]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
ax.set_ylim(Y_LIMS)

fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')


x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

x = bad_profiles2.where(bad_profiles2.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='purple')


######
profile_lims = xlim_3
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[2]
```

```
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
ax.set_ylim(Y_LIMS)

x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

x = bad_profiles2.where(bad_profiles2.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='purple')

#########
profile_lims = xlim_4
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[3]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['salinity'],rasterized=True)
ax.set_ylim(Y_LIMS)
x = bad_profiles.where(bad_profiles.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='r')

x = bad_profiles2.where(bad_profiles2.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='purple')



fig.colorbar(pc, ax=ax, label = 'Salinity [psu]')
axs[0].set_title('Salinity',loc='left')

print('Zooming in along the glider deployment to visualize salinity spikes. Red␣
 ↪arrows identify profiles already flagged as QC4.')
print('Purple arrows show profiles that will be manually removed.' )
```

```
Zooming in along the glider deployment to visualize salinity spikes. Red arrows
identify profiles already flagged as QC4.
Purple arrows show profiles that will be manually removed.
```



There is really bad asymmetry near the suface in a lot of profiles. These will be flagged as Q3.

[25]:
```
ds['salinity_QC'] = xr.where(ds.profile_index.
 ↪isin(bad_profiles2),3,ds['salinity_QC'])
ts['salinity_QC'] = xr.where(ts.profile_index.
 ↪isin(bad_profiles2),3,ts['salinity_QC'])
```

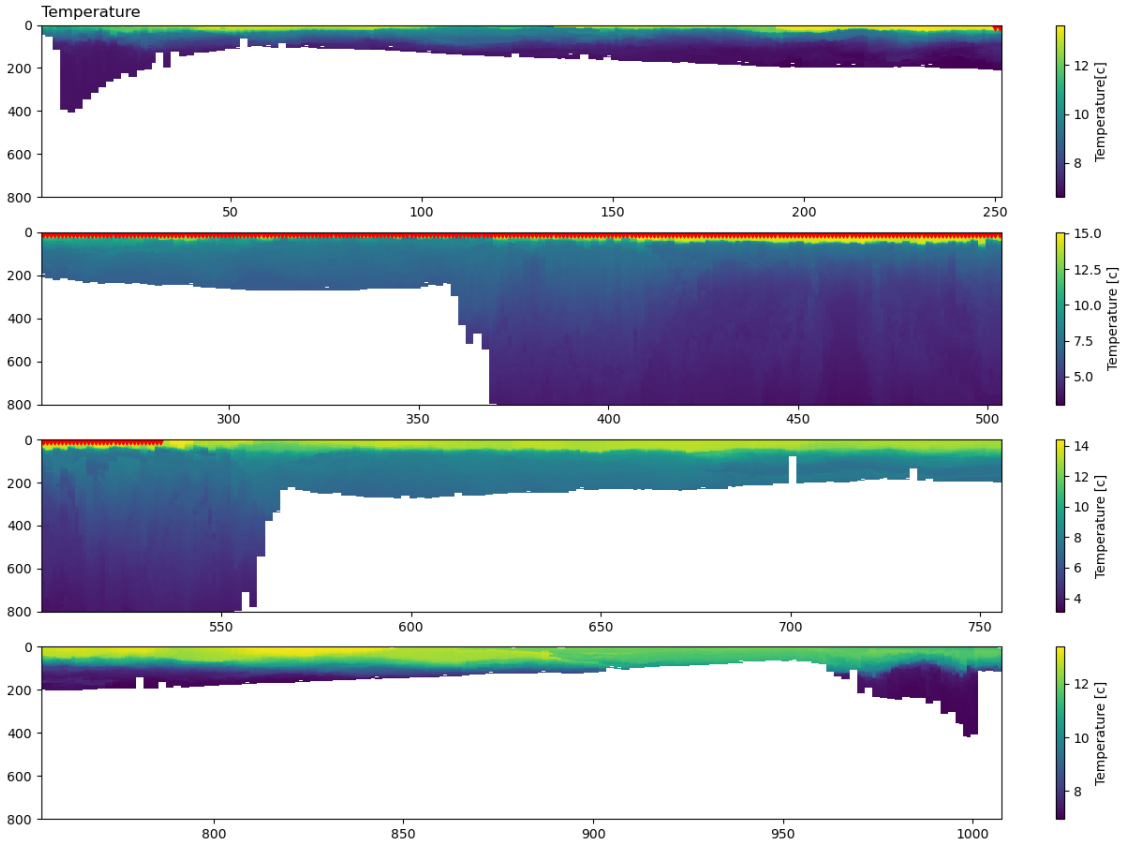
Biofouling would likely affect both T and S measurements. We would confirm that the bad profiles we identified are also where bad temperature values are.

[26]:
```
###### Manually identify questionable profiles
bad_profiles3 = bad_profiles2

# Now adding zoomed plot
xlim_1 = [0, int(NUM_PROFILES/4)]
xlim_2 = [int(NUM_PROFILES/4), int(NUM_PROFILES/4*2)]
xlim_3 = [int(NUM_PROFILES/4*2), int(NUM_PROFILES/4*3)]
xlim_4 = [int(NUM_PROFILES/4*3), NUM_PROFILES]

Y_LIMS = [800, 0] #######modified
```

```
fig, axs = plt.subplots(4, 1, #height_ratios=[1, 4],
                        figsize = [12,9],
                        layout='constrained', sharex=False)



profile_lims = xlim_1
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)
#ds_sub = ds_sub.isel(depth=range(200,800))

ax = axs[0]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['temperature'],rasterized=True)
axs[0].set_ylim(Y_LIMS)

x = bad_profiles3.where(bad_profiles3.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='red')

fig.colorbar(pc, ax=ax, label = 'Temperature[c]')
axs[0].set_title('Temperature',loc='left')


########
profile_lims = xlim_2
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[1]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['temperature'],rasterized=True)
ax.set_ylim(Y_LIMS)

x = bad_profiles3.where(bad_profiles3.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='red')

fig.colorbar(pc, ax=ax, label = 'Temperature [c]')
axs[0].set_title('Temperature',loc='left')

######
profile_lims = xlim_3
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[2]
```

```
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['temperature'],rasterized=True)
ax.set_ylim(Y_LIMS)

x = bad_profiles3.where(bad_profiles3.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='red')

fig.colorbar(pc, ax=ax, label = 'Temperature [c]')
axs[0].set_title('Temperature',loc='left')

#########
profile_lims = xlim_4
ds_sub = ds.where((ds.profile >=profile_lims[0]) & (ds.profile <=␣
 ↪profile_lims[1]), drop=True)

ax = axs[3]
pc = ax.pcolormesh(ds_sub.profile, ds_sub.depth,␣
 ↪ds_sub['temperature'],rasterized=True)
ax.set_ylim(Y_LIMS)

x = bad_profiles3.where(bad_profiles3.isin(ds_sub.profile),drop=True)
y = 0 + np.zeros_like(x)
ax.scatter(x,y,60,marker='v',color='red')

fig.colorbar(pc, ax=ax, label = 'Temperature [c]')
axs[0].set_title('Temperature',loc='left')

print('Zooming in along the glider deployment to visualize temperature spikes.␣
 ↪Red arrows identify some spikes to be manually flagged.')
```

```
Zooming in along the glider deployment to visualize temperature spikes. Red
arrows identify some spikes to be manually flagged.
```



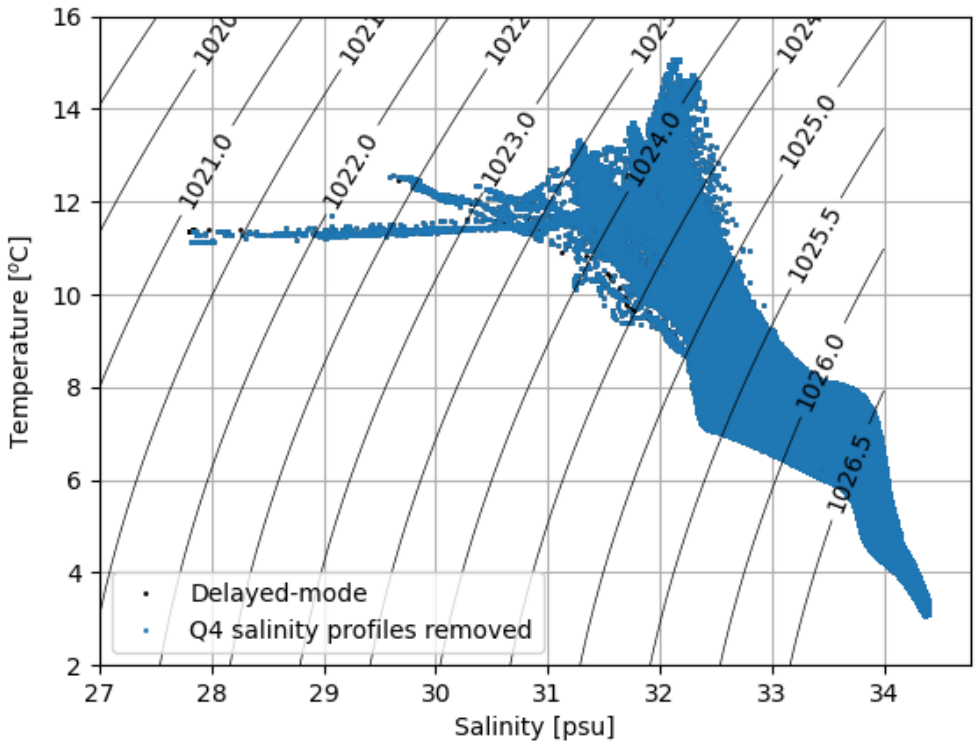
We also see that bad asymmetry near the syrface in the temperature data.

[27]:
```
ds['temperature_QC'] = xr.where(ds.profile_index.isin(bad_profiles3),3,1)
ts['temperature_QC'] = xr.where(ts.profile_index.isin(bad_profiles3),3,1)
```

[28]:
```
# T-S diagram to select near-surface water density range to exclude

srate = stats.mode((np.diff(ts.time)).astype('timedelta64[s]')).mode
fig,ax=plt.subplots()

ax.plot(ts.salinity,ts.temperature,'k.',markersize=2, label = 'Delayed-mode')

ts1 = ts.where((ts.salinity_QC!=4 )&(ts.temperature_QC!=4))
ax.plot(ts1.salinity, ts1.temperature, '.', markersize = 2, label = 'Q4␣
 ↪salinity profiles removed')

