The OCE package

Dan E. Kelley

May 20, 2015

Abstract

The `oce` package makes it easy to read, summarize and plot data from a variety of Oceanographic instruments, isolating the researcher from the quirky data formats that are common in this field. It also provides functions for working with basic seawater properties such as the equation of state, and with derived quantities such as the buoyancy frequency. Although simple enough to be used in a teaching context, `oce` is powerful enough for a research setting. These things are illustrated here, in the context of some practical examples. Worked examples are provided, in order to help readers take early steps towards using the `oce` package in their research.

1 Introduction

Oceanographers must deal with measurements made by a wide variety of instruments, a task that is complicated by the delight instrument manufacturers seem to take in inventing new data formats. The manufacturers often provide software for scanning the data files and producing some standard plots, but this software is of limited use to researchers who work with several instrument types at the same time, and who need to carry the analysis beyond the first steps, e.g. moving beyond engineering plots to scientific plots and to statistical analysis.

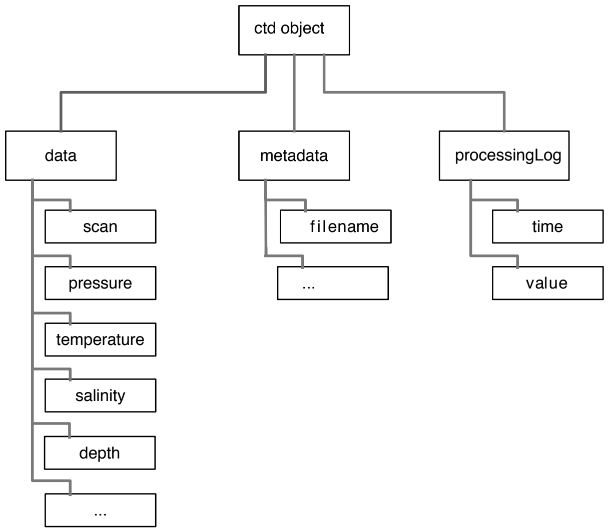


Figure 1: Sketch of the contents of a `ctd` object. All `oce` objects contain slots named `data`, `metadata`, and `processingLog`, with the contents depending on the type of data.

The need to scan diverse data files was one motivation for the creation of `oce`, but an equal goal was to make it easy to work with the data once they are in the system. This was accomplished partly by the provision of functions to work with the data, and partly by developing a uniform object design that lets users reach inside without guesswork.

At the core of the `oce` design process is a policy of adding features according to the priorities of practical research. As a result, `oce` is a fairly comfortable tool today, and it should remain so as it grows.

2 Object design

As illustrated in Figure 1, each `oce` object contains three slots: (a) `data`, a list or a data frame containing the actual data (e.g., for a CTD object, this will contain pressure, temperature, etc.), (b) `metadata`, a list containing data such things as file headers, the location of a CTD cast, etc., and (c) `processingLog`, a list that documents how the file was created (often by a `read` or `as` method) and how it was changed thereafter (e.g. by decimating a CTD cast).

The uniformity of the various `oce` objects makes it easy to build skill in examining and modifying objects. The package provides for simple and general processing of `oce` objects. For example, if `x` is an `oce` object, then the function call `plot(x)` will produce one type of plot if `x` contains hydrographic data from a CTD, and quite another type of plot if it contains velocity data from an ADCP. This applies for a variety of actions that can be undertaken on the `oce` objects, and it makes for easy programming, e.g. for over a dozen types of oceanographic data files, the following general code produces a summary plot:

```
library(oce)
d <- read.oce(filename)
plot(d)
```

where `filename` is the name of a file containing the data.

Some notes on oce function names.

1. The function used above to read a dataset ends in `.oce`, which is a signal that the returned object is of class `oce`. Depending on the file contents, `d` will also have an additional class, e.g. if `filename` contains CTD data, then the object would have two classes: `oce` and `ctd`, and the second of these is used in plotting, meaning that R will call a function named `read.ctd` for the plot.

2. Generally, `oce` functions employ a "camel case" naming convention, in which a function that is described by several words is named by stringing the words together, capitalizing the second and subsequent words. For example, `ctdAddColumn` takes a `ctd` object, and adds a column.

3. Function names begin with "`oce.`" in cases where the most natural function name would otherwise be in conflict with a function in the base R system or a package commonly used by Oceanographers. For example, `oce.approx` is used for a function that is analogous to `approx` in the `stats` package. The reason for not naming the function as `oceApprox` is a desire to provide a clear hint about the similarity of the functions. The reason for avoiding a naming conflict (by calling it `approx`) is to avoid the necessity of writing `oce::approx`, a notation that can be confusing to users.

3 Calculations of seawater properties

The `oce` package provides many functions for dealing with seawater properties. Perhaps the most used is `swRho(S,T,p)`, which computes seawater density $\rho = \rho(\mathit{salinity}, \mathit{temperature}, \mathit{pressure})$, where salinity follows the practical salinity scale, *temperature* is *in-situ* temperature in °C, and *pressure* is seawater pressure, i.e. the excess over atmospheric pressure, in dbar. (This and similar functions starts with the letters `sw` to designate that they relate to seawater properties; a future version of `oce` may provide air properties, with functions names starting with `air.`) The result is a number in the order of $1000\,\mathrm{kg/m^3}$. For many purposes, Oceanographers prefer to use the density anomaly $\sigma = \rho - 1000\,\mathrm{kg/m^3}$, provided with `swSigma(salinity,temperature,pressure)`, or its adiabatic cousin σ_θ, provided with `swSigmaTheta`.

