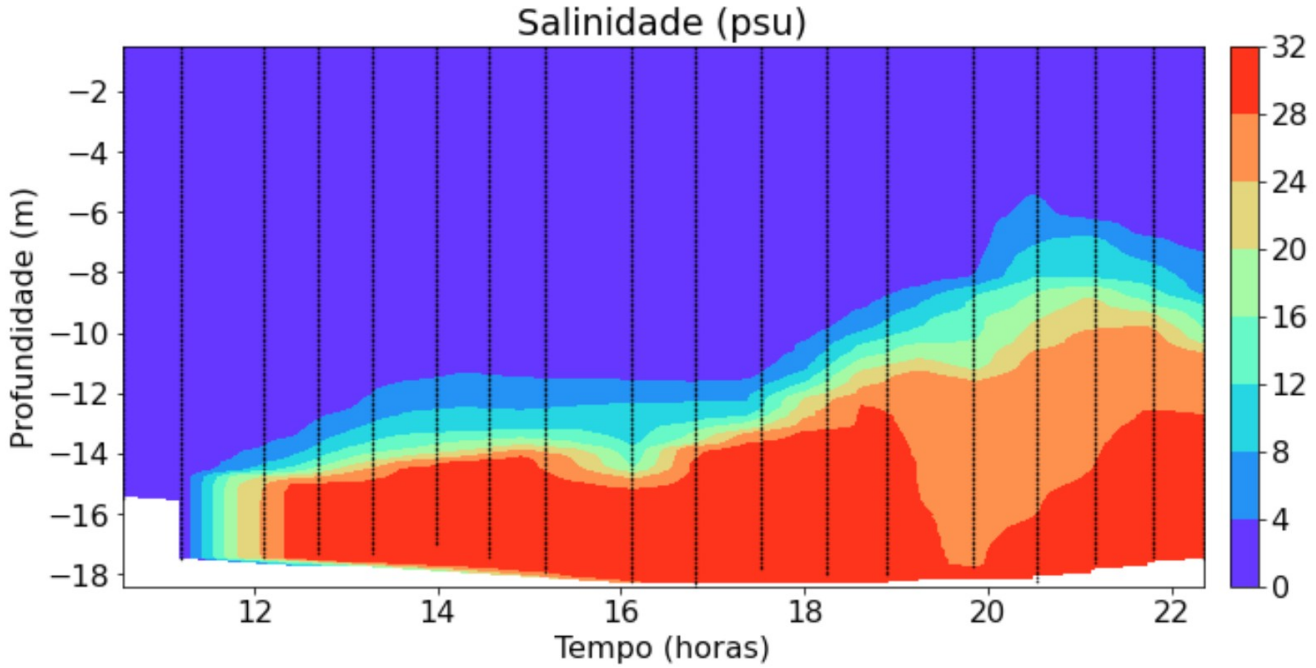
Analisando dados com Python

Fazendo um gráfico de distribuição temporal/vertical de salinidade



Carlos A.F. Schettini, 2021-Jan-19

Laboratório de Oceanografia Costeira e Estuarina

LOCostE / IO / FURG

Objetivos:

1. Carregar diversos arquivos de dados sequencialmente
2. Organizar uma base de dados
3. Interpolar os dados em uma matriz regular
4. Visualizar graficamente

Em um primeiro tutorial vimos como baixar dados de maré do IBGE pela Internet, processar e gerar um gráfico da variação do nível da água pelo tempo. Esse é um caso simples, desconsiderando a complexidade de se trabalhar com o tempo. Mas é o nível em função do tempo. A coisa fica um pouco mais complexa quando temos que fazer um gráfico de distribuição de alguma variável (ex: salinidade) em duas dimensões. Em especial quando as informações não estão regularmente arranjadas nas dimensões.

Neste tutorial vamos gerar um gráfico da distribuição da salinidade vertical e temporal, obtidos em um levantamento de dados para acompanhar variações mareais. São as tais campanhas de 12 ou 13 horas. É uma abordagem Euleriana, onde em um ponto fixo são realizadas medições de variáveis (correntes, salinidade, etc...) ao longo do tempo. No caso, o período de maré semi-diurno tem 12.4 horas, então faz-se as medições em intervalos de 30' ou 1 hora, ou até mesmo 1,5 hora. Intervalo maior do que 1,5 hora é 'forçação de barra' e não permite resolver minimamente as variações intra-mareais das variáveis. Há outros 'poréns' a serem considerados no planejamento de uma campanha, mas não é assunto para agora.

Os dados que temos foram coletados durante um dia (~12 horas) no centro do canal do estuário da Lagoa dos Patos, em frente ao terminal de containers, usando uma sonda tipo CTD da marca JFE-Advantech, modelo Rinko Profiler. Mas informações sobre este instrumento em https://www.jfe-advantech.co.jp/eng/ocean/rinko/rinko-profiler.html. Lembrando que antes de analisar qualquer dado, é bom saber como esses dados foram gerados. No presente casos, em intervalos regulares o barco vai até o ponto de amostragem, o CTD é ligado, descido na coluna de água até chegar no fundo, depois é recolhido e desligado. Depois de um intervalo, o procedimento é repetido. Aqui o importante é que o perfil é realizado sempre no mesmo ponto. 'No mesmo ponto' não significa em absoluto que é 'no mesmo ponto', pois sempre há um erro de posicionamento. E, ainda que se use um DGPS com precisão sub-métrica, para baixar o CTD é preciso parar o barco e ocorre deriva… então 'o mesmo ponto' há um erro que pode estar nas dezenas de metros. E… nisso a profundidade pode variar consideravelmente! Se o fundo for irregular, uma deriva pode mudar drásticamente a profundidade local. Outro 'porém' é se o CTD chegou no fundo ou não. Alguns CTDs tem indicador de fundo. O Rinko não tem. E, quando a deriva é muito grande fica difícil saber se o CTD chegou no fundo ou não. Assim, a variação vertical dos dados pode haver erros devidos aos posicionamento e erros devido à

operação do instrumento, além dos erros do próprio instrumento. Mas estes últimos costumam ser bem menores do que os primeiros.

Durante a campanha, os dados gerados em cada perfil ficam armazenados na memória interna do CTD. Posteriormente, usando um software proprietário da JFE os dados são trasferidos para um computador. Cada perfil (cada ligada e desligada do CTD) gera um arquivo. Os arquivos de dados brutos levam a extensão '.raw'. Usando o mesmo software, os dados brutos são convertidos para format o ASCII e salvos com extensão '.csv'. O nome é um padrão automatizado que vai a data e hora da coleta, o número de série do instrumento e algo mais. Estes dados chegaram para mim em uma pasta em que existem outros arquivos também, com mesmo nome mas extensão '.png'.

A primeira parte do processamento então será gerar uma lista com o nome dos arquivos de CTD para podermos carregá-los, e para isso precisaremos do pacote 'OS' que é um acrônimo para 'operation system' é permite trabalhar com arquivos no sistema de diretórios do sistema. E, também já vamos carregar os pacotes 'padrões' numpy e matplot lib…

Obs: não reparem nos números das células do código, entre [ ] do lado esquerdo! Não segui a ordem quando copiei para cá!