#Create a density grid to contour plot isopycnals
S_range = np.linspace(int(np.min(ts.salinity)-0.5),
                      int(np.max(ts.salinity)+0.5), 1000)
T_range = np.linspace(int(np.min(ts.temperature)-1),
                      int(np.max(ts.temperature)+1), 1000)
```

```
S_grid, T_grid = np.meshgrid(S_range, T_range)
density_grid = seawater.eos80.dens0(S_grid, T_grid)

CS = ax.contour(S_range, T_range, density_grid,
                np.arange(1014,
                          np.round(np.max(density_grid)),0.5),
                colors='k', linewidths=0.5);
ax.clabel(CS, CS.levels, inline=True, fontsize=10)
ax.set_xlabel('Salinity [psu]')
ax.set_ylabel('Temperature [$^o$C]')
# plt.xlim(28,35)
ax.grid()
ax.legend(prop={'size': 10})
print('Temperature vs. salinity diagram. ',
     'Black contours give density in kg/m^3:')
```

```
Temperature vs. salinity diagram.  Black contours give density in kg/m^3:
```



[29]:
```
###### grid to make finding bad profiles easier
ts.to_netcdf(f'{filepath}/{deploy_name}_QC3.nc')
# Save a gridded version as well
```

```
ds.to_netcdf(f'{filepath}/{deploy_name}_gridQC3.nc')
```

[30]:
```
display(Markdown('./docs/CTD_2_Sensor_lag_LT.md'))
```

3.3 2.3 Sensor alignment correction

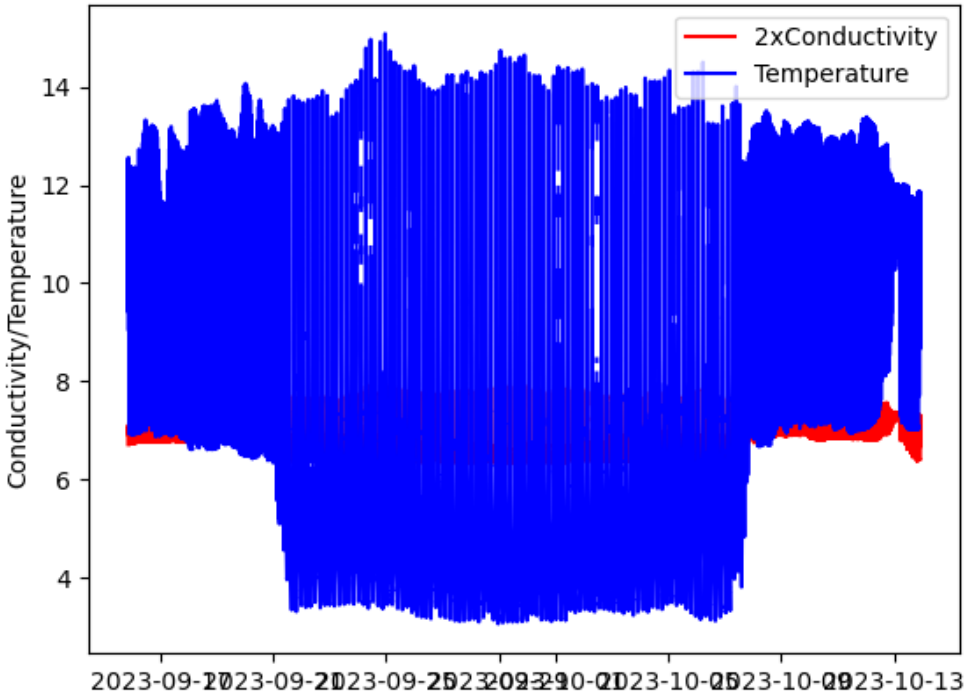
We now test application of a sensor alignment correction. In the literature this correction is often used to align the temperature and conductivity in time, relative to the pressure. This correction reduces the occurrence of salinity spikes near sharp gradients in T and S, and ensures calculations are made using the same parcel of water for all variables. The misalignment between the sensors is caused by: 1. The physical separation between sensors causing a transit time delay for water being pumped through the CTD, and, 2. Different sensor response times

We follow the SeaBird Electronics Data Processing Manual (page 80) to determine if there is any time lag between the temperature and conductivity sensors on our pumped CTD.

Sources: Sea-Bird Electronics, Inc. SEASOFT V2: SBE Data Processing (https://misclab.umeoce.maine.edu/ftp/instruments/CTD%2037SI%20June%202011%20disk/website/pdf_docun

[31]:
```
fig,ax=plt.subplots(sharex=True)

ts1 = ts.where(ts.conductivity_QC!=4)

ax.plot(ts1.time,2*ts1.conductivity, c='red', label='2xConductivity')
ax.set_ylabel('Conductivity/Temperature',)

ax.plot(ts1.time,ts1.temperature ,c='blue',label='Temperature')
ax.legend()

print('Plot timeseries of temperature and Q1 conductivity to observe any time␣
 ↪offset.')
```