Most of the functions use the UNESCO formulations of seawater properties, but new formulations may be added as they come into use in the literature. A partial list of seawater functions is as follows: `swDynamicHeight` (dynamic height), `swN2` (buoyancy frequency), `swSCTp` (salinity from conductivity, temperature and pressure), `swSTrho` (salinity from temperature and density), `swTSrho` (temperature from salinity and density), `swTFreeze` (freezing temperature), `swAlpha` (thermal expansion coefficient $\alpha = -\rho_0^{-1}\partial\rho/\partial T$), `swBeta` (haline compression coefficient $\beta = \rho_0^{-1} \partial\rho/\partial S$), `swAlphaOverBeta` ($\alpha/\beta$), `swConductivity` (conductivity from S, T and p), `swDepth` (depth from p and latitude), `swLapseRate` (adiabatic lapse rate), `swRho` (density ρ from S, T and p), `swSigma` ($\rho - 1000\,\mathrm{kg/m^3}$), `swSigmaT` ($\sigma$ with p set to zero and temperature unaltered), `swSigmaTheta` (σ with p set to zero and temperature altered adiabatically), `swSoundSpeed` (speed of sound), `swSpecificHeat` (specific heat), `swSpice` (a quantity used in double-diffusive research), `swTheta` (potential temperature θ), and `swViscosity` (viscosity). Details and examples are provided in the documentation of these functions.

Exercise 1. (a) What is the density of a seawater parcel at pressure 100 dbar, with salinity 34 PSU and temperature 10°C? (b) What temperature would the parcel have if raised adiabatically to the surface? (c) What density would it have if raised adiabatically to the surface? (d) What density would it have if lowered about 100m, increasing the pressure to 200dbar? (e) Draw a blank T-S diagram with S from 30 to 40 PSU and T from -2 to 20°C. (Answers are provided at the end of this document.)

4 CTD data

4.1 Example with pre-trimmed data

To get you started with CTD data, `oce` provides a sample data set that has been trimmed to just the downcast portion of the sampling. (See the next section to learn how to do this trimming.). The commands

```
library(oce)
data(ctd)
plot(ctd)
```

produce Figure 2. You may also get a summary of the data with

```
summary(ctd)
```

The object used to hold CTD data stores not just the data, but also the raw header sequence, and whatever other metadata has been discovered about the dataset by

```
names(ctd@metadata)
```

etc.

Of course, you may apply any `R` techniques to the data in `oce` objects, e.g. `hist(ctd@data$temperature)` would produce a histogram of temperature for the `ctd` object. It is always worth checking, though, to see if `oce` has already defined a function that you may be applying, e.g. `plotTS` will produce a lovely temperature-salinity diagram, with isopycnals and proper units on the axes.

The package provides facilities for some common operations with oceanographic data, such as trimming CTD profiles with `ctdTrim()`, but of course you may do that sort of work by acting on the data directly, if necessary. Just make sure you realize that the metadata will not be altered if you do that. Also, it is a good idea to add log entries to any objects that you change, by using the `processingLog()` function. (You can see an example of this in action with `?section`.)

Exercise 2. Plot a profile of σ_θ and N^2, for the data in the pycnocline.

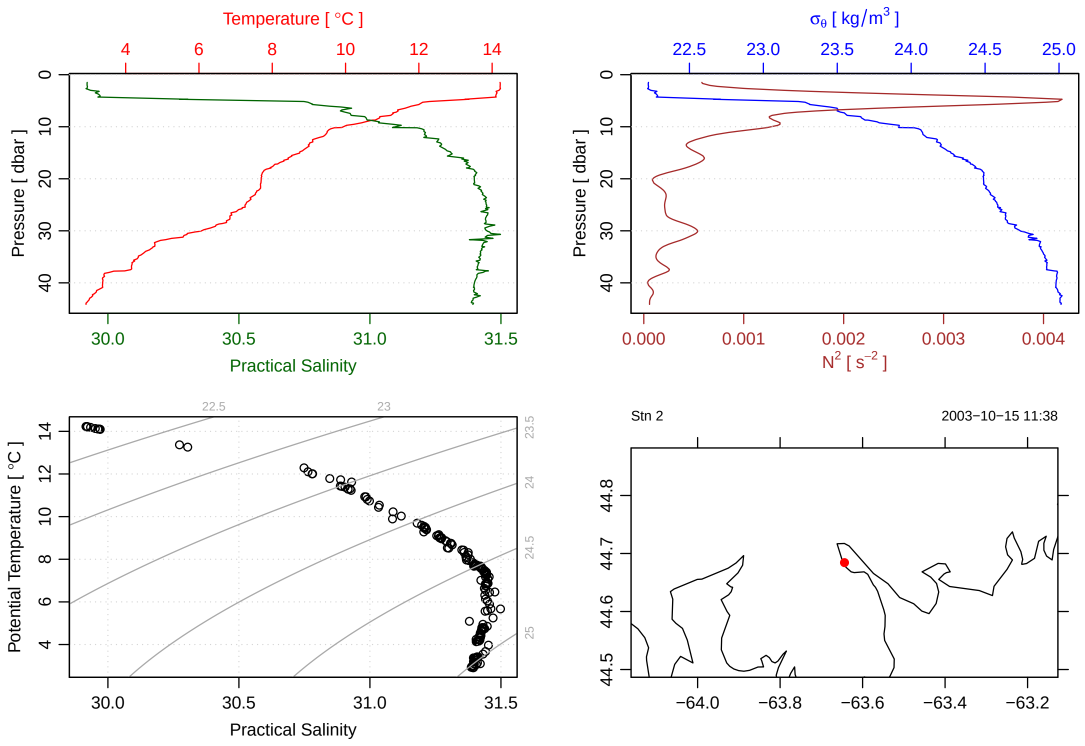


Figure 2: Overview graph of the dataset `ctd`, acquired by the author's class at Dalhousie University.