[2]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
```

Pacotes carregados, agora eu tenho que indicar o diretório onde estão os arquivos. Lembre-se que no Windows 10, para funcionar tem que colocar um 'r' antes e uma segunda '\' no final! Porquê? 42…! Em seguida usamos o módulo 'listdir' do pacote 'os' para nos fornecer uma lista de todo o conteúdo da pasta, arquivos e diretórios. Para separar o joio do trigo, temos que verificar cada elemento da lista 'mydir', e aqueles que nos agradam, ou seja, os que tem extensão '.csv', nos guardamos em outra lista 'ctd_file'. Como vamos separar um sub-conjunto de um conjunto maior, temos que criar uma variável que irá conter esse sub-conjunto, antes do looping.

No looping, para cada elemento 'file' da lista 'mydir' eu vou testar se a extensão é igual à '.csv' usando o método 'endswith'. Na verdade não estamos testando a extensão, mas os últimos 4 caracteres de 'file' que é um objeto 'string'. Poderia ser só os últimos 3, ou até mesmo só o último. Se o 'if'

resultar no operador booleano ‘True’, é executado o que está dentro do ‘if’. Se o booleano for ‘False’, não. E ao fim, colocamos um ‘print(ctd_file)’ para verificarmos se funcionou.

[11]:
```
path = r'c:\GUTO\Academia\Aulas_AnalisedeDados\CTD_temporal\csv\\'

mydir = os.listdir(path)

ctd_file = []
for file in mydir:
    if file.endswith('.csv'):
        ctd_file.append(file)

print(ctd_file)
```

```
[]
```

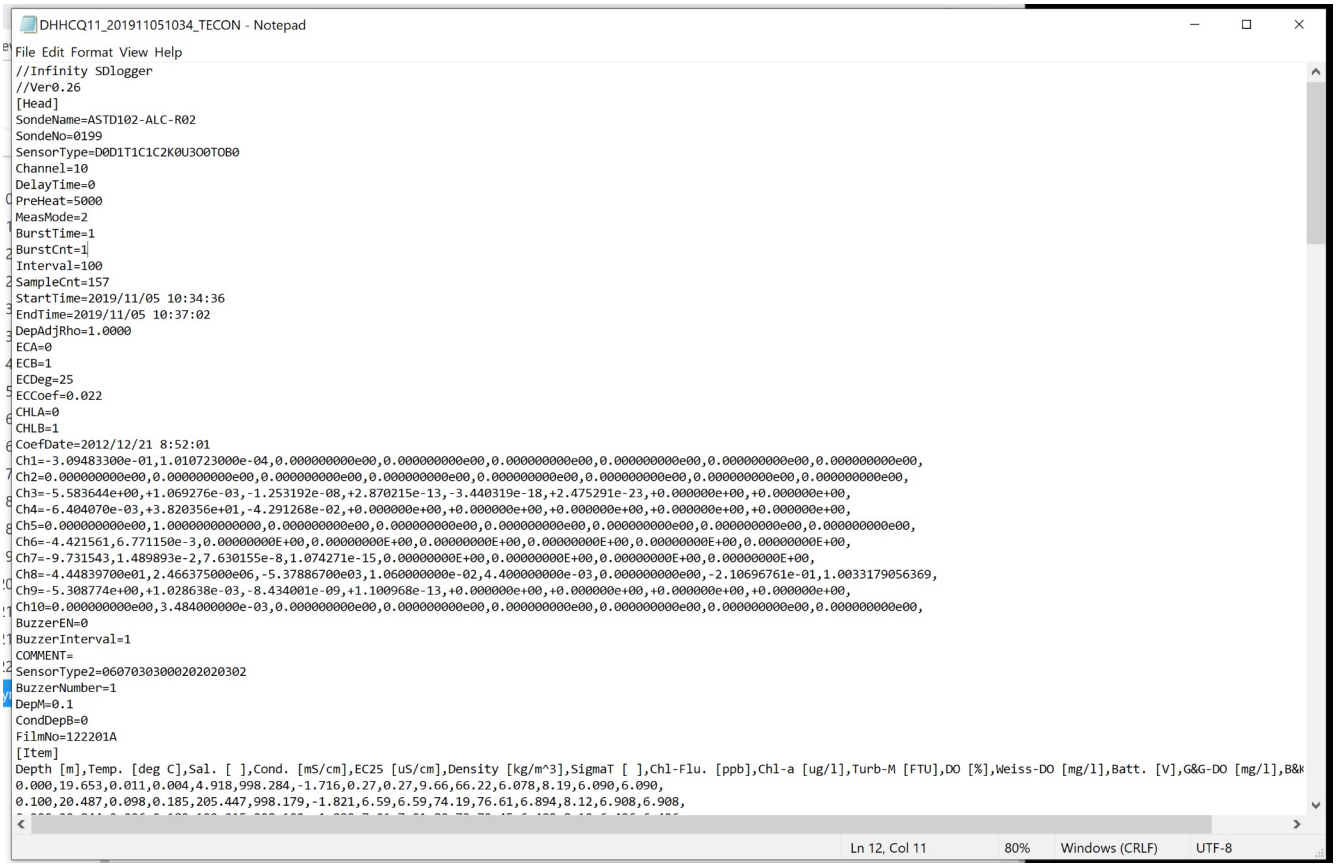
É sempre bom ter uma expectativa do resultado. O resultado esperado seria uma lista com os nomes dos arquivos de CTD que estão na pasta. Mas o resultado foi uma lista vazia. Se der um ‘print(mydir)’, veremos a lista do conteúdo da pasta, e os arquivos estão de fato lá! Ou seja, os nomes dos arquivos estão em ‘mydir’, mas ‘ctd_file’está vazio… o problema está entre a linha de código depois de ‘mydir = … ‘ e o ‘print()’. Como não há erros de sintaxe, que apareceriam explicitamente após a execução do código, o erro está no ‘operador’! Eu (ou tu!). Se tivesse prestado atenção, teria percebido que o ‘c’ do ‘csv’ em ‘mydir’ é maiúsculo. Corrigindo…

[13]:
```
path = r'c:\GUTO\Academia\Aulas_AnalisedeDados\CTD_temporal\csv\\'

mydir = os.listdir(path)

ctd_file = []
for file in mydir:
    if file.endswith('.Csv'):
        ctd_file.append(file)

print(ctd_file)
```

```
['DHHCQ11_201911051034_TECON.Csv', 'DHHCQ11_201911051112_TECON.Csv', 'DHHCQ11_201911051206_TECO
N.Csv', 'DHHCQ11_201911051242_TECON.Csv', 'DHHCQ11_201911051318_TECON.Csv', 'DHHCQ11_2019110513
58_TECON.Csv', 'DHHCQ11_201911051433_TECON.Csv', 'DHHCQ11_201911051510_TECON.Csv', 'DHHCQ11_201
911051607_TECON.Csv', 'DHHCQ11_201911051649_TECON.Csv', 'DHHCQ11_201911051732_TECON.Csv', 'DHHC
Q11_201911051815_TECON.Csv', 'DHHCQ11_201911051854_TECON.Csv', 'DHHCQ11_201911051950_TECON.Cs
v', 'DHHCQ11_201911052032_TECON.Csv', 'DHHCQ11_201911052110_TECON.Csv', 'DHHCQ11_201911052148_T
ECON.Csv', 'DHHCQ11_201911052221_TECON.Csv']
```

Detalhes… mas que travam o andar da análise. Agora que sabemos que funcionou, podemos apagar o ‘print()’ e ir adiante, que é carregar o conteúdo dos arquivos. Inicialmente fazemos uma inspeção em um dos arquivos abrindo no ‘bloco de notas’ ou outro editor de texto qualquer. O que

vemos (abaixo) são várias linhas de cabeçalho (44), e os dados arranjados em colunas da linha 45 em diante.



Sabendo que os dados numéricos são a partir da linha 45 e delimitados por vírgula, uma opção é usar o 'np.loadtxt'. Com o argumento 'skiprows' podemos pular as linhas de cabeçalho, e usando o argumento 'delimiter' informamos que o delimitador é a vírgula. Vamos fazer com um arquivo, e se funcionar, depois fazemos com todos.

[23]:
```
arquivo = ctd_file[0]

f = np.loadtxt(path + arquivo, skiprows=44, delimiter=',')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-6bd1cec54030> in <module>
      1 arquivo = ctd_file[0]
      2
----> 3 f = np.loadtxt(path + arquivo, skiprows=44, delimiter=',')
```

Não funcionou. Se mudar o ‘skiprows’ para 43, vai dar o mesmo erro mas no final na mensagem de erro vai dizer “could not convert string to float: ‘Depth [m]”. E usando o ‘skiprows’ com 44, a mensagem de erro é “could not convert string to float: ”. Sutilezas à parte, há algo estranho na linha 45, a primeira linha de dados numéricos. E o que está estranho é que ela termina com uma vírgula, e assim, o módulo procura algo entre a vírgula e o comando invisível /n que indica quebra de linha, não acha e considera um espaço vazio e interpreta como ‘string’. Para saber disto eu fiz um teste. Peguei um dos arquivos, deletei todas as linhas de dados menos umas 5, das quais deletei a vírgula manualmente, e salvei com outro nome. Este arquivo modificado foi carregado. Ou seja, é a maldita vírgula…

Eu poderia usar o pacote ‘pandas’ que é mais flexível para carregar arquivos. Mas eu prefiro ficar no numpy! Uma alternativa é usar o mesmo procedimento que usamos para carregar os dados de maré do IBGE, com o ‘with’ e o ‘readlines()’. Mesmo que mais prolixo, esse método seria preferível de qualquer jeito… pois temos que obter também a data e hora do perfil que estão no cabeçalho. Ou seja, mesmo que tivesse funcionado usar o ‘np.loadtxt()’, eu teria que voltar atrás para resgatar a data e hora do cabeçalho com o ‘readlines()’.

[5]:
```
with open(path + ctd_file[0]) as file:

    lines = file.readlines()

    c = 0
    for line in lines:
        c += 1

print(len(lines))
print(c)
```

```
201
201
```

No código acima, com (with!) o arquivo aberto como (as!) ‘file’, ‘lines’ acumulará o conteúdo de ‘file’ linha por linha (método ‘readlines()’). E para testar, fazemos um looping para contar quantos objetos (linhas!) foram acumulados na lista ‘lines’, o que é mostrado pelos ‘print()’s. Assim, sabemos que podemos pegar e trabalhar cada linha individualmente, achando a data, hora e os dados. Mas (nada como a experiência…), como usaremos um looping, a linha [lines = file.readlines()] é inútil, como veremos.

Lendo o conteúdo do arquivo linha a linha, podemos primeiramente achar onde está o 'StartTime' que consta no cabeçalho do arquivo com a data e hora que o CTD iniciou o registro (e esperando que o relógio do CTD tenha sido ajustando antes da campanha!). Como o cabeçalho são todos iguais, eu poderia pegar pelo número da linha! Opções diferentes.

[11]:
```
with open(path + ctd_file[0]) as file:

    for line in file:

        if line[0:10] == 'StartTime=':

            date_time = line

            break

print(date_time)
```

```
StartTime=2019/11/05 10:34:36
```

Lembrando que em um fatiamento como 'line[0:10]' o último valor não é incluso (o 10), ou seja, se os caracteres 0, 1, …, até 9 de 'line' forem idênticos à 'StartTime=', o código executa o que está dentro do 'if', que basicamente é guardar o conteúdo de 'line' em 'date_time'. O 'break' é para interromper aí, e depois vemos o que deu.

A campanha é de 12 horas, começou de manha e terminou de tarde. Então pegar as horas e minutos já resolveria. Mas já que estamos aqui, façamos o trabalho completo. Nesse estágio do processamento, eu ainda não sei exatamente o como vou usar o tempo depois. Então vou guardá-lo como obeto 'datetime'. Para isso antes temos que carregar o pacote 'datetime' lá onde carregamos os outros pacotes, incluindo 'import datetime', e executando a célula. E aqui, fazemos

[18]:
```
with open(path + ctd_file[0]) as file:

    for line in file:

        if line[0:10] == 'StartTime=':

            date_time = line

            year = int(date_time[10:14])
            month = int(date_time[15:17])
            day = int(date_time[18:20])
            hour = int(date_time[21:23])
            minute = int(date_time[24:26])

            ctd_time = datetime.datetime(year, month, day, hour, minute)

            break

print(date_time, type(date_time))
print(ctd_time, type(ctd_time))
```

```
StartTime=2019/11/05 10:34:36
 <class 'str'>
2019-11-05 10:34:00 <class 'datetime.datetime'>
```

O conteúdo de ‘date_time’ é muito parecido com o conteúdo de ‘ctd_time’, mas são coisas muiiiiito diferentes para o python. O primeiro é um objeto string (str), e só isso… o segundo é um objeto datetime que é uma referência de tempo numérica que o Python entende que é o tempo em calendário gregoriano. É aquela complexa estória do tempo que vimos no tutorial das marés.

Agora, com o tempo resolvido, vamos pegar os dados. Pra ter certeza que vamos pegar a partir da primeira linha de dados, testamos mudando o valor do teste da linha de código 21. Como está ficando grande, eu ativei a função de mostrar o número de linhas de cada célula. Aliás, para saber o que é cada coluna de dados, podemos rodar esta célula com o teste ‘c == 44’ e irá mostrar a linha de cabeçalho. Marcamos e copiamos ela, criamos uma célula acima desta, mudamos ela para ‘markdown’, e colamos o cabeçalho para servir de cola!

[20]:

```
with open(path + ctd_file[0]) as file:

    c = 0

    for line in file:

        if line[0:10] == 'StartTime=':

            date_time = line

            year = int(date_time[10:14])
            month = int(date_time[15:17])
            day = int(date_time[18:20])
            hour = int(date_time[21:23])
            minute = int(date_time[24:26])

            ctd_time = datetime.datetime(year, month, day, hour, minute)

        c += 1

        if c == 45:

            print(line)
            break

```

```
0.000,19.653,0.011,0.004,4.918,998.284,-1.716,0.27,0.27,9.66,66.22,6.078,8.19,6.090,6.09
0,
```

A cola…

Depth [m],Temp. [deg C],Sal. [ ],Cond. [mS/cm],EC25 [uS/cm],Density [kg/m^3],SigmaT [ ],Chl-Flu. [ppb],Chl-a [ug/l],Turb-M [FTU],DO [%],Weiss-DO [mg/l],Batt. [V],G&G-DO [mg/l],B&K-DO [mg/l],

Destes dados, vamos pegar o que realmente será útil:

- Depth, que é o nível de medição

- Temp. - temperatura da água

- Sal. - salinidade. O CTD mede condutividade, mas este já converte para salinidade (PSS-78)

- Chl-a – clorofila – a. É quase igual à outra clorofila, que é de fluorescência. (Não sei pra que serve!)

- Turb-M – turbidez ótica

- DO [%] - saturação de oxigênico dissolvido

- DO [mg/l] – concentração de oxigênio dissolvido

Line é um objeto string ‘str’, que contém números separados por vírgulas. Deste modo, podemos usar o método ‘split()’ usando o separador ‘,’, o que resultará em uma lista de strings, que então podemos pegar as que nos interessa e converter para números (float). Reparem que agora aparece os caracteres que antes eram invisíveis ‘\n’.

[31]:
```
1  with open(path + ctd_file[0]) as file:
2
3      c = 0
4
5      for line in file:
6
7          if line[0:10] == 'StartTime=':
8
9              date_time = line
10
11             year = int(date_time[10:14])
12             month = int(date_time[15:17])
13             day = int(date_time[18:20])
14             hour = int(date_time[21:23])
15             minute = int(date_time[24:26])
16
17             ctd_time = datetime.datetime(year, month, day, hour, minute)
18
19         c += 1
20
21         if c == 45:
22
23             line_break = line.split(',')
24             break
25
26 print(line_break)
```