```
Plot timeseries of temperature and Q1 conductivity to observe any time offset.
```



There is **no significant lag between the temperature (T) and conductivity (C) signals**. Examination of individual casts revealed no measurable offset between T and C.Furthermore, the T and C sensors on the Glider Pumped CTD (CPCTD) are spatially co-located, ebsuring both sensors sample the same parcel of water simultaneously. Therefore, **no sensor offset correction** was applied to the data.

[32]:
```
display(Markdown('./docs/CTD_2_Thermal_lag_Calvert.md'))
```

3.4 2.4 Thermal lag correction

The thermal lag effect is caused by the thermal inertia of the conductivity cell affecting the temperature of the water as it passes through the cell. To determine the thermal lag correction, the temperature inside the conductivity cell is estimated, then salinity is recalculated using the estimated temperature and the measured conductivity. To estimate the temperature, a recursive filter is applied to the temperature field with parameters (the amplitude of the error), and (the time constant for the thermal lag). Two methods for this are mentioned below.

Sea-Bird GPCTDs are pumped with a constant flow rate. As such, we expect the thermal lag to be approximately constant over the full mission, and it is sufficient to find a single value of and for the entire mission. It is ideal to use profile pairs from regions with large temperature gradients, but small conductivity gradients, when comparing up- and down-casts.

Janzen and Creed (2011) determined a cell thermal mass correction for the GPCTD using data from a prototype CTD that sampled twice as rapidly as the GPCTD nominally samples, with a pumped flow rate of 10 ml/s. **They found = 0.06 and = 10s.** These values are considered when retrieving α and τ to see how much the results differ.

In this study, we consistently find 10 s, in agreement with Janzen and Creed (2011), and therefore determine as the free parameter that minimizes the RMSD while holding = 10 s constant.

This mission on the **Calvert Line** occurred in a highly energetic environment, so near-surface differences between a downcast and the subsequent upcast are likely to be caused by spatiotemporal variability. As such, we exclude segments of each profile in the upper water column for which the **density is $<1023kg/m^3$** from the minimization routine.

Considerations for using Janzen and Creed values: Prior processing used Janzen and Creed (2011) values α and τ for the thermal lag. For each mission, the thermal lag parameters were directly estimated. The steps are outlined below, but can be found in much greater detail here: **https://cproof.uvic.ca/gliderdata/deployments/reports/C-PROOF_SBE_CTDProcessingReport_v0.2.pdf**

- It was confirmed that the directly estimated value of τ was within $\pm$10s of the Janzen and Creed (2011) value of 10s
- The improvement with the directly estimated, as well as Janzen and Creed, parameters was quantified.
- The thermal lag correction was applied with the parameters that resulted in the greater improvement, and did not result in an over-correction.
- If a given sensor had a directly estimated value of τ that is significantly higher or lower than 10s, investigate further

Thermal lag parameters typically vary slightly among individual GPCTD sensors, more so than alignment correction constants. **Once an optimal is identified for a particular sensor, subsequent missions are tested with those constants**, and the correction is applied only if it improves the profile agreement. If not, the parameters are re-evaluated.

With this method, the recursive filter seeks to minimize the root-mean squared difference (RMSD), which is calculated as the square root of the sum of the squared areas between pairs of salinity profiles (binned by temperature), normalized by the number of pairs of profiles. The values of and that minimize the area between pairs of profiles (each dive and subsequent climb along the glider path) were determined using a brute force minimization scheme. This method also uses a subset of the remaining data, consisting of **100 pairs of profiles** equally spaced in time, to determine the correction.

Updated thermal lag correction procedure: Same to the above, this is based on Morison et al's (2011) second method which derives a modified temperature that is the "best guess" for what the temperature is in the conductivity cell based on the temperature observed by the thermistor. The temperature is corrected using `lfilter` which is just a recursive filter:

$T_T(n) = -bT_T(n-1) + aT(n) - aT(n-1)$

where

$$a = \frac{4 f_n \alpha \tau}{1 + 4 f_n \tau}$$

and

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

τ can be thought of as the time constant of the thermal lag (in seconds) and α as its strength. Following Gaurau et al, f_n is the sampling frequency. The cell temperature is then"

$$T_c(n) = T(n) - T_T(n)$$

and can be used to calculate salinity with the measured conductivity and pressure.

3.4.1 2.4.1 Pre-processing steps:

We **exlude** profiles from the correction for which the area between subsequent downcasts is more than one standard deviation from the mission mean. This ensures that no data crossing fronts or intrusions is included in the correction, in line with the key assumption that a downcast and the subsequent upcast be identical.

Furthermore, we impose a cutoff for the area between pairs of profiles that will be included in the subset used to estimate the parameters. Any pair of profiles whose area is more than 3 standard deviations away from the mean will be excluded from the determination of the RMSD. This ensures that a small number of anomalous profiles do not bias the results.

During this step, the **suspicious salinity profiles** identified earlier are excluded as well.

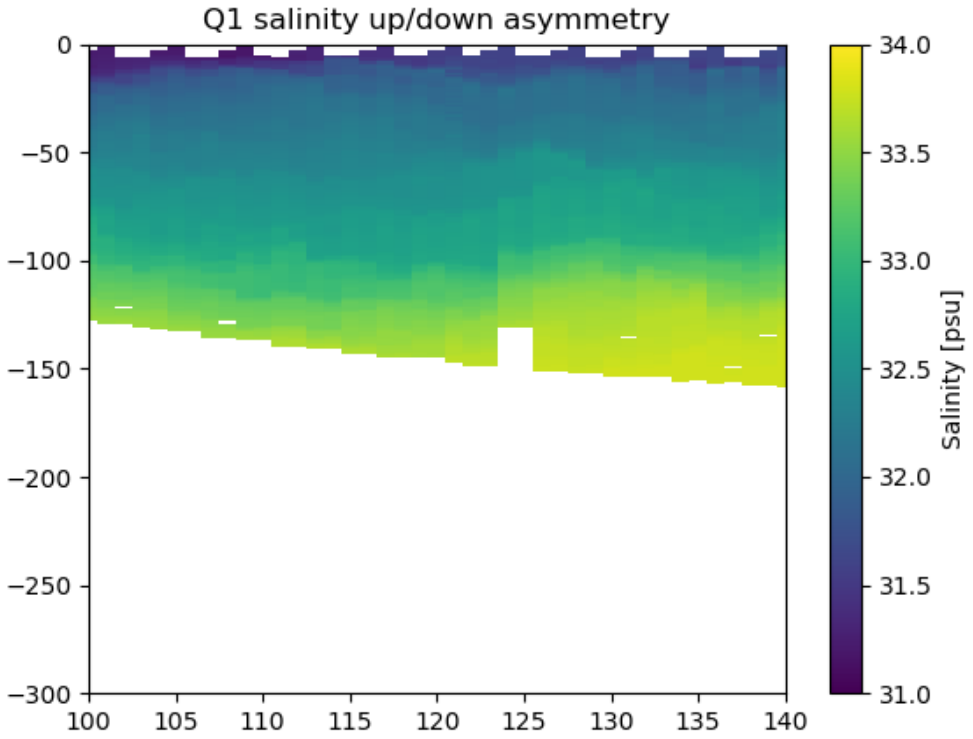
[33]:
```
# Set up our constants

density_cutoff = 1023 #this is not excluding the top of profiles (exclude␣
 ↪everything less dense than this from the minimization)
num_profs = 100 #number of profiles to include in the subset of data # This is␣
 ↪not used for the correction, but is used in the q/c steps
clean_profs_start = 50 #number of profiles to exclude from the start
clean_profs_end = 0 #number of profiles to exclude from the end
dn_stdev = 1 #how many standard deviations from the mean the area between␣
 ↪downcasts can be

# Load time series
ts = xr.open_dataset(f'{filepath}/{deploy_name}_QC3.nc')
# Save a gridded version as well
ds = xr.open_dataset(f'{filepath}/{deploy_name}_gridQC3.nc')

srate = stats.mode((np.diff(ds.time)).astype('timedelta64[s]')).mode
fs = 1/srate.astype(float)
fn = 0.5*fs #frequency for Sea-Bird GPCTD

ts = ts.assign_coords(pind=ts.profile_index) #add a profile index coordinate
```

```
tot_profs = int(np.nanmax(ts.profile_index.values))
print('Total number of profiles:', tot_profs)

####keep values with Q1 data
ds1 = ds.where(ds.salinity_QC!=4)
ts1 = ts.where(ts.salinity_QC!=4)
```

```
Total number of profiles: 1007
```

[34]:
```
fig, ax = plt.subplots()
vmin=31
vmax=34
pc=ax.pcolormesh(ds1.profile, -ds1.depth,␣
 ↪ds1['salinity'],rasterized=True,vmin=vmin,vmax=vmax)
fig.colorbar(pc,label='Salinity [psu]')
ax.set_title('Q1 salinity up/down asymmetry')
ax.set_xlim(100,140)
ax.set_ylim(-300,0)


print('There is a visible asymmetry in the up and down casts of succesive␣
 ↪profiles, causing stipes to appear in the Q1 data' )
```

```
There is a visible asymmetry in the up and down casts of succesive profiles,
causing stipes to appear in the Q1 data
```



We **exlude the first 50 profiles**, which were primarily collected in the highly energetic environment on the shelf near the shelf. We use a subet of the remianing data, consisting of **20 pairs of profiles** equally spaced in time to determine the corrected.

[35]:
```
print('Calculating profile pairs')

bad_profiles = ts.profile_index.where(ts.salinity_QC==4) ##don't include Q4 data

ts_sub, profile_bins, profile_bins_all, direction = pgs.profile_pairs(
    ts1, clean_profs_start, clean_profs_end, num_profs, bad_profiles
)

# Identify boolean index for application of density cutoff
density_bool = ts_sub.density>=density_cutoff
```

```
Calculating profile pairs
```

Restricting profiles $>$ **1 standard deviation** greater than the mean space between profiles

[36]:
```
#Determine the area between subsequent downcasts to restrict profiles included␣
 ↪in correction
print(f'Restricting profiles')
```

```
dn_area, area_bad = pgs.TS_preprocess(
    density_bool, dn_stdev,
    profile_bins, profile_bins_all,
    direction, ts_sub)
print('Max and min area between downcasts = ', np.nanmax(dn_area), np.
 ↪nanmin(dn_area))

ts_bad = ts_sub.where(
    ts_sub.profile_index.isin(
        profile_bins_all[area_bad]),
        drop=True)
prof_list = ts_bad.profile_index
print('List of profiles to exclude:', np.unique(prof_list.values))
```

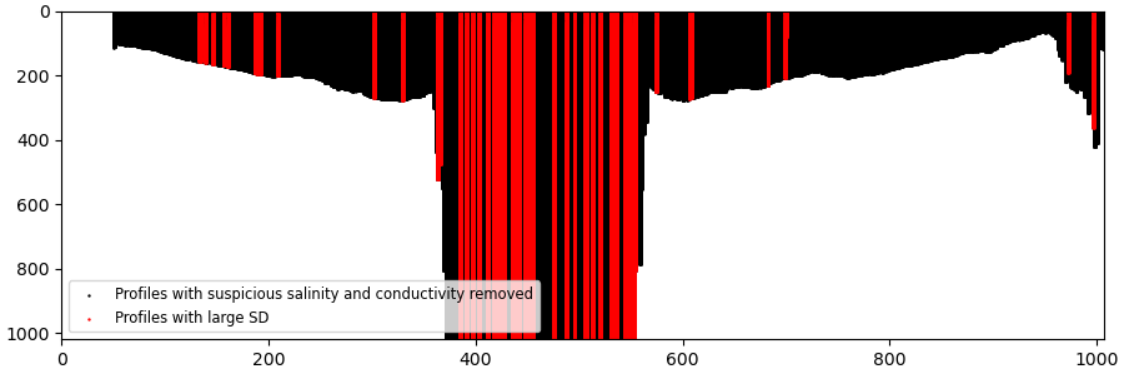
```
Restricting profiles
Max and min area between downcasts =  1.2153072351509757 9.381148700995484e-05
List of profiles to exclude: [132. 133. 134. 135. 138. 139. 146. 147. 156. 157.
160. 161. 186. 187.
 188. 189. 192. 193. 208. 209. 301. 302. 329. 330. 363. 364. 365. 366.
 385. 386. 391. 392. 397. 398. 403. 404. 411. 412. 417. 418. 421. 422.
 423. 424. 427. 428. 435. 436. 439. 440. 441. 442. 447. 448. 449. 450.
 451. 452. 455. 456. 475. 476. 487. 488. 495. 496. 505. 506. 507. 508.
 512. 513. 520. 521. 530. 531. 532. 533. 536. 537. 544. 545. 546. 547.
 548. 549. 552. 553. 554. 555. 574. 575. 606. 607. 608. 609. 682. 683.
 698. 699. 700. 701. 972. 973. 996. 997.]
```

[37]:
```
ts_sub['profiles_to_exclude'] = ts_sub.profile_index.isin(prof_list.values)
```

[38]:
```
# Plot the profiles that were kept for the comparison!!
print('Red indicates profile pairs that were identified in this process, where␣
 ↪the area between \nprofile pairs was considered to be too large, and so are␣
 ↪not included in the thermal lag correction. \nWhite bands indicate salinity␣
 ↪profiles removed during step 2.2.')