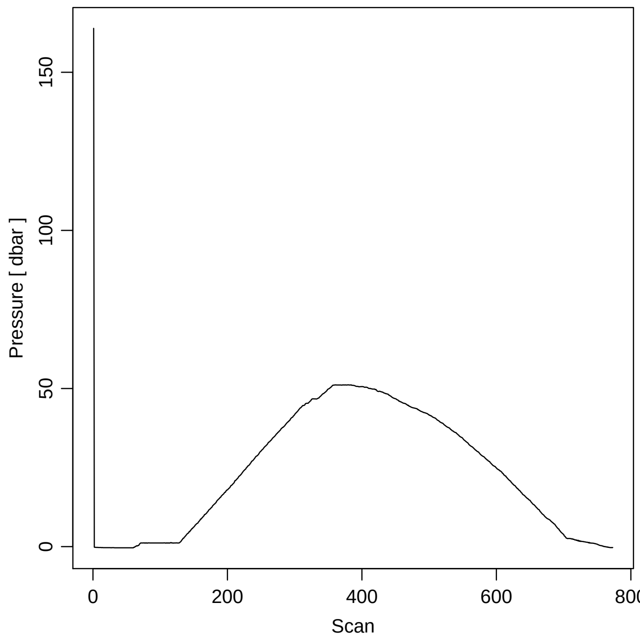


Figure 3: Scanwise plot of the `ctdRaw` sample data set. Note the spike at the start, the equilibration phase before the downcast, and the spurious freshening signal near the start of the upcast.

4.2 Example with raw data

Practicing Oceanographers may be wondering how the CTD cast used in the preceding section was trimmed of equilibration-phase and upcast-phase data. Spurious data from these phases must be trimmed as a first step in processing. For example, consider the following code.

```
data(ctdRaw)
plotScan(ctdRaw)
```

This produces a two-panel plot (Figure 3) of the data as a time-series, revealing not just the (useful) downcast, but also the subsequent upcast sequence. The x-axis in this plot is the scan number, which is a convenient index for extraction of the downcast portion of the profile by an essentially manual method, e.g. proceeding with a sequence of commands such as

```
plotScan(ctdTrim(ctdRaw, "range",
                 parameters=list(item="scan", from=140, to=250)))
plotScan(ctdTrim(ctdRaw, "range",
                 parameters=list(item="scan", from=150, to=250)))
```

This is the "gold standard" method, which is recommended for detailed work. However, for quick work, you may find that the automatic downcast detection scheme works adequately, e.g.

```
ctdTrimmed <- ctdTrim(ctdRaw)
```

It should be noted that `ctdTrim` inserts entries into the object's log file, so that you (or anyone else with whom you share the object) will be able to see the details of how the trimming was done.

Once the profile has been trimmed, you may wish to use `ctd.decimate()` to smooth the data and interpolate the smoothed results to uniformly-spaced pressure values. For example, a quick examination of a file might be done by the following:

```
plot(ctdDecimate(ctdTrim(read.ctd("stn123.cnv"))))
```

4.3 Example with WOCE archive data

The package has a harder time scanning the headers of data files in the WOCE archive format than it does in the Seabird format illustrated in the previous examples. This is mainly because front-line researchers tend to work in the Seabird format, and partly because the WOCE format is odd. For example, the first line of a WOCE file is of the form `CTD,20060609WHPOSIODAM` (or `BOTTLE,...`). Scanning the item to the left of the comma is not difficult (although there are variants to the two shown, e.g. `CTDO` sometimes occurs). The part to the right of the comma is more difficult. The first part is a date (yyyymmdd) so that's no problem. But then things start to get tricky. In the example provided, this text contains the division of the institute (WHPO), the institute itself (SIO), and initial of the investigator (DAM). The problem is that no dividers separate these items, and that there seem to be no standards for the item lengths. Rather than spend a great deal of time coding special cases (e.g. scanning to see if the string `WHOI` occurs in the header line), the approach taken with `oce` is to ignore such issues relating to quirky headers. This frees up time to work on more important things, such as plotting the data.

It is possible to access object constituents directly, e.g. `x@data$salinity` is the salinity stored in a CTD object named `x`, but this is not the preferred method of access. The better scheme is to use the *accessor functions* that are provided with all oce objects. This relies on double-square-bracket notation, e.g. `x[["salinity"]]` yields salinity (from the `data` slot of the object) and `x[["station"]]` yields the station identifier (from the `metadata` slot).

But even though it's possible to manipulate object elements directly, it is typically not a good idea. The reason is that the object will be modified without having an entry inserted into its processing log. Thus, it is a bad idea to write

```
x <- read.ctd("nnsa_00934_00001_ct1.csv", type="WOCE")
x[["institute"]] <- "SIO" # better (using an accessor) but still bad
```

and a better idea to write

```
x <- read.ctd("nnsa_00934_00001_ct1.csv", type="WOCE")
x <- oce.edit(x, "institute", "SIO") # better way
```

Even better, provide a reason for the change, and sign the change with your name:

```
x <- read.ctd("nnsa_00934_00001_ct1.csv", type="WOCE")
x <- oce.edit(x, "institute", "SIO", "human-parsed", "Dan Kelley")
```