```
['0.000', '19.653', '0.011', '0.004', '4.918', '998.284', '-1.716', '0.27', '0.27', '9.6
6', '66.22', '6.078', '8.19', '6.090', '6.090', '\n']
```

Como vamos pegar valores, temos que ter variáveis para guardá-los, então criamos as variáveis vazias antes do looping (linhas 4 a 10), e ao acumular já fazemos a transformação de string para float (linhas 31 a 37).

[36]:

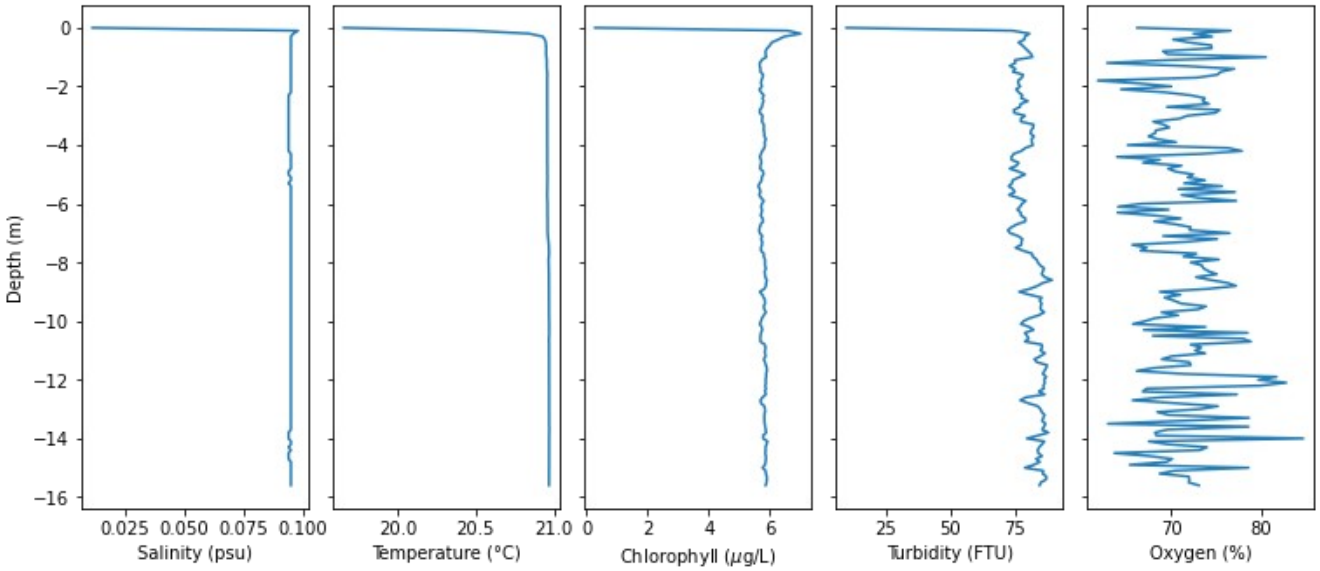
```
with open(path + ctd_file[0]) as file:
    c = 0

    depth = []
    temperature = []
    salinity = []
    chlorophyll = []
    turbidity = []
    oxygen_sat = []
    oxygen = []

    for line in file:

        if line[0:10] == 'StartTime=':

            date_time = line

            year = int(date_time[10:14])
            month = int(date_time[15:17])
            day = int(date_time[18:20])
            hour = int(date_time[21:23])
            minute = int(date_time[24:26])

            ctd_time = datetime.datetime(year, month, day, hour, minute)

        c += 1
        if c >= 45:

            line_break = line.split(',')

            depth.append(float(line_break[0]))
            temperature.append(float(line_break[1]))
            salinity.append(float(line_break[2]))
            chlorophyll.append(float(line_break[8]))
            turbidity.append(float(line_break[9]))
            oxygen_sat.append(float(line_break[10]))
            oxygen.append(float(line_break[11]))

```

Agora temos o tempo do perfil e os dados do perfil, para um dos arquivos. Temos que repetir este procedimento para os outros arquivos. Antes disso, vamos dar uma olhada nestes dados. Sem explicar muito, este é um jeito de fazer isso!

[58]:

```
fig = plt.figure(figsize=(10,5))

# parametros para as dimensões de cada painel
px = .1
py = .1
dx = .18
dy = .8
intervalo = .02

# cria os paineis
ax1 = fig.add_axes([px, py, dx, dy])
ax2 = fig.add_axes([px+(dx+intervalo), py, dx, dy])
ax3 = fig.add_axes([px+(dx+intervalo)*2, py, dx, dy])
ax4 = fig.add_axes([px+(dx+intervalo)*3, py, dx, dy])
ax5 = fig.add_axes([px+(dx+intervalo)*4, py, dx, dy])

depth_g = -np.array(depth) # faz a profundidade negativa

# faz os plots
ax1.plot(salinity, depth_g)
ax2.plot(temperature, depth_g)
ax3.plot(chlorophyll, depth_g)
ax4.plot(turbidity, depth_g)
ax5.plot(oxygen_sat, depth_g)

axes = [ax1, ax2, ax3, ax4, ax5]
labs = ['Salinity (psu)', 'Temperature (°C)',
        'Chlorophyll ($\mu$g/L)','Turbidity (FTU)', 'Oxygen (%)']
for i in range(len(axes)):
    axes[i].set_xlabel(labs[i])
    if i > 0:
        axes[i].set_yticklabels('')

ax1.set_ylabel('Depth (m)')
plt.show()
```

E o resultado será



Para interpretar estes perfis temos que conhecer um pouco do sistema. A parte da desembocadura da lagoa varia bastante. As vezes só tem água doce, as vezes só salgada, as vezes estratificada. Aqui a salinidade está praticamente zero, o que é possível. Os demais parâmetros parecem razoáveis.

Antes de fazermos o gráfico de distribuição da salinidade, todos os perfis devem ser inspecionados individualmente para avaliação de consistência. As vezes o CTD pode dar algum problema e gerar dados espúrios muito óbvios. Por exemplo, nestes dados a primeira medição na superfície está claramente destoando das demais, e deverá ser removida. O que é usual… sempre eliminamos o topo dos perfil de dados de CTD em função do ambiente que estamos trabalhando. Em estuários com condições calmas podemos pegar a partir de 0,5 m com segurança, e neste caso que a coluna de água tem 16 m, isso não fará muita diferença.

Para repetir o processo que fizemos até aqui para todos os arquivos devemos converter os respectivos códigos em funções. Uma para pegar os dados do arquivo, outra para gerar os gráficos. Para fazer isso, usamos a palavra chave ‘def’ para definir a função, que deve ter ao final a palavra-chave ‘return’ informando o que a função devolverá para o código. Há algumas mudanças sutis entre o código anterior e as funções abaixo. Então, preste muita atenção aos detalhes e às mensagens de erro. Todo detalhe faz diferença!

Função para abrir um arquivo de CTD e retornar a data e hora como objeto datetime e as variáveis profundidade, temperatura, salinidade, clorofila, turbidez e oxigênio dissolvido (saturação e concentração) como listas de valores 'float'. É basicamente a mesma coisa, com 1) tabulação à direita, 2) acréscimo da linha 1 que define o nome da função e os parâmetros de entrada, e a linha 40 que define o que retorna ao código.

[87]:

```
1  def carrega_ctd(path, ctd_file):
2      with open(path + ctd_file) as file:
3          c = 0
4
5          depth = []
6          temperature = []
7          salinity = []
8          chlorophyll = []
9          turbidity = []
10         oxygen_sat = []
11         oxygen = []
12
13         for line in file:
14
15             if line[0:10] == 'StartTime=':
16
17                 date_time = line
18
19                 year = int(date_time[10:14])
20                 month = int(date_time[15:17])
21                 day = int(date_time[18:20])
22                 hour = int(date_time[21:23])
23                 minute = int(date_time[24:26])
24
25                 ctd_time = datetime.datetime(year, month, day, hour, minute)
26
27             c += 1
28             if c >= 45:
29
30                 line_break = line.split(',')
31
32                 depth.append(float(line_break[0]))
33                 temperature.append(float(line_break[1]))
34                 salinity.append(float(line_break[2]))
35                 chlorophyll.append(float(line_break[8]))
36                 turbidity.append(float(line_break[9]))
37                 oxygen_sat.append(float(line_break[10]))
38                 oxygen.append(float(line_break[11]))
39
40     return ctd_time, depth, salinity, temperature, chlorophyll, turbidity, oxygen_sat, oxygen
41
```

Função para gerar os gráficos dos perfis verticais das variáveis do CTD. Retorna um objeto 'fig', que contém a figura. Similarmente à anterior, muda pouca coisa, mas a medida que vão surgindo demandas, mais coisas podem entrar aqui. Detalhe para a linha 37 que o comando 'plt.show()' foi

desativado (virou comentário), e o acréscimo das linha 40 a 42 para incluir no gráfico a indicação do perfil (arquivo) que está sendo mostrado. Isso surgiu depois durante a inspeção dos dados.

[33]:

```
1  def grafico_perfis(depth, salinity, temperature, chlorophyll, turbidity, oxygen, n_perf):
2
3      fig = plt.figure(figsize=(10,5))
4
5      # parametros para as dimensões de cada painel
6      px = .1
7      py = .1
8      dx = .18
9      dy = .8
10     intervalo = .02
11
12     # cria os paineis
13     ax1 = fig.add_axes([px, py, dx, dy])
14     ax2 = fig.add_axes([px+(dx+intervalo), py, dx, dy])
15     ax3 = fig.add_axes([px+(dx+intervalo)*2, py, dx, dy])
16     ax4 = fig.add_axes([px+(dx+intervalo)*3, py, dx, dy])
17     ax5 = fig.add_axes([px+(dx+intervalo)*4, py, dx, dy])
18
19     depth_g = -np.array(depth) # faz a profundidade negativa
20
21     # faz os plots
22     ax1.plot(salinity, depth_g)
23     ax2.plot(temperature, depth_g)
24     ax3.plot(chlorophyll, depth_g)
25     ax4.plot(turbidity, depth_g)
26     ax5.plot(oxygen, depth_g)
27
28     axes = [ax1, ax2, ax3, ax4, ax5]
29     labs = ['Salinity (psu)', 'Temperature (°C)',
30             'Chlorophyll ($\mu$g/L)','Turbidity (FTU)', 'Oxygen (%)']
31     for i in range(len(axes)):
32         axes[i].set_xlabel(labs[i])
33         if i > 0:
34             axes[i].set_yticklabels('')
35
36     ax1.set_ylabel('Depth (m)')
37 #     plt.show()
38
39     # para indicar o perfil mostrado!
40     ax6 = fig.add_axes([.1,.9,.1, .1])
41     ax6.axis('off')
42     ax6.text(.5, .5, 'Perfil ' + str(n_perf), size=14, weight='bold')
43
44     return fig
```

Com as funções prontas, eu tenho que repetir o processos tantas vezes quanto for o número de arquivos de CTD. Não me interessa saber o número, pois é o tamanho da lista (len()) ‘ctd_file’. Fazemos então um looping tal como mostrado abaixo. Em vez de dar o nome explícito das variáveis (depth, salinity, etc.) como usamos acima, para sermos óbvios, aqui vamos ser mais abstratos.

Observação. O código até aqui eu fiz de maneira rápida… digo, fui fazendo com a minha experiência com o Python. Embora hajam muitas linhas de código, são coisas que costumo fazer frequentemente, e saiu 'fluido'. Digamos, 15 minutos se não estivesse escrevendo este tutorial. O código abaixo (linhas 5 a 11) eu levei umas boas (4 a 6, juntando tudo) horas para deixá-lo Ok, pois são coisas que não faço rotineiramente. Foram diveras tentativas e erro e dezenas de consultas à fórums para chegar nesta solução. Vai abaixo um desenho desses que circulam por aí que expressa bem o que é programar! Muita falha e desapontamento e muita persistência. Mas aqui está somente o que funciona, e daí parece fácil.

Mas o que há de tão difícil nisso? Por partes (Ripper, 1888). Estou fazendo um gráfico através de uma função, que cria um objeto (fig) que será apresentado aqui. Experimentem excluir (ou comentar) as linhas 8 a 11 para ver o que acontece. Talvez esse resultado seja satisfatório para vocês, mas não para mim. O meu objetivo é dar uma olhada em cada perfil e não tumultuar meu notebook. Imaginem se fossem 50 ou 100 perfis! Então eu tenho que atualizar o gráfico para cada perfil, sem criar um novo gráfico a cada loop. Bem, na verdade cria-se um novo gráfico mas elimina-se o anterior.

No Google uma das tentativas eu usei "matplotlib update plot in loop jupyter" como palavras de busca. Verifiquei vários resultados. Fazemos isso copiando o código para um notebook e testando para ver o que acontece, se funciona, se há dependências de outros pacotes, etc… Neste resultado

https://stackoverflow.com/questions/53903396/how-to-update-matpplotlib-lib-plots-in-a-python-loop-with-the-plot-sitting-in-th

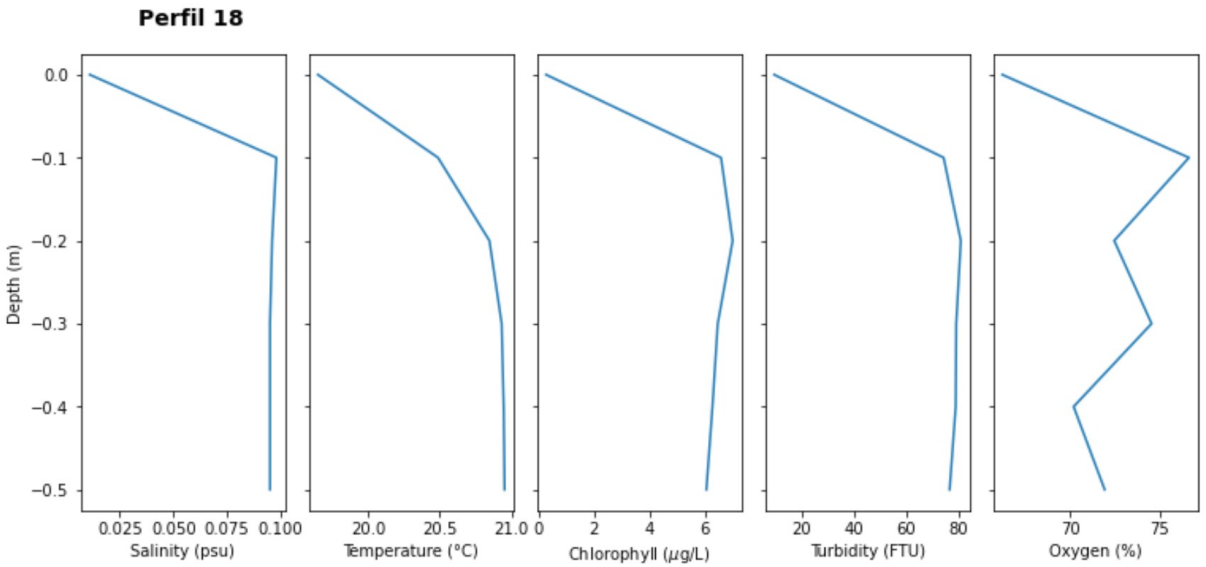
encontrei algo que poderia ajudar, na única resposta. Copiei o código e testei mudando o looping the 10 para 3. O resultado foi interessante, mas há ainda um efeito colateral, que ao final do processo é gerado uma cópia da figura, e ficamos com duas. Porque é gerado uma segunda figura está além da minha compreensão, mas é bem melhor que uma fieira de figuras. Mas ainda insatisfatório. Além de que neste caso os gráficos vão sendo adicionados, e não substituídos como queremos. Mas é algo para começar.

Percebam que são utilizadas outros pacotes. O 'pylab' que nada mais é o 'numpy' e o 'matplotlib.pyplot' juntos, mas não é recomendado seu uso (por partes!). O módulo 'display' do pacote 'Ipython' que permite gerenciar a visualicação de objetos fig. E o pacote 'time' que permite criar uma pausa no ciclo do loop. Disso, usaremos os pacotes 'time' e o 'Ipyhon' que devem ser importados lá na

célula onde fazemos as importações de pacotes. Para eliminar a figura duplicada, após investigar em outros resultados de busca encontrei que basta usar o 'plt.close()'. Então, testando e errando muitas (muitas mesmo!), cheguei ao código abaixo. Simples!

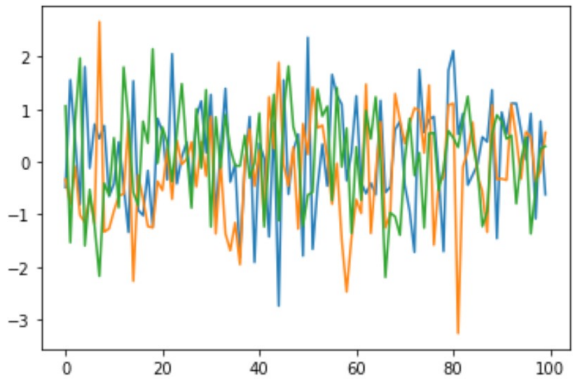
[34]:

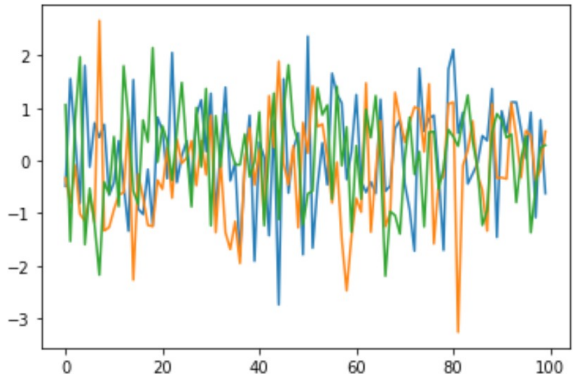
```
for i in range(len(ctd_file)):
    a, b, c, d, e, f, g, h =  carrega_ctd(path, ctd_file[i])