subdata = ts_sub.where(ts_sub.profiles_to_exclude == True) #profile_index.
 ↪isin(prof_list.values)#(profile_bins)

fig, ax = plt.subplots(1,1, figsize=(9, 3), constrained_layout=True)

ax.scatter(ts_sub.profile_index, ts_sub.pressure, marker = '.', color='black',␣
 ↪s = 2,
           rasterized=True, label='Profiles with suspicious salinity and␣
 ↪conductivity removed')


ax.set_ylim([MAX_DEPTH, 0])
ax.set_xlim([0,np.nanmax(ts_sub.profile_index)])
```

```
ax.scatter(subdata.profile_index, subdata.pressure,
           color='red', marker = '.', s=2, rasterized=True, label = 'Profiles␣
 ↪with large SD')

ax.legend(fontsize='small', loc='lower left');
```

```
Red indicates profile pairs that were identified in this process, where the area
between
profile pairs was considered to be too large, and so are not included in the
thermal lag correction.
White bands indicate salinity profiles removed during step 2.2.
```



[39]:
```
# SAVING INTERMEDIATE FILE TO NETCDF
ts_sub.to_netcdf(f'{filepath}/{deploy_name}_goodprofiles.nc')
# Save a gridded version as well
outfile = make_gridfiles(f'{filepath}/{deploy_name}_goodprofiles.nc',␣
 ↪f'{filepath}', deployfile, fnamesuffix='goodprofiles')
```

3.4.2 2.4.2 Defining the range to calculate α and τ:

From examining the asymmetry in up-down profiles, we manually choose a range to apply the thermal lag correction to. It is ideal to pick areas with high temperature gradients in the water column, but generally low salinity gradients.

[40]:
```
# Select a subset of profiles to calculate tau and alpha
profile_lims = [620,640]
print(f'Using profile limits {profile_lims} for tau and alpha calculation')
```

```
Using profile limits [620, 640] for tau and alpha calculation
```

[41]:
```
fname = f'{filepath}/{deploy_name}_goodprofiles.nc'
gridfname= f'{filepath}/{deploy_name}_gridgoodprofiles.nc'

ds=xr.open_dataset(f'{filepath}/{deploy_name}_gridgoodprofiles.nc')
```

```
tbins = ds.temperature.mean(dim='time', skipna=True)
tbins = np.sort(tbins[::6])
tbins = tbins[np.isfinite(tbins)]
# print(tbins)
depbins = ds.depth[::6]
```

[42]:
```
# Switches back to the time series....
# switching ds to ts
with xr.load_dataset(fname) as ds0:
    # USING PROFILE_LIMS DECIDED ABOVE!
    # print(f'Loading {fname}')
    inds=np.arange(profile_lims[0], profile_lims[1])

    indbins = np.arange(inds[0]-0.5, inds[-1]+0.5, 1.0)

    ts = ds0.where((ds0.profile_index >= inds[0]) & (ds0.profile_index <=␣
 ↪inds[-1]), drop=False)
    #ts = ts.where(ts.depth >=200) ######### modified... just look deeper in␣
 ↪the water column

    # Once again, make sure to use density cutoff
    ts = ts.where((ts.density > density_cutoff), drop=True)
    # Also, profiles to exclude:
    ts = ts.where(ts.profiles_to_exclude == False, drop=True)

    sal = ts.salinity
```

3.4.3 Comparing the error measurements between estimated alpha & tau and Janzen and Creed values:

Below shows the subset of profiles, limited to 200 m depth, without any thermal lag correction, and using literature values (Janzen and Creed 2011). The Janzen and Creed α and τ values visibly reduce the error in the water column, but better results can be retrieved by calculating our own.

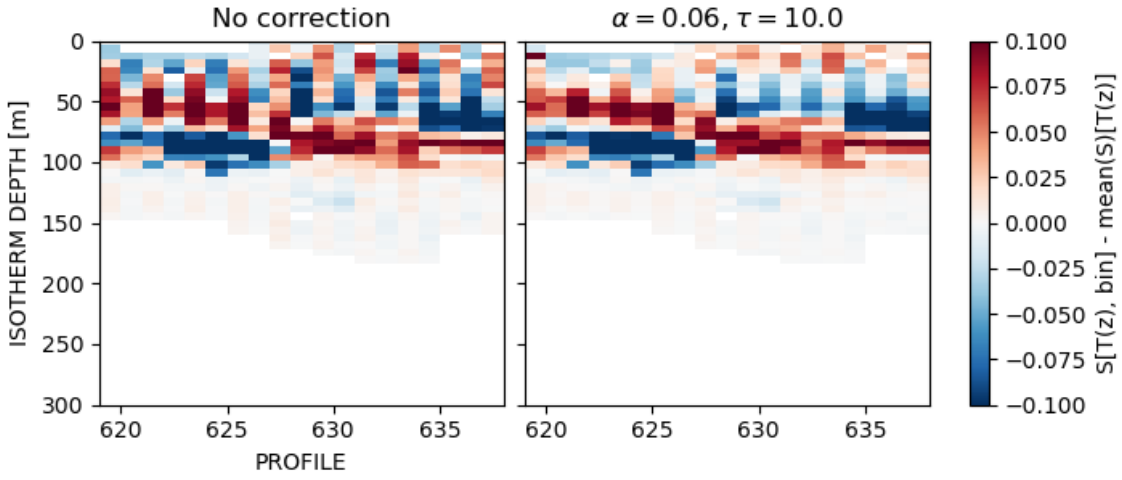
First we **bin our salinity data** into temperature bins of width 0.1 and profile index. We **sum the salinities in each bin and divide by the number of samples in each bin**. Error is the **difference in the mean salinity for successive profiles**, normlaized by salinity variance for each temperature bin.

[43]:
```
# Comparison - correcting the salinity with the janzen and creed values, and␣
 ↪comparing the error before and after
dt = ts.time.diff(dim='time').mean(dim='time').astype('float') / 1e9
fn = 1.0 / dt
alpha = 0.06
tau = 10.0
```

```
sal = pgs.correct_sal(ts, fn, alpha, tau) ###applied lag correction formula to␣
 ↪"good" salinity values

ss0, err0, totalerr = pgs.get_error(ts, ts.salinity, tbins, indbins) ### "good"␣
 ↪data with no lag correction
ss, err, totalerr = pgs.get_error(ts, sal, tbins, indbins)
```

[44]:
```
sp0 = np.nanmean(ss0, axis=1)

Y_LIMS = [300,0]
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True,␣
 ↪layout='constrained',figsize = [7,3])
pc = ax[0].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss0-sp0[:, np.
 ↪newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
ax[0].set_ylim(Y_LIMS)
ax[0].set_ylabel('ISOTHERM DEPTH [m]')
ax[0].set_xlabel('PROFILE')
ax[0].set_title('No correction')

pc = ax[1].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss-sp0[:, np.
 ↪newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
ax[1].set_title(f'$\\alpha = {alpha}, \\tau = {tau}$')
fig.colorbar(pc, ax=ax, label='S[T(z), bin] - mean(S)[T(z)]')
```

[44]:
```
<matplotlib.colorbar.Colorbar at 0x377918050>
```



As seen above, the up/down asymmetry slightly improved when applying the Janzen and Creed constants.

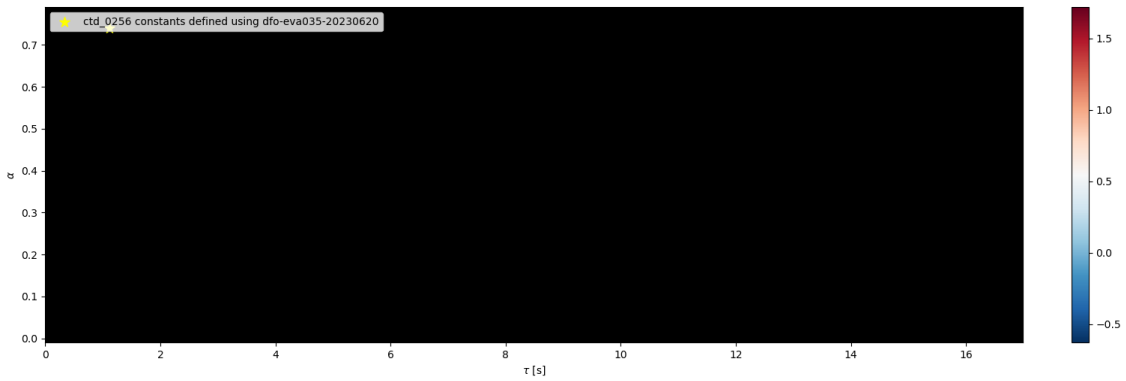
3.4.4 Finding alpha and tau values with lowerest error estimates:

[45]:
```
## scan:
alphas = np.arange(0, 0.8, 0.02) #cannot be negative; it is an amplitude␣
 ↪scaling of how strongly the T mismatch affects C
taus = np.arange(0, 17, 0.02)