For a real-world example (with warts!), visit `http://cchdo.ucsd.edu/data_access?ExpoCode=58JH199410` and download the zip file containing the Arctic section called "CARINA", measured in 1994. Expand the zip file, enter the directory, and run the code below.

```
library(oce)
# Source: http://cchdo.ucsd.edu/data_access?ExpoCode=58JH199410
files <- system("ls *.csv", intern=TRUE)
for (i in 1:length(files)) {
    cat(files[i], "\n")
    x <- read.ctd(files[i])
    if (i == 1) {
        plotTS(x, xlim=c(31, 35.5), ylim=c(-1, 10), type="l", col="red")
    } else {
        lines(x[["salinity"]], x[["temperature"]], col="red")
    }
}
```

What you'll see is an overall T-S diagram for the entire dataset. It may take a while, since the dataset contains over 90,000 observations. You may note that, even though this is an official, quality-controlled dataset, it is not without problems. The graph that is produced by this code has several spurious lines oriented horizontally (indicating spurious salinity) and vertically (indicating spurious temperature). One way to find such values is to put the lines

```
print(range(x[["temperature"]]))
print(range(x[["salinity"]]))
```

after the `read.ctd()` command. One thing you'll find is that station 987 has a minimum salinity range of 0.0009 to 987. These values are clearly in error, as are the temperatures at this spot in the file. (It is perhaps revealing that the spurious salinity is equal to the station number.) Indeed, at this spot in the file it can be seen that the pressure jumps from 1342 to 0, and then starts increasing again; the file contains two profiles, or the same profile twice. This is not the only flaw that is revealed by the graph, and by `range` commands; a generous user would spend a week tracking down such issues, and would then contact the data provider (or the chief scientist of the field work) with specific suggestions for correcting the files. The point here is to highlight how this package can be used with real-world data.

4.4 Section plots

The commands

```
data(section)
plot(section, which=c(1, 2, 3, 99))
```

will plot a summary diagram containing sections of T, S, and σ_θ, along with a chart indicating station locations. In addition to such overview diagrams, `plot` can also create individual plots of individual properties.

Exercise 3. Draw a T-S diagram for the section data, colour-coded by station

The Halifax section is supplied in a pre-gridded format, but some datasets have different pressure levels at each station. For such cases, the `sectionGrid` function may be used, e.g.

```
data(section)
GS <- subset(section, 102<=stationId&stationId<=124)
GSg <- sectionGrid(GS, p=seq(0, 1600, 25))
plot(GSg, which=c(1,99), map.xlim=c(-85,-(64+13/60)))
```

produces Figure 4. The ship doing the sampling was travelling westward from the Mediterranean towards North America, taking 124 stations in total; the `indices` value selects the last few stations of the section, during which the ship heading was changed to run in a northwesterly direction, to cross isobaths (and perhaps, the Gulf Stream) at right angles.

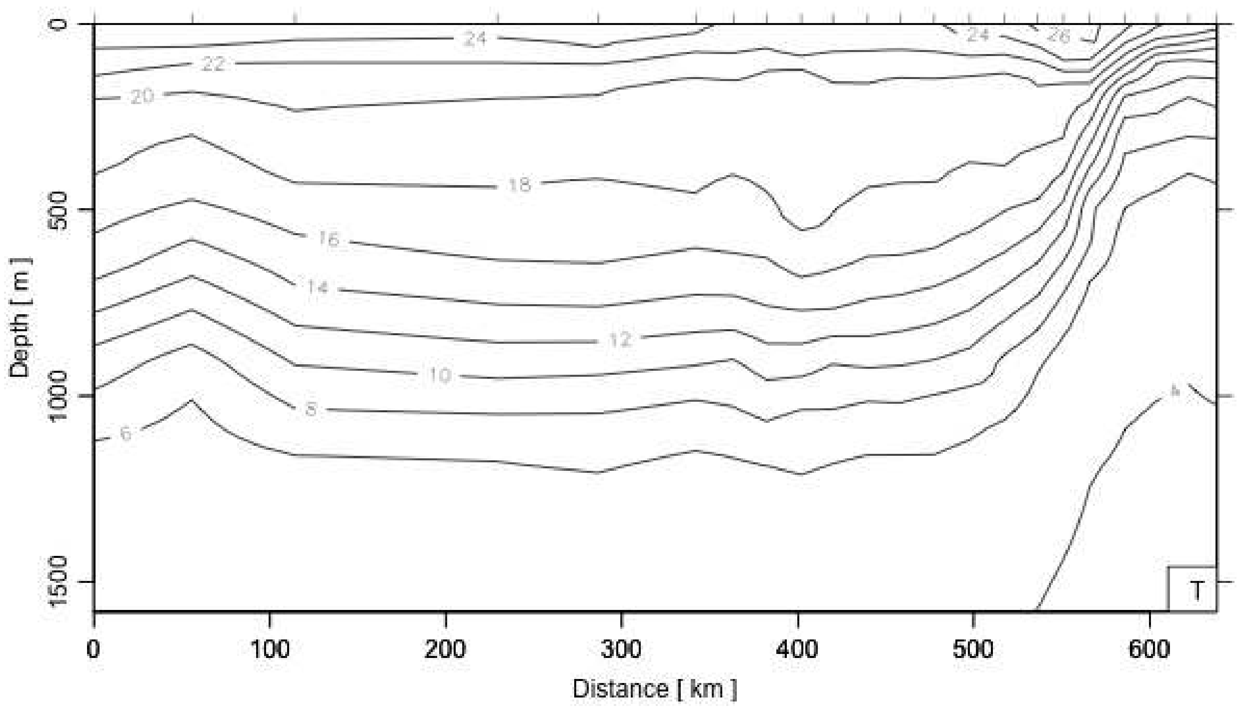
Exercise 4. Plot dynamic height and geostrophic velocity across the Gulf Stream.