    fig = grafico_perfis(b, c, d, e, f, g, i)


    display.clear_output(wait=True)
    display.display(plt.gcf())
    time.sleep(.5)
    plt.close()

```



{ código copiado de https://stackoverflow.com/questions/53903396/how-to-update-matplotlib-lib-plots-in-a-python-loop-with-the-plot-sitting-in-th }

[12]:
```
%matplotlib inline
import time
import pylab as pl
from IPython import display
for i in range(3):
      pl.plot(pl.randn(100))
      display.clear_output(wait=True)
      display.display(pl.gcf())
      time.sleep(1.0)
```





Após a inspeção, constatamos que os dados do último arquivo são lixo*, e os demais estão bem, exceptuando-se o topo que será retirado depois. Para juntar todos os dados de todos os arquivos poderíamos fazer no mesmo loop que fizemos a inspeção dos dados. Mas, por questões didáticas vamos fazer separado. Basicamente repetimos o procedimento de repetir e acumular. E, como vamos acumular, devemos criar as variáveis para fazer isso. No caso, voltamos aos nomes explícitos, já que estamos usando a, b, c, etc., na function que carrega os dados.

Aqui, considerando que o tempo é horas do dia, já vou adiantar e criar um arranjo que conterá o tempo que terá o mesmo tamanho que os demais, que será necessário para realizar a interpolação. E aproveito e embalo para eliminar o topo dos perfis. O CTD foi configurado para registrar dados em intervalos de pressão (0.1 dB) (dB = deciBar ~ m de coluna de água). Então, se pegarmos a partir do 6.o valor estamos eliminando primeiro 0,5 m de coluna de água. Para juntar os dados dos diferentes arquivos, aqui eu uso o 'np.append()', que vai juntando o conteúdo a partir do último elemento. Assim, eu crio arranjos uni-dimensionais (1D) para cada variável e que contém todos os dados. Seria como se fosse uma coluna do Excell.

Um 'detalhe' a mais aqui é que como usaremos o 'np.append' para juntar os conteúdos, o resultado será um objeto tipo arranjo, e não lista. É quase igual, mas há operações com lista que não se aplicam em arranjos e vice-versa. Por exemplo, não dá para usar o '.shape' em listas e nem fazer operações do tipo 'broadcasting'. É confuso, mas em um código a gente vai transformando a mesma coisa ora em arranjo ora em lista por conveniência.

* Errata: na verdade estes dados não são lixo, mas sim os dados que eu usei para testar se a vírgula no final de cada linha fazia ou não diferença no modo de carregar os dados. Para não apagar um monte te vírgulas, eu deletei a maior parte dos dados. Por descuido (vacilo!) eu deixei na mesma pasta! Isso é um típico problema de CO. O acrônimo por extenso é 'grosseiro'...

[59]:

```
time = []
depth = []
temperature = []
salinity = []
chlorophyll = []
turbidity = []
oxygen_sat = []
oxygen = []

for i in range(len(ctd_file) - 1):
    a, b, c, d, e, f, g, h =  carrega_ctd(path, ctd_file[i])

    time_b = a.hour +  a.minute/60
    time_b = np.linspace(time_b, time_b, len(b))

    time_b = time_b[5:-1]
    b = b[5:-1]
    c = c[5:-1]
    d = d[5:-1]
    e = e[5:-1]
    f = f[5:-1]
    g = g[5:-1]
    h = h[5:-1]


    time = np.append(time, time_b)
    depth = np.append(depth, b)
    salinity = np.append(salinity, c)
    temperature = np.append(temperature, d)
    chlorophyll = np.append(chlorophyll, e)
    turbidity = np.append(turbidity, f)
    oxygen_sat = np.append(oxygen_sat, g)
    oxygen = np.append(oxygen, h)

```

Agora estamos chegando nos finalmentes! Para eu poder fazer um gráfico de distribuição, eu preciso que os números da variável de interesse estejam arranjados em uma matriz regular. Se todos os perfis tivessem exatamente o mesmo número de medições verticais, bastaria transformar o arranjo 1D em um 2D usando o 'np.reshape()', e com as matrizes de tempo, profundidade e variável, gerar o gráfico de distribuição. Há diferentes opções para se chegar nisso. Aqui vamos criar um domínio onde os dados serão interpolados em 2D, usando o módulo 'griddata' do pacote 'scipy.interpolate'.

Primeiramente precisamos determinar os domínios das dimensões horizontal (tempo) e vertical (profundidade). Basta usar o 'np.linspace()' com o valor mínimo e máximo do arranjo que contém o tempo e determinar o número de elementos. Fazemos o mesmo para a profundidade. O resultado são dois arranjos 1D. Os domínios 2D são criados usando o 'np.meshgrid', que basicamente replicará o arranjo de tempo em tantas linhas quanto o número de elementos do arranjo de profundidade, e o

arranjo de profundidade em tantas colunas quanto o número de elementos do arranjo de tempo. Simples assim… investigue dando um print neles!

[63]:
```
ti = np.linspace(np.min(time), np.max(time), 20)
zi = np.linspace(np.min(depth), np.max(depth), 20)

tt, zz = np.meshgrid(ti, zi)
```

Adiantando, depois de ver novamente como funciona o griddata (Google!), as coordenadas de cada ponto no nosso domínio deve ser fornecido como uma matriz com duas colunas, a primeira o tempo e a segunda a profundidade. Então temos que juntar os arranjos, ou fazer o contrário de um fatiamento. Há diferentes maneiras de fazer isso… talvez tu aches uma mais elegante. Primeiro eu faço uma fusão dos dois arranjos, depois reformo e depois transponho. Isso feito, posso usar o 'griddata'. Repare que aqui eu não preciso chamar o apelido do pacote (np, plt, etc.) porque eu importei o 'griddata' de forma difernte (from scipy.interpolate import griddata). O resultado é que a variável que contém o resultado será uma matriz com as mesmas dimensões do domínio que eu criei antes.

Há toda uma teoria em relação a interpolação de dados. Não entraremos no mérito. Mas o bom senso sugere que não devemos 'exagerar' na resolução do domínio. Temos 18 perfis temporais, e usamos 20. Nossa resolução vertical é de 0,1 m, e a coluna de água é de 17 m, e usamos 20, mas podemos aumentar para 100 ou mais depois, em função da densidade vertical de observações.

[93]:
```
ti = np.linspace(np.min(time), np.max(time), 20)
zi = np.linspace(np.min(depth), np.max(depth), 20)