errors = np.zeros((len(alphas), len(taus)))
for ny, alpha in enumerate(alphas):
    for nx, tau in enumerate(taus):

        sal = pgs.correct_sal(ts, fn, alpha, tau)
        ss, err, totalerr = pgs.get_error(ts, sal, tbins, indbins)
        errors[ny, nx] = np.nansum(np.nanmean(err, axis=1))
```

[46]:
```
fig, ax = plt.subplots(layout='constrained',  figsize = [15,5])
pc = ax.pcolormesh(taus, alphas, np.log10(errors-np.min(errors[:])),
                   cmap='RdBu_r', edgecolors='k', rasterized=True, linewidth =␣
 ↪0.1)
fig.colorbar(pc)
ax.set_xlabel('$\\tau$ [s]')
ax.set_ylabel('$\\alpha$')

safe_errors= np.log10(errors-np.min(errors[:]))
safe_errors[~np.isfinite((safe_errors))]=np.inf #will only keep finite numbers
min_idx = np.unravel_index(np.argmin(safe_errors), errors.shape)

best_tau = taus[min_idx[1]]
best_alpha = alphas[min_idx[0]]


print('The range of alpha and tau values tested, with log10-transformed errors␣
 ↪coloured. The dark blue indicates the alpha and tau with lowest errors:')
print('the best tau = ' + str(best_tau))
print('the best alpha = ' + str(best_alpha))

ax.scatter(1.12,0.74,marker='*', s=100,c='yellow',label='ctd_0256 constants␣
 ↪defined using dfo-eva035-20230620')
ax.legend(loc='upper left')
```

```
The range of alpha and tau values tested, with log10-transformed errors
coloured. The dark blue indicates the alpha and tau with lowest errors:
the best tau = 4.3
the best alpha = 0.22
```

[46]:
```
<matplotlib.legend.Legend at 0x3779c96d0>
```



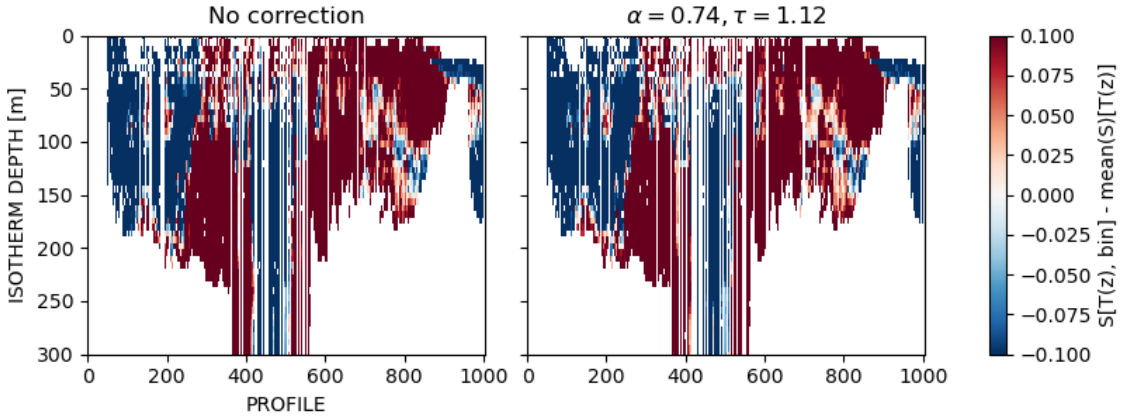
The error follows a trough-like shape, close to our constants. We will test if using the constants defined in dfo-eva035-20230620 that reduce the error correct the up/down asymmetry at the beginning, middle, and end of the data.

[47]:
```
dt = ts.time.diff(dim='time').mean(dim='time').astype('float') / 1e9
fn = 1.0 / dt
alpha = 0.74
tau = 1.12

print('******')
print(f'Applying alpha = {alpha} and tau = {tau}')
print('*****')

alpha2 = 0
tau2 =0
##########

with xr.load_dataset(fname) as ts0:
    inds = np.arange(0, NUM_PROFILES)

    indbins = np.arange(inds[0]-0.5, inds[-1]+0.5, 1.0)

    ts = ts0.where((ts0.profile_index >= inds[0]) & (ts0.profile_index <=␣
 ↪inds[-1]), drop=False)
    # Once again, make sure to use density cutoff? maybe?
    ts = ts.where((ts.density > density_cutoff), drop=True)
    # Also, profiles to exclude:
    ts = ts.where(ts.profiles_to_exclude == False, drop=True)

    sal = pgs.correct_sal(ts, fn, alpha, tau)

    ss0, err0, totalerr = pgs.get_error(ts, ts.salinity, tbins, indbins)
    ss, err, totalerr = pgs.get_error(ts, sal, tbins, indbins)
```

```
    sal2 = pgs.correct_sal(ts, fn, alpha2, tau2)
    ss2, err2, totalerr2 = pgs.get_error(ts, sal2, tbins, indbins)


Y_LIMS = [300,0]

fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, layout='constrained',␣
 ↪figsize = [8,3])
sp0 = np.nanmean(ss0, axis=1)
pc = ax[0].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss0-sp0[:, np.
 ↪newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
ax[1].set_ylim([200, 0])
ax[0].set_ylabel('ISOTHERM DEPTH [m]')
ax[0].set_xlabel('PROFILE')
ax[0].set_title('No correction')
ax[0].set_ylim(Y_LIMS)

pc = ax[1].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss-sp0[:, np.
 ↪newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
ax[1].set_title(f'$\\alpha = {alpha}, \\tau = {tau}$')
fig.colorbar(pc, ax=ax, label='S[T(z), bin] - mean(S)[T(z)]')
```

```
******
Applying alpha = 0.74 and tau = 1.12
*****
```

[47]: <matplotlib.colorbar.Colorbar at 0x37780f890>

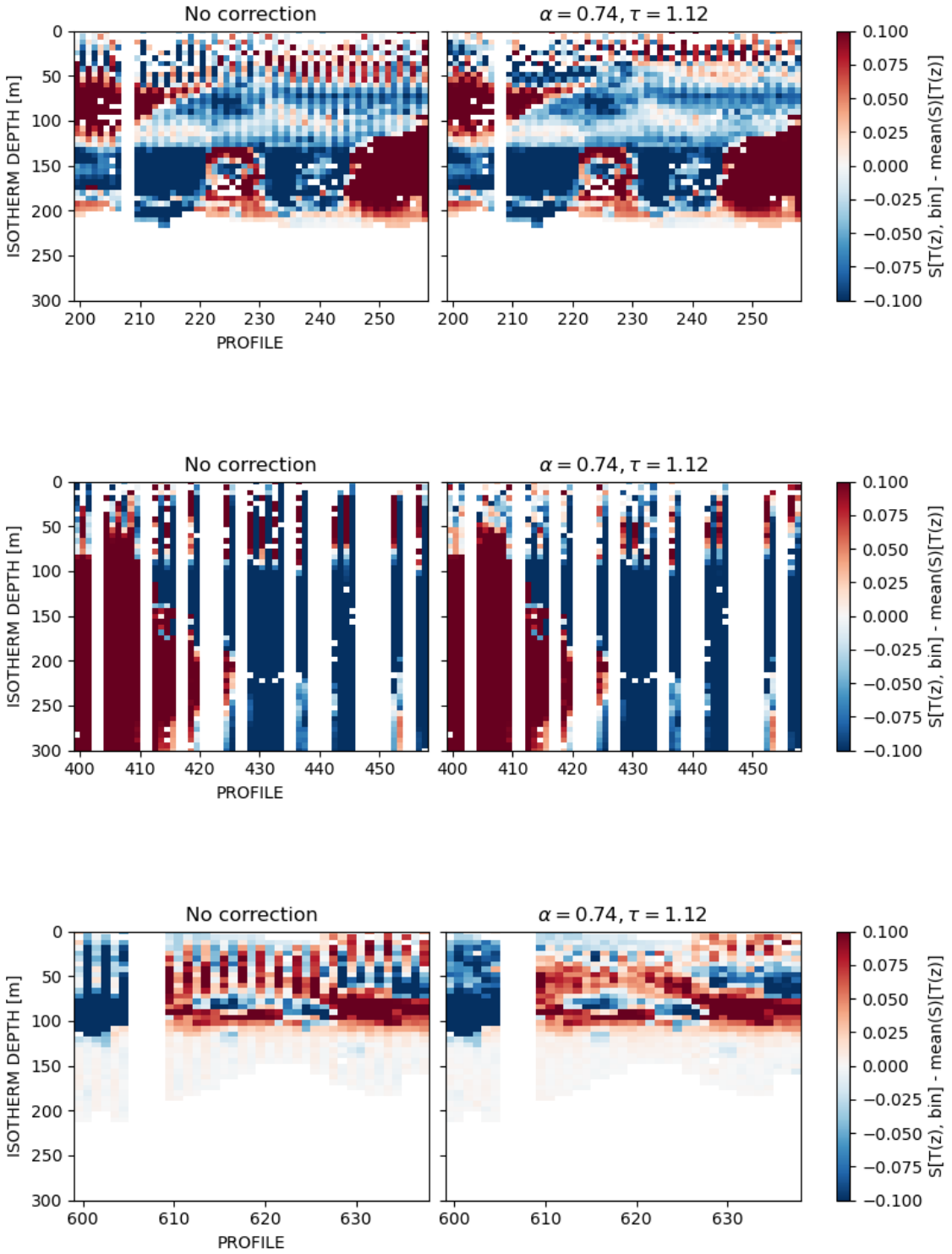


Zoom into three areas to observe change

[48]:
```
def zoom_in_thermal_chg(min_ind,max_ind,fname,density_cutoff):
    with xr.load_dataset(fname) as ts0:
        inds = np.arange(min_ind, max_ind)
```