5 Coastline and topographic data

Coastline data are available from a variety of sources. In years past, the former NOAA site `http://www.ngdc.noaa.gov/mg` was particularly popular, having the advantage of providing data in Splus format. The function `read.coastline` can handle reading that format (plus some other formats), and `plot` on the resulting object will produce a simple coastline map. The only real advantage over plotting things yourself is that latitude and longitude are scaled to give natural shapes near the centre of the plot. (All `oce` functions that have arguments for latitude and longitude place latitude first, in accordance with convention as documented in the ISO standard 6709 "Standard representation of geographic point location by coordinates.")

Bathymetric charts, or more generally topographic maps, can be produced easily, e.g. (Figure 5)

```
library(oce)
data(topoWorld)
plot(topoWorld, clatitude=30, clongitude=370, span=9000)
```



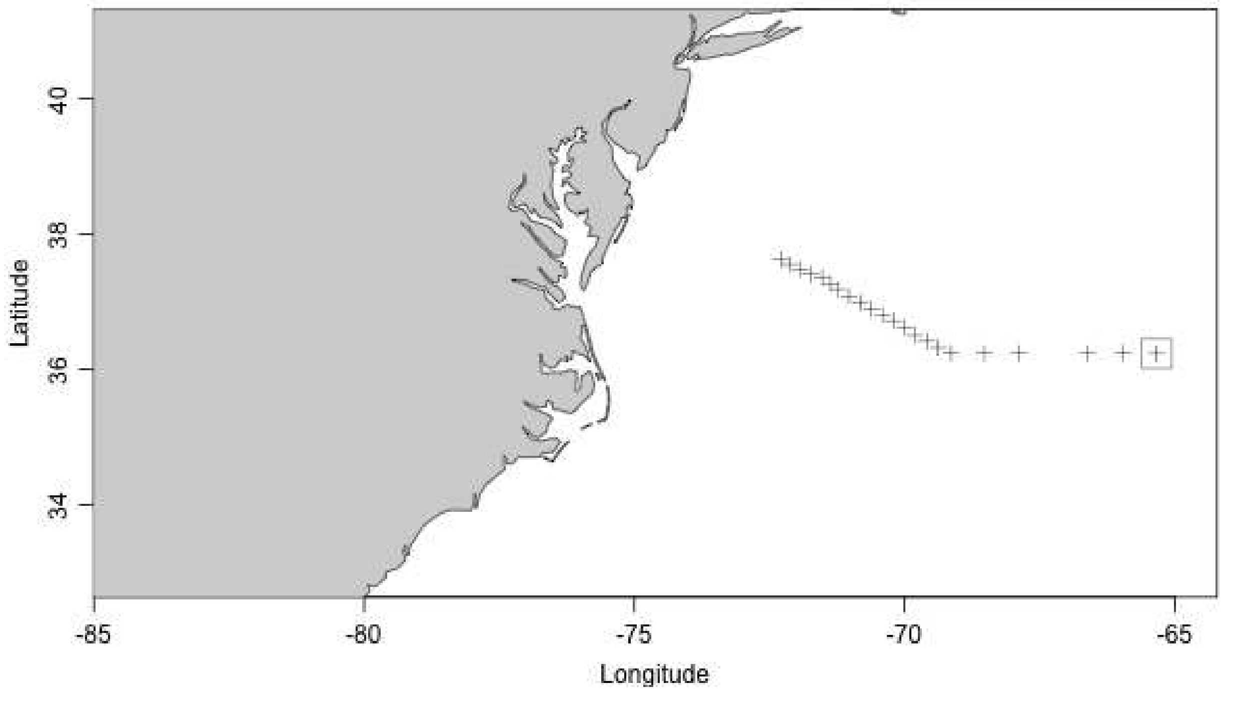


Figure 4: Portion of the CTD section designated A03, showing the Gulf Sream region. The square on the cruise track corresponds to zero distance on the sections.

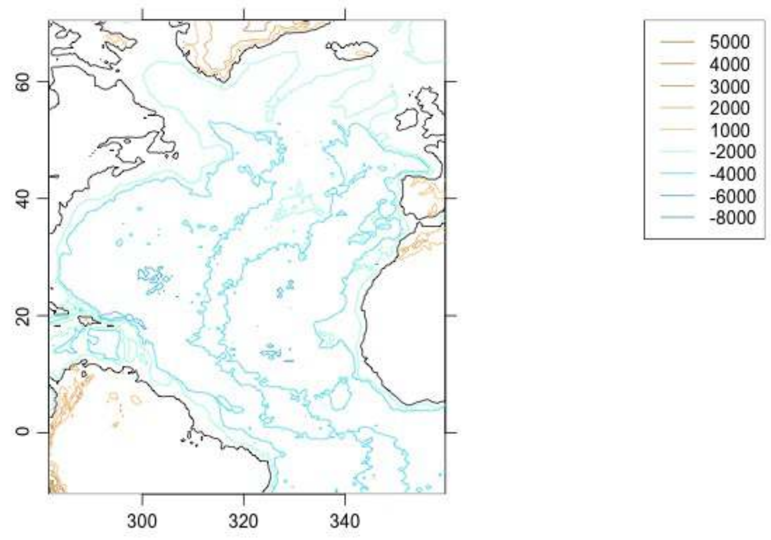


Figure 5: Topography of eastern Canada, centred on Prince Edward Island.

6 Sea-level data

6.1 Time-domain analysis

The commands

```
library(oce)
#sealevel <- read.oce("../../tests/h275a96.dat")
data(sealevel)
plot(sealevel)
```

load and graph a build-in dataset of sea-level timeseries. The result, shown in Figure 6, is a four-panel plot. The top panel is a timeseries view that provides an overview of the entire data set. The second panel is narrowed to the most recent month, which should reveal spring-neap cycles if the tide is mixed. The third panel is a spectrum, with a few tidal constituents indicated. At the bottom is a cumulative spectrum, which makes these narrow-banded constituents quite visible.