tt, zz = np.meshgrid(ti, zi)
```

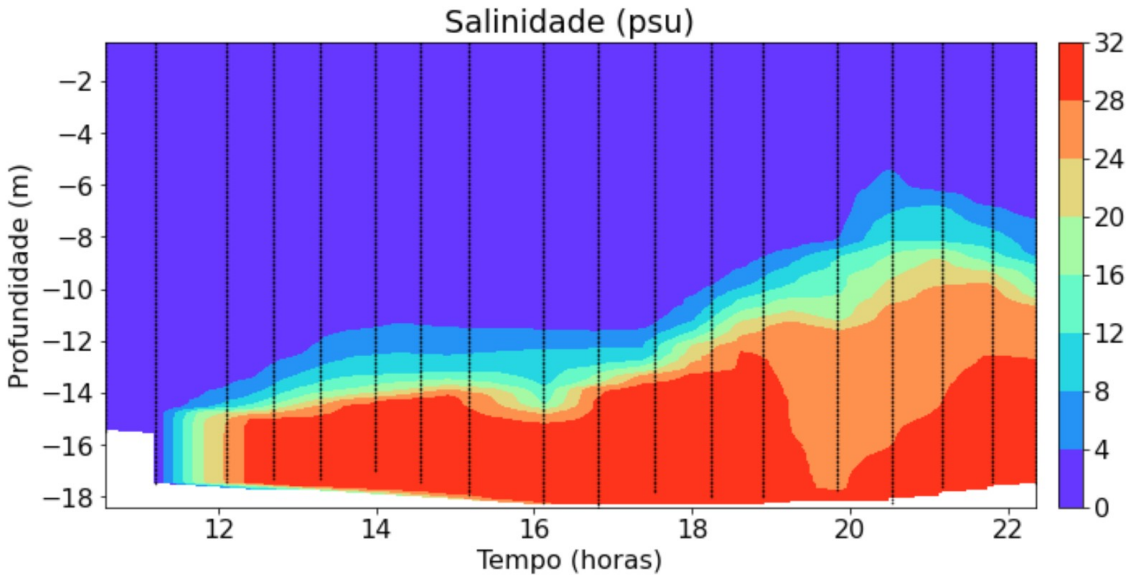
[94]:
```
points = [time, depth]
points = np.reshape(points,(2,-1))
points = np.transpose(points)
```

[95]:
```
salinity_i = griddata(points, salinity, (tt, zz), method='linear')
```

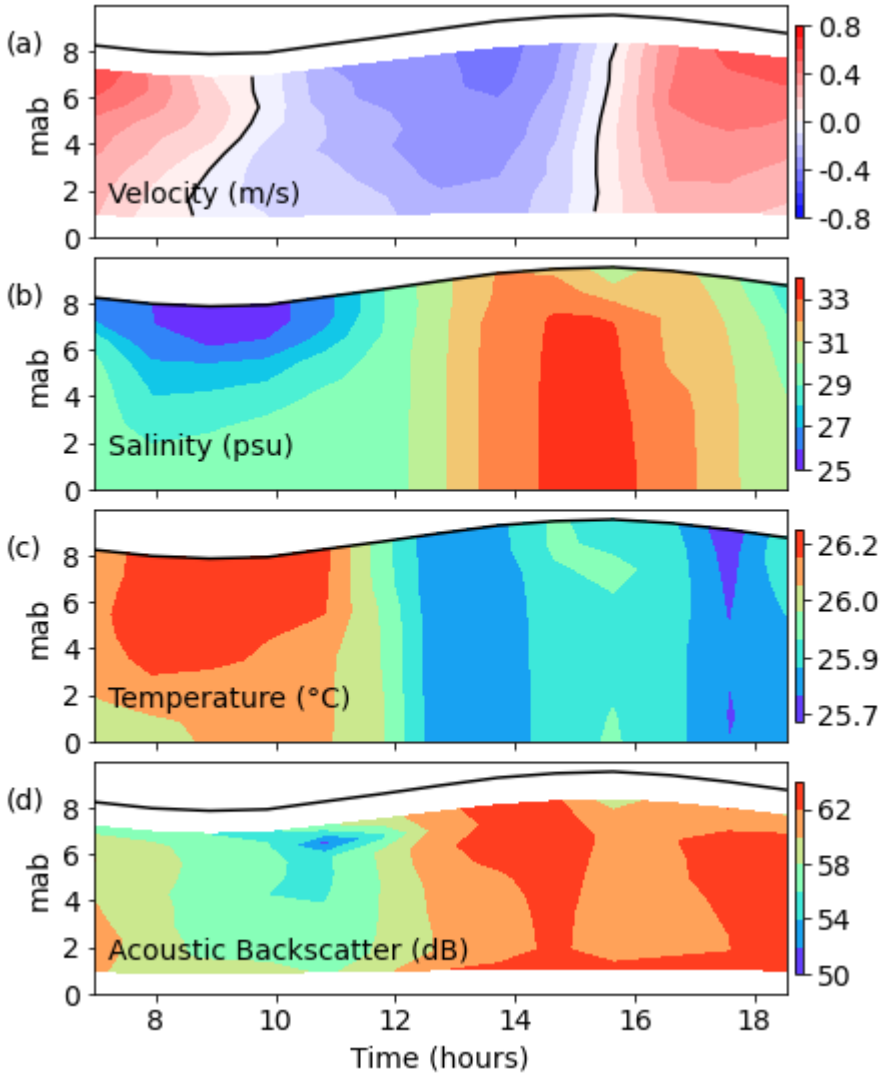
Agora os finalmentes final! O gráfico.

[109]:

```
plt.rcParams.update({'font.size': 16})
fig = plt.figure(figsize=(10,5))

ax1 = fig.add_axes([.1, .1, .8, .8])
ax1cb = fig.add_axes([.92, .1, .02, .8])

ax1.plot(time, -depth, 'k.', markersize=1)

cont = ax1.contourf(ti, -zi, salinity_i, cmap='rainbow')
plt.colorbar(cont, cax=ax1cb)

ax1.set_xlabel('Tempo (horas)')
ax1.set_ylabel('Profundidade (m)')
ax1.set_title('Salinidade (psu)')

plt.show()

```



Este gráfico não está completo. Durante o período de coleta o nível da água oscilou, e isto não foi considerado neste tutorial. Todos os perfis estão referenciados pelo sensor de pressão do CTD que é a superfície da água. Se dados de nível estão disponíveis para este período e local, é possível fazer a correção vertical de cada perfil antes de interpolar. Abaixo está um exemplo de distribuições de dados onde o referenciamento é feito pelo fundo (mab = meters above the bed) e é apresentado a variação do nível da água durante a campanha, o que torna muito mais fácil interpretar os resultados.


(a)
mab
Velocity (m/s)
0.8
0.4
0.0
-0.4
-0.8
(b)
mab
Salinity (psu)
33
31
29
27
25
(c)
mab
Temperature (°C)
26.2
26.0
25.9
25.7
(d)
mab
Acoustic Backscatter (dB)
62
58
54
50
0
2
4
6
8
8
10
12
14
16
18
Time (hours)

O notebook final completo:

Processando dados de CTD - lavantamento temporal

[1]:
```
import os
import time
import datetime
import numpy as np
import matplotlib.pyplot as plt
import IPython.display as display
from scipy.interpolate import griddata

```

[2]:
```
path = r'c:\GUTO\Academia\Aulas_AnalisedeDados\CTD_temporal\csv\\'

mydir = os.listdir(path)

ctd_file = []
for file in mydir:
    if file.endswith('.Csv'):
        ctd_file.append(file)

# print(ctd_file)
```

Depth [m],Temp. [deg C],Sal. [ ],Cond. [mS/cm],EC25 [uS/cm],Density [kg/m^3],SigmaT [ ],Chl-Flu. [ppb],Chl-a [ug/l],Turb-M [FTU],DO [%],Weiss-DO [mg/l],Batt. [V],G&G-DO [mg/l],B&K-DO [mg/l],

[3]:

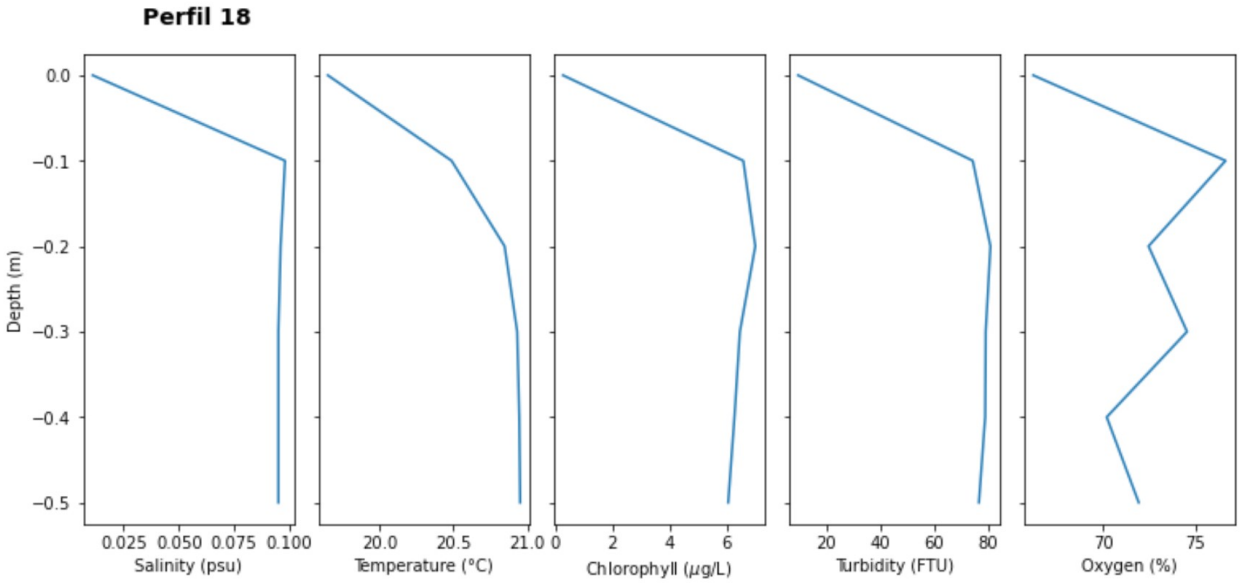
```
def carrega_ctd(path, ctd_file):
    with open(path + ctd_file) as file:
        c = 0

        depth = []
        temperature = []
        salinity = []
        chlorophyll = []
        turbidity = []
        oxygen_sat = []
        oxygen = []

        for line in file:

            if line[0:10] == 'StartTime=':

                date_time = line

                year = int(date_time[10:14])
                month = int(date_time[15:17])
                day = int(date_time[18:20])
                hour = int(date_time[21:23])
                minute = int(date_time[24:26])

                ctd_time = datetime.datetime(year, month, day, hour, minute)

            c += 1
            if c >= 45:

                line_break = line.split(',')

                depth.append(float(line_break[0]))
                temperature.append(float(line_break[1]))
                salinity.append(float(line_break[2]))
                chlorophyll.append(float(line_break[8]))
                turbidity.append(float(line_break[9]))
                oxygen_sat.append(float(line_break[10]))
                oxygen.append(float(line_break[11]))

    return ctd_time, depth, salinity, temperature, chlorophyll, turbidity, oxygen_sat, oxygen

```

[4]:

```
def grafico_perfis(depth, salinity, temperature, chlorophyll, turbidity, oxygen, n_perf):

    fig = plt.figure(figsize=(10,5))

    # parametros para as dimensões de cada painel
    px = .1
    py = .1
    dx = .18
    dy = .8
    intervalo = .02

    # cria os paineis
    ax1 = fig.add_axes([px, py, dx, dy])
    ax2 = fig.add_axes([px+(dx+intervalo), py, dx, dy])
    ax3 = fig.add_axes([px+(dx+intervalo)*2, py, dx, dy])
    ax4 = fig.add_axes([px+(dx+intervalo)*3, py, dx, dy])
    ax5 = fig.add_axes([px+(dx+intervalo)*4, py, dx, dy])

    depth_g = -np.array(depth) # faz a profundidade negativa

    # faz os plots
    ax1.plot(salinity, depth_g)
    ax2.plot(temperature, depth_g)
    ax3.plot(chlorophyll, depth_g)
    ax4.plot(turbidity, depth_g)
    ax5.plot(oxygen, depth_g)

    axes = [ax1, ax2, ax3, ax4, ax5]
    labs = ['Salinity (psu)', 'Temperature (°C)',
            'Chlorophyll ($\mu$g/L)','Turbidity (FTU)', 'Oxygen (%)']
    for i in range(len(axes)):
        axes[i].set_xlabel(labs[i])
        if i > 0:
            axes[i].set_yticklabels('')

    ax1.set_ylabel('Depth (m)')
#     plt.show()

    # para indicar o perfil mostrado!
    ax6 = fig.add_axes([.1,.9,.1, .1])
    ax6.axis('off')
    ax6.text(.5, .5, 'Perfil ' + str(n_perf), size=14, weight='bold')

    return fig
```

[5]:

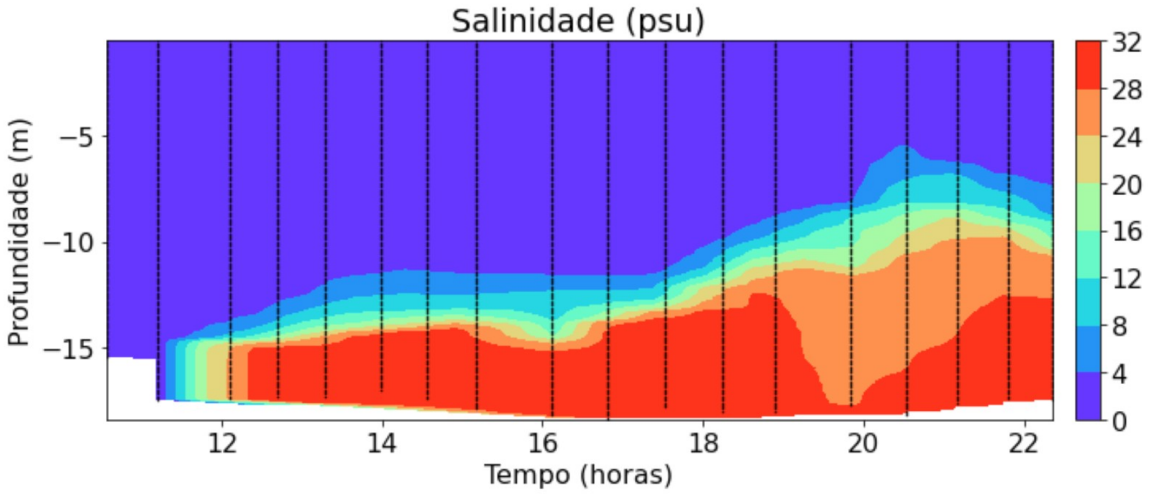
```
for i in range(len(ctd_file)):
    a, b, c, d, e, f, g, h =  carrega_ctd(path, ctd_file[i])

    fig = grafico_perfis(b, c, d, e, f, g, i)


    display.clear_output(wait=True)
    display.display(plt.gcf())
    time.sleep(.5)
    plt.close()

```



[8]:
```
points = [time, depth]
points = np.reshape(points,(2,-1))
points = np.transpose(points)
```

[9]:
```
salinity_i = griddata(points, salinity, (tt, zz), method='linear')
```

[11]:
```
plt.rcParams.update({'font.size': 16})
fig = plt.figure(figsize=(10,4))

ax1 = fig.add_axes([.1, .1, .8, .8])
ax1cb = fig.add_axes([.92, .1, .02, .8])

ax1.plot(time, -depth, 'k.', markersize=1)

cont = ax1.contourf(ti, -zi, salinity_i, cmap='rainbow')
plt.colorbar(cont, cax=ax1cb)

ax1.set_xlabel('Tempo (horas)')
ax1.set_ylabel('Profundidade (m)')
ax1.set_title('Salinidade (psu)')

plt.show()
```



[6]:

```
time = []
depth = []
temperature = []
salinity = []
chlorophyll = []
turbidity = []
oxygen_sat = []
oxygen = []

for i in range(len(ctd_file) - 1):
    a, b, c, d, e, f, g, h =  carrega_ctd(path, ctd_file[i])

    time_b = a.hour +  a.minute/60
    time_b = np.linspace(time_b, time_b, len(b))

    time_b = time_b[5:-1]
    b = b[5:-1]
    c = c[5:-1]
    d = d[5:-1]
    e = e[5:-1]
    f = f[5:-1]
    g = g[5:-1]
    h = h[5:-1]


    time = np.append(time, time_b)
    depth = np.append(depth, b)
    salinity = np.append(salinity, c)
    temperature = np.append(temperature, d)
    chlorophyll = np.append(chlorophyll, e)
    turbidity = np.append(turbidity, f)
    oxygen_sat = np.append(oxygen_sat, g)
    oxygen = np.append(oxygen, h)

```

[7]:

```
ti = np.linspace(np.min(time), np.max(time), 20)
zi = np.linspace(np.min(depth), np.max(depth), 150)

tt, zz = np.meshgrid(ti, zi)

```