```
        indbins = np.arange(inds[0]-0.5, inds[-1]+0.5, 1.0)
        ts = ts0.where((ts0.profile_index >= inds[0]) & (ts0.profile_index <=␣
↪inds[-1]), drop=False)
        # Once again, make sure to use density cutoff? maybe?
        ts = ts.where((ts.density > density_cutoff), drop=True)
        # Also, profiles to exclude:
        ts = ts.where(ts.profiles_to_exclude == False, drop=True)

        sal = pgs.correct_sal(ts, fn, alpha, tau)

        ss0, err0, totalerr = pgs.get_error(ts, ts.salinity, tbins, indbins)
        ss, err, totalerr = pgs.get_error(ts, sal, tbins, indbins)

        sal2 = pgs.correct_sal(ts, fn, alpha2, tau2)
        ss2, err2, totalerr2 = pgs.get_error(ts, sal2, tbins, indbins)


    Y_LIMS = [300,0]

    fig, ax = plt.subplots(1, 2, sharex=True, sharey=True,␣
↪layout='constrained', figsize = [8,3])
    sp0 = np.nanmean(ss0, axis=1)
    pc = ax[0].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss0-sp0[:
↪, np.newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
    ax[1].set_ylim([200, 0])
    ax[0].set_ylabel('ISOTHERM DEPTH [m]')
    ax[0].set_xlabel('PROFILE')
    ax[0].set_title('No correction')
    ax[0].set_ylim(Y_LIMS)

    pc = ax[1].pcolormesh(indbins[:-1], depbins[:len(tbins)-1][::-1], ss-sp0[:,␣
↪np.newaxis], cmap='RdBu_r', vmin=-0.1, vmax=0.1) #, vmin=3.34, vmax=3.52)
    ax[1].set_title(f'$\\alpha = {alpha}, \\tau = {tau}$')
    fig.colorbar(pc, ax=ax, label='S[T(z), bin] - mean(S)[T(z)]')
```

[49]:
```
zoom_in_thermal_chg(200,260,fname,density_cutoff)
zoom_in_thermal_chg(400,460,fname,density_cutoff)
zoom_in_thermal_chg(600,640,fname,density_cutoff)
```



The applied constants reduced the up- and down-cast asymmetry at the beginning, middle, and end of the time series. Therefore, the thermal lag correction is applied to the dataset.

4 Save corrected data :

These fields, adjusted and re-calculated using the new alpha and tau values, are saved in the output file as fields **salinity_adjusted**, **temperature_adjusted** and **density_adjusted**. The original delayed-mode, uncorrected fields are saved as **salinity**, **temperature** and **density**.

[50]:
```
print('*****')
print(f'Saving with alpha = {alpha} and tau = {tau} applied')
print('*****')

# We're going to apply the changes to the data with quality control flags data. That step happened last here:

ts = xr.open_dataset(f'{filepath}/{deploy_name}_QC3.nc')
ds = xr.open_dataset(f'{filepath}/{deploy_name}_gridQC3.nc')

##apply thermal lag correction
s, t, d = pgs.correct_sal_temp_dens(ts, fn, alpha, tau)

########
ts.attrs['processing_details'] = 'Processing details are located on the C-PROOF website for this mission under the reports tab.'
ts.attrs['processing_tech'] = 'Lauryn Talbot; ltalbot@uvic.ca'
ts.attrs['citation'] = '"Klymak, J., & Ross, T. (2025). C-PROOF Underwater Glider Deployment Datasets [Data set]. Canadian-Pacific Robotic Ocean Observing Facility.doi:10.82534/44DS-K310"'
ts.attrs['references'] = 'https://doi.org/10.82534/44DS-K310'
ts.attrs['correction_constants'] = f'alpha = {alpha}; tau={tau}'

# Uncorrected conductivity
ts['conductivity'].attrs['comment'] = 'uncorrected conductivity'

# Adjusted (aka cleaned) conductivity

ts['conductivity_adjusted'] = ts.conductivity.copy()
ts['conductivity_adjusted'].attrs['comment'] = 'adjusted conductivity'
ts['conductivity_adjusted'].attrs['processing_report'] = processing_report
ts['conductivity_adjusted'].attrs['processing_date'] = processing_date
ts['conductivity_adjusted'].attrs['processing_date'] = processing_protocol
ts['conductivity_adjusted_QC'] =  ts['temperature_QC']#let users know these are not estimated values
ts['conductivity_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 = estimated data']
ts['conductivity_adjusted_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 = estimated data']

#remove conductivityClean - an intermediate step
```

```
ts = ts.drop_vars('conductivityClean')


# Uncorrected temperature
ts['temperature'].attrs['comment'] = 'uncorrected temperature [degC]'

# Adjusted temperature
ts['temperature_adjusted'] = ts.temperature.copy()
ts['temperature_adjusted'].attrs['comment'] = 'temperature [degC]'
ts['temperature_adjusted'].attrs['processing_report'] = processing_report
ts['temperature_adjusted'].attrs['processing_date'] = processing_date
ts['temperature_adjusted'].attrs['processing_date'] = processing_protocol
ts['temperature_adjusted_QC'] =  ts['temperature_QC']#let users know these are␣
 ↪not estimated values
ts['temperature_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 =␣
 ↪estimated data']
ts['temperature_adjusted_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8␣
 ↪= estimated data']

# Uncorrected salinity
ts['salinity'].attrs['comment'] = 'uncorrected salinity [psu]'

# Corrected salinity
ts['salinity_adjusted'] = ('time', s)
ts['salinity_adjusted'].attrs['comment'] = f'adjusted salinity [psu] using a␣
 ↪thermal lag correction with alpha = {alpha} and tau = {tau} '
ts['salinity_adjusted'].attrs['method'] = ' '
ts['salinity_adjusted'].attrs['processing_report'] = processing_report
ts['salinity_adjusted'].attrs['processing_date'] = processing_date
ts['salinity_adjusted'].attrs['processing_protocol'] = processing_protocol
ts['salinity_adjusted_QC'] =  ts['salinity_QC'].where(ts['salinity_QC']!=1,8)
ts['salinity_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 = estimated␣
 ↪data']
ts['salinity_adjusted_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 =␣
 ↪estimated data']

# Unadjusted density
ts['density'].attrs['comment'] = 'unadjusted density'

# Adjusted density
ts['density_adjusted'] = ('time', d)
ts['density_adjusted'].attrs['comment'] = 'density from adjusted salinity [psu]␣
 ↪and temperature [degC]'
ts['density_adjusted'].attrs['method'] = ' '
ts['density_adjusted_QC'] =  ts['salinity_adjusted_QC'] #density is only as␣
 ↪good as the salinity values used to determine it
```

```
ts['density_QC'] = ts['salinity_QC']
ts['density_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 = estimated␣
 ↪data']
ts['density_adjusted_QC'].attrs['comment']=['1 = good data; 4 = bad data; 8 =␣
 ↪estimated data']
```