Exercise 5. Illustrate Halifax sealevel variations during Hurricane Juan.

Exercise 6. Draw a spectrum of sea-level variation, with the M2 tidal component indicated.

6.2 Tidal analysis

In a future version, tidal analysis will be provided, along the lines of the t-tide package in Matlab. A preliminary version of tidal analysis is provided by the `tidem` function provided in this version of the package, but readers are cautioned that the results are certain to change in a future version. (The problems involve phase, and the inference of satellite nodes.)

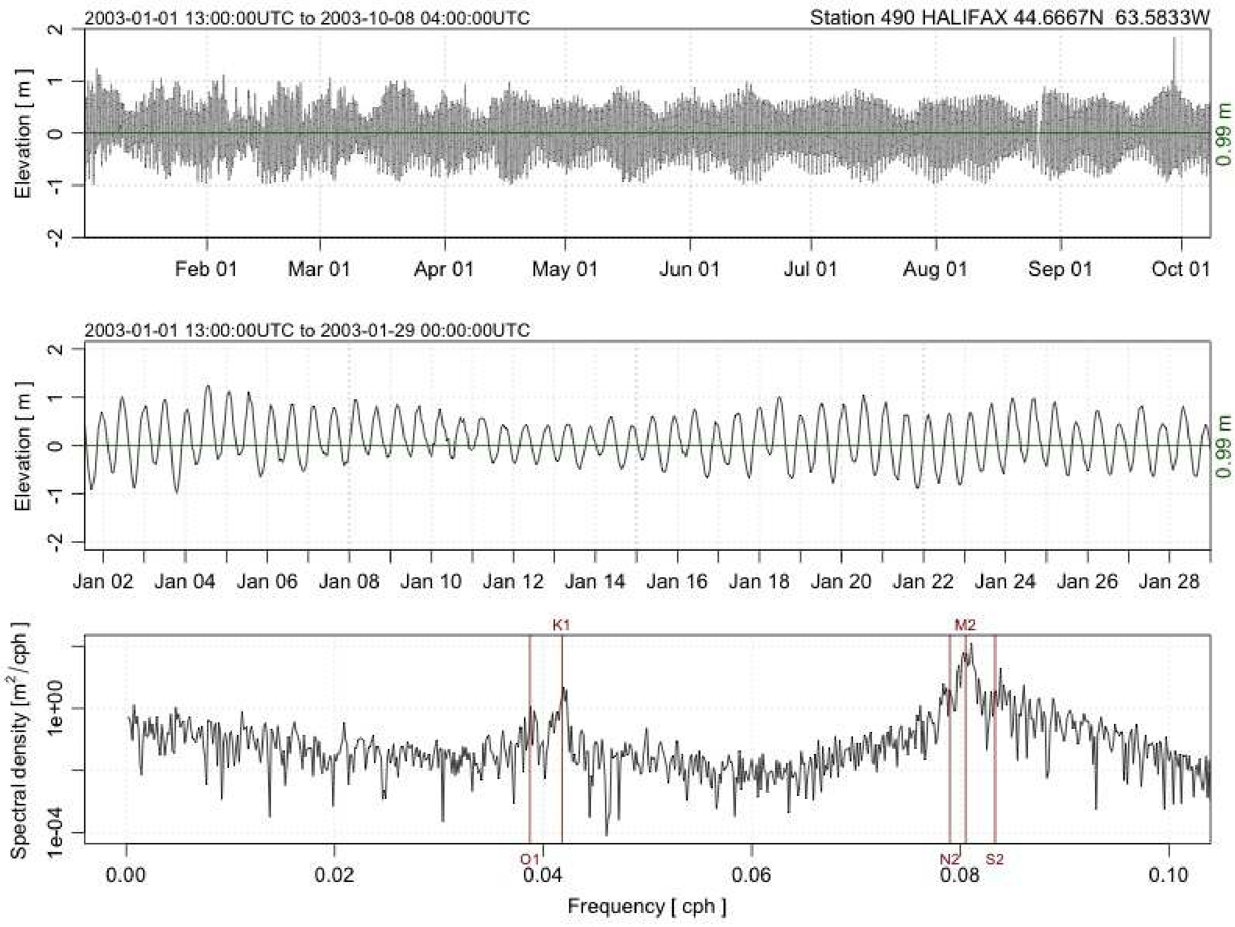


Figure 6: Sea-level timeseries measured in 2003 in Halifax Harbour. (The spike in September is the storm surge associated with Hurricane Juan, regarded by the Canadian Hurricane Centre to be one of the most powerful and damaging hurricanes to ever hit Canada.)