```
*****
Saving with alpha = 0.74 and tau = 1.12 applied
*****
```

Calculate potential_temperature_adjusted and potential_density_adjusted using adjusted_salinity

[51]:
```
long = ts.longitude.fillna(ts.longitude.mean(skipna=True))
lat = ts.latitude.fillna(ts.latitude.mean(skipna=True))
sa_adj = gsw.SA_from_SP(ts['salinity_adjusted'],ts['pressure'],long,lat)
ct_adj = gsw.CT_from_t(sa_adj,ts['temperature_adjusted'],ts['pressure'])
ts['potential_density_adjusted'] = (('time'),1000 + gsw.density.
 ↪sigma0(sa_adj,ct_adj).values)
ts['potential_density_adjusted'].attrs['comment'] = 'calculated using adjusted␣
 ↪salinity'
ts['potential_density_adjusted_QC'] = ts['salinity_adjusted_QC']

ts['potential_temperature_adjusted'] = (('time'),
                                        gsw.conversions.pt0_from_t(ts.
 ↪salinity_adjusted,ts.temperature_adjusted,ts.pressure).values)
ts['potential_temperature_adjusted'].attrs['comment'] = 'calculated using␣
 ↪adjusted salinity'
ts['potential_temperature_adjusted_QC'] = ts['salinity_adjusted_QC']
```

[52]:
```
##### Save our final datasets
ts.to_netcdf(f'{filepath}{deploy_name}_CTDadjusted.nc')
print(f'Corrected data saved to file: {filepath}/{glider_name}/{deploy_name}/
 ↪{deploy_name}_CTDadjusted.nc')
make_gridfiles(f'{filepath}{deploy_name}_CTDadjusted.nc',
                         f'{filepath}', deployfile, fnamesuffix='_CTDadjusted')
```

```
Corrected data saved to file: deployments/dfo-eva035/dfo-eva035-20230915//dfo-
eva035/dfo-eva035-20230915/dfo-eva035-20230915_CTDadjusted.nc
```

[52]:
```
'deployments/dfo-eva035/dfo-eva035-20230915//dfo-
eva035-20230915_grid_CTDadjusted.nc'
```

5 3.0 Summary of corrections applied to delayed mode data for this mission

Identification of anomalous conductivity values: * Anomalous conductivity values, values that still differ from the mean by more than **3 standard deviations**, are flagged as 'bad' (QC 4).

* Profiles with spikes in the salinity data from biofouling were flagged as Q4.

Identification of questionable salinity profiles: * Anomolous salinity profiles were flagged as Q4 if the data are **4 standard deviations** away from the overall mean for the salinity time series within a given temperature bin.

Sensor alignment correction: * No sensor alignment correction was applied.

Thermal lag correction: * The directly determined values for the thermal lag correction produced an improvement that was larger than the recommended values from Janzen and Creed (2011). * The correction overall significantly reduced the root-mean squared difference for the area between between pairs of profiles. * The final thermal lag correction was applied using the calculated values of:

[53]:
```
print(f'alpha = {alpha} and tau = {tau}')
```

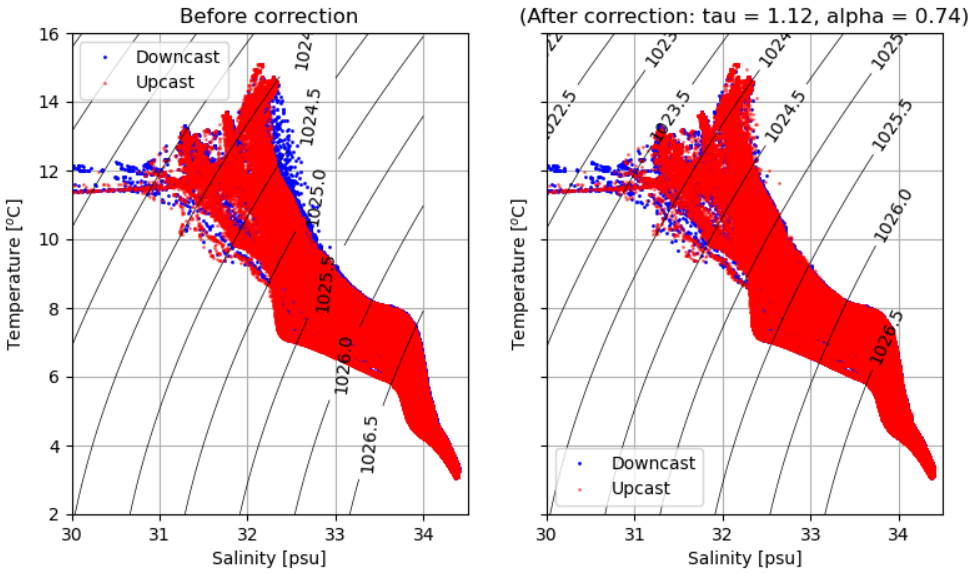
```
alpha = 0.74 and tau = 1.12
```

[54]:
```
ds=xr.open_dataset(f'{filepath}{deploy_name}_grid_CTDadjusted.nc')
ts=xr.open_dataset(f'{filepath}{deploy_name}_CTDadjusted.nc')
```

[55]:
```
#Compare the uncorrected and corrected data in T-S space

print('Temperature-salinity diagrams for all profiles, '
      'showing the difference between upcasts (red) and downcasts (blue), '
      'for the data without the thermal lag correction applied (left panel) and␣
 ↪'
      'the data with the thermal lag correction applied (right panel):')

x_lim=[30, 34.5]

#Plotting
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(9,5))

ind = np.where(ts.profile_direction.values== 1)[0]
ax[0].plot(ts.salinity[ind], ts.temperature[ind], 'b.', markersize=2,␣
 ↪rasterized=True, label = 'Downcast')
ax[1].plot(ts.salinity_adjusted[ind], ts.temperature_adjusted[ind], 'b.',␣
 ↪markersize=2, rasterized=True, label = 'Downcast')

ind = np.where(ts.profile_direction.values == -1)[0]
ax[0].plot(ts.salinity[ind], ts.temperature[ind], 'r.', markersize=2, alpha = 0.
 ↪5, rasterized=True, label = 'Upcast')
ax[1].plot(ts.salinity_adjusted[ind], ts.temperature_adjusted[ind], 'r.',␣
 ↪markersize=2, alpha = 0.5, rasterized=True, label = 'Upcast')

S_range = np.linspace(int(np.min(ts.salinity)-0.5),
                      int(np.max(ts.salinity)+0.5), 1000)
T_range = np.linspace(int(np.min(ts.temperature)-1),
```

```
                    int(np.max(ts.temperature)+1), 1000)
S_grid, T_grid = np.meshgrid(S_range, T_range)
density_grid = seawater.eos80.dens0(S_grid, T_grid)


CS = ax[0].contour(S_range, T_range, density_grid,
                np.arange(1021,np.round(np.max(density_grid)),0.5),
                colors='k', linewidths=0.5);
ax[0].clabel(CS, CS.levels, inline=True, fontsize=10)
ax[0].set_ylabel('Temperature [$^o$C]')
ax[0].set_xlabel('Salinity [psu]')
ax[0].set_title('Before correction')
ax[0].set_xlim(x_lim)
ax[0].grid()

CS = ax[1].contour(S_range, T_range, density_grid,
                np.arange(1021,np.round(np.max(density_grid)),0.5),
                colors='k', linewidths=0.5);
ax[1].clabel(CS, CS.levels, inline=True, fontsize=10)

ax[1].set_ylabel('Temperature [$^o$C]')
ax[1].set_xlabel('Salinity [psu]')
ax[1].set_title(f'(After correction: tau = {tau}, alpha = {alpha})' )
ax[1].grid()

ax[0].legend(prop={'size': 10});
ax[1].legend(prop={'size': 10});
```

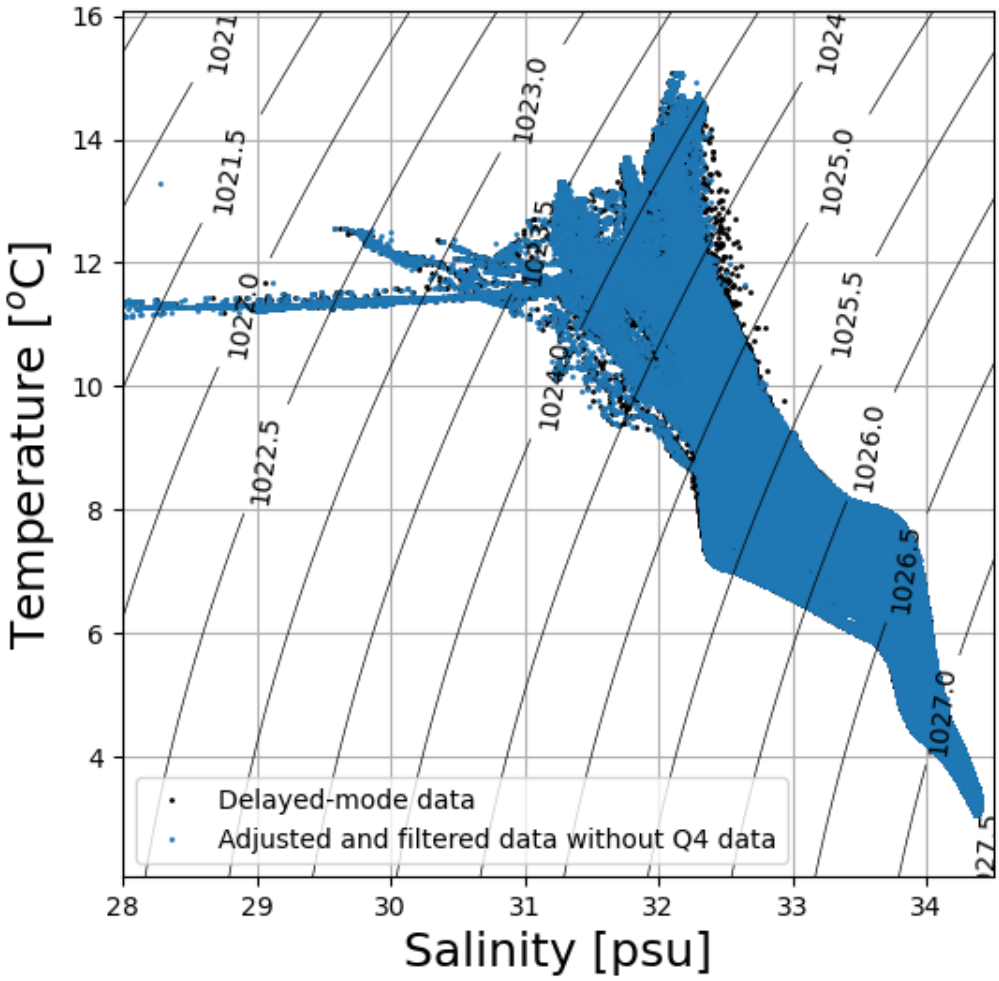
Temperature-salinity diagrams for all profiles, showing the difference between upcasts (red) and downcasts (blue), for the data without the thermal lag correction applied (left panel) and the data with the thermal lag correction applied (right panel):



[56]:
```
# Visualize the final data
fig, ax = plt.subplots(1, 1, figsize=(6, 6))
X_LIM = [28,34.5]
# T-S diagram for fully corrected data
ax0 = ax
ax0.plot(ts.salinity,ts.temperature,'k.',markersize=2, label = "Delayed-mode␣
 ↪data")