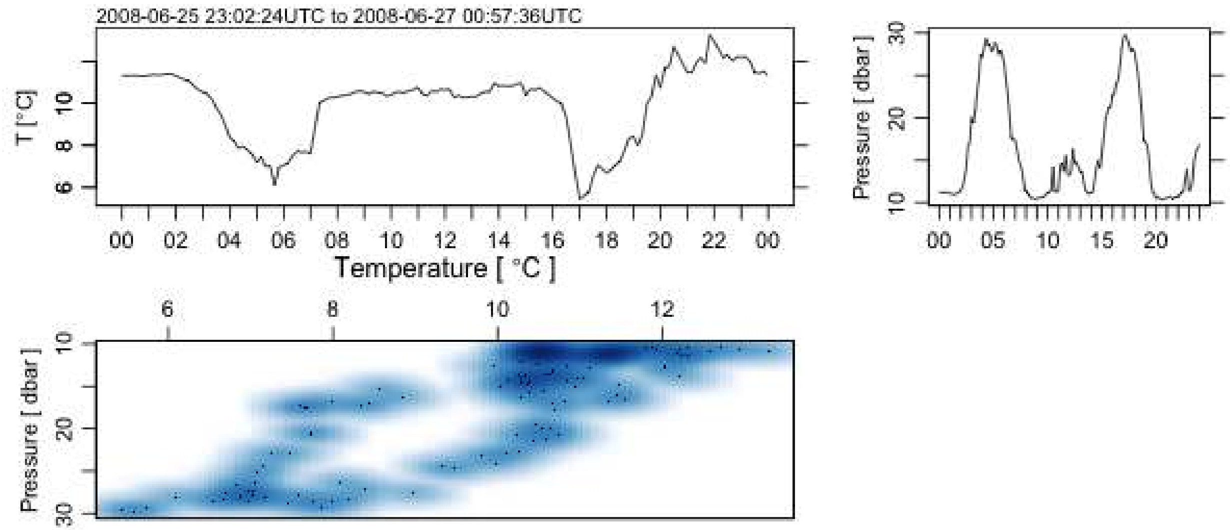


Figure 7: Measurements made with a pressure-temperature recorder in the St Lawrence Estuary. Note that the record contains in-air measurements made during an hour prior to lowering into the water, and during several days of transit from the field site back to Dalhousie University. The text suggests a procedure for isolating in-water data, and removing the approximately 10 dbar atmospheric component of pressure.

7 Temperature-Depth Recorder data

The commands

```
library(oce)
data(logger)
plot(logger, useSmoothScatter=TRUE)
```

produce a plot (Figure 7) of temperature-depth logger measurements made during St Lawrence Estuary Internal Wave Experiment, an CFCAS/NSERC-funded research program in which the `oce` author was a principal investigator. The device was mounted on a mooring that tilted significantly with the tide, yielding strong tidal signals in pressure as well as temperature, thus providing a rough indication of the temperature profile. Note that this graph contains in-air as well as in-water data.

Exercise 7. Trim the in-air measurements from this `logger` record, and then remove the atmospheric pressure from the signal.

8 Acoustic Doppler Current Profiler data

The commands

```
library(oce)
data(adp)
plot(adp, which=1, adorn=expression({lines(x[["time"]], x[["pressure"]])}))
```

produce Figure 8, showing currents made in the St Lawrence Estuary Internal Wave Experiment.

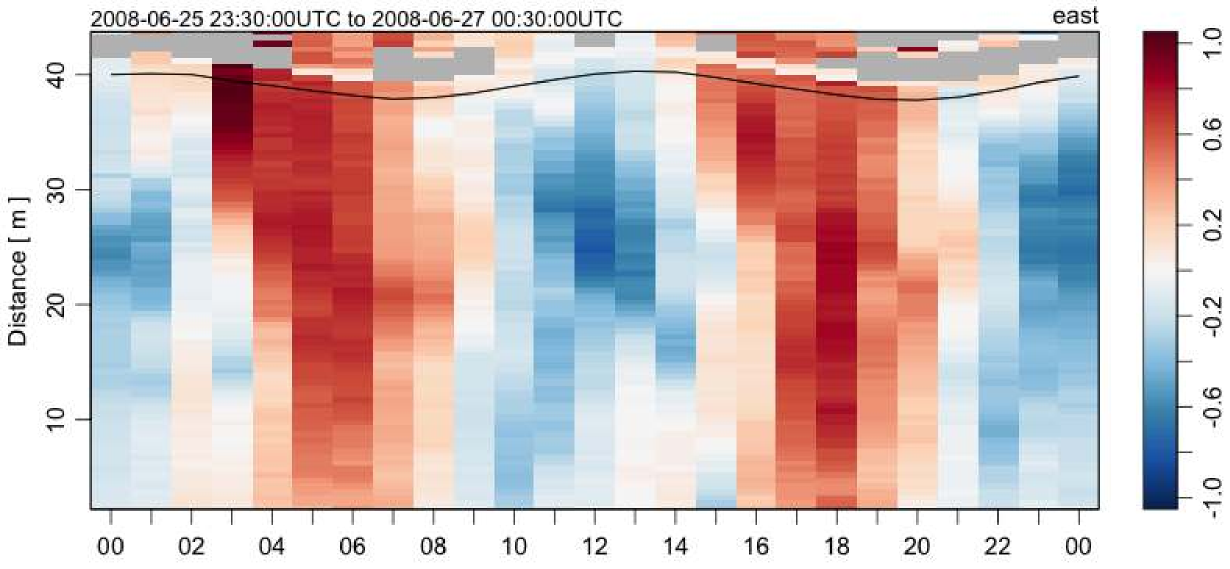


Figure 8: Measurements made with an ADP in the St Lawrence Estuary. The data were recorded in beam format, so a two-step process is used to convert to enu coordinates. The black curve indicates the water surface, inferred from the pressure measured by the ADP.

9 Working in non-English languages

Many of the `oce` plotting functions produce axis labels that can be displayed in languages other than English. At the time of writing, French, German, Spanish, and Mandarin are supported in at least a rudimentary way. Setting the language can be done at the general system level, or within R; the latter is illustrated below for French (which is coded `"fr"` in the ISO 639 language standard).

```
library(oce)
Sys.setenv(LANGUAGE="fr")
data(ctd)
plot(ctd)
```

Most of the translated items were found by online dictionaries, and so they may be incorrect for oceanographic usage. Readers can help out in the translation effort, if they have knowledge of how nautical words such as “Pitch” and “Roll” and technical terms such as “Absolute Salinity” and “Potential Temperature” should be written in various languages.

10 The future of oce

The present version of `oce` can only handle data types that the author has been using lately in his research. New data types will be added as the need arises in that work, but the author would also be happy to add other data types that are likely to prove useful to the Oceanographic community. (The data types need not be restricted to Physical Oceanography, but the author will need some help in dealing with other types of data, given his research focus.)

Two principles will guide the addition of data types and functions: (a) the need, as perceived by the author or by other contributors and (b) the ease with which the additions can be made. One might call this development by triage, by analogy to the scheme used in Emergency Rooms to focus medical effort.

11 Development website

The site `http://github.com/dankelley/oce` provides a window on the development that goes on between the CRAN releases of the package. Please visit the site to report bugs, to suggest new features, or just to see how `oce` development is coming along. Note that the `development` branch is used by the author in his work, and is updated so frequently that it must be considered unstable, at least in those spots being changed on a given day. Every week or so, as the `development` branch stabilizes, the changes are merged back into the `master` branch. Official CRAN releases derive from the `master` branch, and are done when the code is considered to be of reasonable stability and quality. This is all in a common pattern for open-source software.

Answers to exercises

Exercise 1 – Seawater properties.

```
library(oce)
swRho(34, 10, 100)
```

```
[1] 1026.624
```

```
swTheta(34, 10, 100)
```

```
[1] 9.988598
```

```
swRho(34, swTheta(34, 10, 100), 0)
```

```
[1] 1026.173
```

```
swRho(34, swTheta(34, 10, 100, 200), 200)
```

```
[1] 1027.074
```

```
plotTS(as.ctd(c(30,40),c(-2,20),rep(0,2)), grid=TRUE, col="white")
```

Exercise 2 – Profile plots. Although one may argue as to the limits of the pycnocline, for illustration let us say it is in 5bar to 12dbar range. One way to do this is

```
library(oce)
data(ctd)
pycnocline <- ctdTrim(ctd, "range",
                      parameters=list(item="pressure", from=5, to=12))
plotProfile(pycnocline, which="density+N2")
```

and another is

```
library(oce)
data(ctd)
pycnocline <- subset(ctd, 5<=pressure & pressure<=12)
plotProfile(pycnocline, which="density+N2")
```

Exercise 3 – TS diagram for section data. The simplest way is to use accessor functions to extract salinity, etc.

```
library(oce)
data(section)
ctd <- as.ctd(section[["salinity"]], section[["temperature"]], section[["pressure"]])
plotTS(ctd)
```

but the point of *this* exercise is to do the extraction manually, so a more appropriate solution is as follows

```
library(oce)
data(section)
SS <- TT <- pp <- id <- NULL
n <- length(section@data$station)
for (stn in section@data$station) {
    SS <- c(SS, stn[["salinity"]])
    TT <- c(TT, stn[["temperature"]])
    pp <- c(pp, stn[["pressure"]])
}
ctd <- as.ctd(SS, TT, pp)
plotTS(ctd)
```

Exercise 4 – Gulf Stream. (Try `?swDynamicHeight` for hints on smoothing.)

```
library(oce)
GS <- subset(section, 102<=stationId&stationId<=124)
dh <- swDynamicHeight(GS)
par(mfrow=c(2,1))
plot(dh$distance, dh$height, type="b", xlab="", ylab="Dyn. Height [m]")
grid()
# 1e3 metres per kilometre
f <- coriolis(GS@data$station[[1]]@metadata$latitude)
g <- gravity(GS@data$station[[1]]@metadata$latitude)
v <- diff(dh$height)/diff(dh$distance) * g / f / 1e3
plot(dh$distance[-1], v, type="l", col="blue", xlab="Distance [km]", ylab="Velocity [m/s]")
grid()
abline(h=0)
```

Exercise 5 – Halifax sealevel during Hurricane Juan.. A web search will tell you that Hurricane Juan hit about midnight, 2003-sep-28. The author can verify that the strongest winds occurred a bit after midnight – that was the time he moved to a room without windows, in fear of flying glass.

```
library(oce)
data(sealevel)
# Focus on 2003-Sep-28 to 29th, the time when Hurricane Juan caused flooding
plot(sealevel,which=1,xlim=as.POSIXct(c("2003-09-24","2003-10-05"), tz="UTC"))
abline(v=as.POSIXct("2003-09-29 04:00:00", tz="UTC"), col="red")
mtext("Hurricane\nJuan", at=as.POSIXct("2003-09-29 04:00:00", tz="UTC"), col="red")
```

Exercise 6 – Sealevel spectrum.

The first step is to load the data.

```
library(oce)
data(sealevel)
```

Next, we extract the elevation data

```
elevation <- sealevel[["elevation"]]
```

and then we return to standard R processing, to compute the spectrum, and indicate the M2 tide.

```
spectrum(elevation, spans=c(3,7))
abline(v=1/12.42)
mtext("M2",at=1/12.42,side=3)
```

Exercise 7 – Pressure-temperature Recorder plot.

The accuracy of automatic removal of the atmospheric component done with `logger.trim()` can be quite good if the instrument has recorded in the air at the start and end of the record. Even so, it always makes sense to investigate the data by eye, in order to select a time window. Manual selection of a time window is also important when a researcher is dealing with a set of instruments whose measuremnts are to be combined and so need a uniform time base (e.g. to make a matrix to be contoured).

The best plan is simply to plot the data interactively, and then to narrow in on start and end times by plotting trial values, e.g.

```
abline(v=as.POSIXct("2008-06-25 00:00:00"),col="red")
```

with arrow keys being used to repeat commands that get edited as the key times are located.

Index