tsno4 = ts.where((ts.salinity_adjusted_QC !=4))
tsno4 = tsno4.where((ts.temperature_adjusted_QC !=4))

ax0.plot(tsno4.salinity_adjusted,tsno4.temperature_adjusted,'.',markersize=2,␣
 ↪label = "Adjusted and filtered data without Q4 data")

#Create a density grid to contour plot isopycnals
S_range = np.linspace(np.nanmin(ts.salinity_adjusted)-0.5,
                      np.nanmax(ts.salinity_adjusted)+0.5, 1000)
T_range = np.linspace(np.nanmin(ts.temperature_adjusted)-1,
                      np.nanmax(ts.temperature_adjusted)+1, 1000)
S_grid, T_grid = np.meshgrid(S_range, T_range)
density_grid = seawater.eos80.dens0(S_grid, T_grid)

CS = ax0.contour(S_range, T_range, density_grid,
                 np.arange(1014,
                           np.round(np.max(density_grid)),0.5),
```

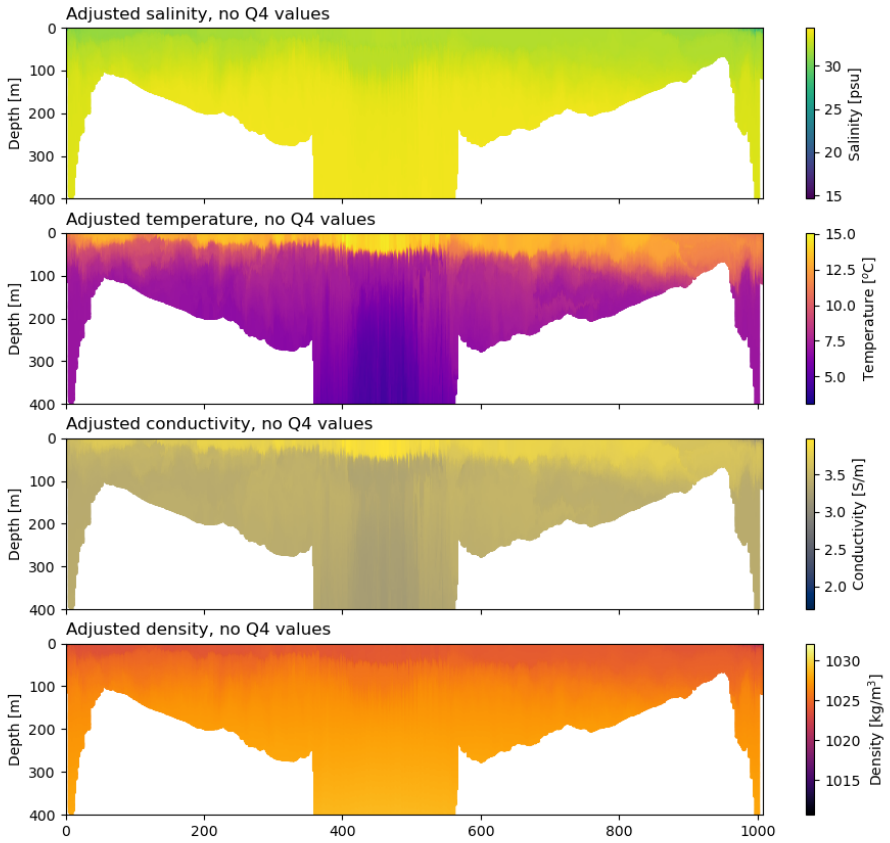
```
                colors='k', linewidths=0.5);
ax0.clabel(CS, CS.levels, inline=True, fontsize=10)
ax0.set_xlabel('Salinity [psu]', fontsize=18)
ax0.set_ylabel('Temperature [$^o$C]', fontsize=18)
ax0.set_xlim(X_LIM)
ax0.grid()
ax0.legend()

print('The corrected temperature and salinity fields '
      'shown in a T-S diagram with density contours:')
```

```
The corrected temperature and salinity fields shown in a T-S diagram with
density contours:
```



[57]:
```
# RE-PLOTTING WITH THE COND FILTER!
fig, axs = plt.subplots(4, 1, figsize=(11, 10), sharey=True, sharex=True)

xlims = [0, NUM_PROFILES]
ylims=[400,0]

pc = axs[0 ].scatter(tsno4.profile_index, tsno4.depth,c=␣
 ↪tsno4['salinity_adjusted'],s=2,rasterized=True)
axs[0].set_ylim(ylims)
axs[0].set_xlim(xlims)
fig.colorbar(pc, ax=axs[0], label = 'Salinity [psu]')
axs[0].set_title('Adjusted salinity, no Q4 values',loc='left')

pc = axs[1].scatter(tsno4.profile_index, tsno4.depth,␣
 ↪c=tsno4['temperature_adjusted'],s=2,rasterized=True,cmap='plasma')
fig.colorbar(pc, ax=axs[1], label = 'Temperature [$^o$C]')
axs[1].set_title('Adjusted temperature, no Q4 values',loc='left')

pc = axs[2].scatter(tsno4.profile_index, tsno4.depth,␣
 ↪c=tsno4['conductivity_adjusted'],s=2,rasterized=True,cmap='cividis')
fig.colorbar(pc, ax=axs[2], label = 'Conductivity [S/m]')
axs[2].set_title('Adjusted conductivity, no Q4 values',loc='left')

pc = axs[3].scatter(tsno4.profile_index, tsno4.depth,␣
 ↪c=tsno4['density_adjusted'],s=2,rasterized=True,cmap='inferno')
fig.colorbar(pc, ax=axs[3], label = 'Density [kg/m$^3$]')
axs[3].set_title('Adjusted density, no Q4 values',loc='left')

axs[0].set_ylabel('Depth [m]')
axs[1].set_ylabel('Depth [m]')
axs[2].set_ylabel('Depth [m]')
axs[3].set_ylabel('Depth [m]')

print('The adjusted salinity and temperature, shown with filtered conductivity␣
 ↪and adjusted density:')
```

```
The adjusted salinity and temperature, shown with filtered conductivity and
adjusted density:
```



[58]:
```
display(Markdown('./docs/CTD_References.md'))
```

6 References

1. Ferrari, R., and Rudnick, D. L. Thermohaline variability in the upper ocean, J. Geophys. Res., 105(C7), 16857-16883, 2000.

2. Garau, B., Ruiz, S., Zhang, W. G., Pascual, A., Heslop, E., Kerfoot, J., & Tintoré, J. Thermal Lag Correction on Slocum CTD Glider Data, J. Atmos. Oceanic Technol., 28(9), 1065-1071, 2011.

3. Janzen, C. D., and Creed, E. L. Physical oceanographic data from Seaglider trials in stratified coastal waters using a new pumped payload CTD, OCEANS'11 MTS/IEEE KONA, Waikoloa, HI, USA, 1-7, 2011.

4. Morison, J., Andersen, R., Larson, N., D'Asaro, E., & Boyd, T. The correction for thermal-lag effects in Sea-Bird CTD data, J. Atmos. Oceanic Technol., 11, 1151-1164, 1994.

5. Sea-Bird Seasoft V2:SBE Data Processing - CTD Data Processing & Plotting Software for Windows, Sea-Bird Scientific, software manual revision 7.26.8, 2017.

6. Sea-Bird User Manual - GPCTD Glider Payload CTD (optional DO) - Conductivity, Temperature, and Pressure (optional DO) Sensor with RS-232 Interface, Sea-Bird Scientific, manual version 008, 2021.

[ ]: