Ranadu: An R Package for NCAR-Aircraft Data Files

Research Aviation Facility, EOL/NCAR

Version: February 2019

Chapter 1

Introduction

The intent of this web site is to assist new users of NCAR/EOL/RAF-produced aircraft data if they would like to use R in their data analysis projects. The intent here is to provide "layers", starting with a very simple guide to constructing preliminary plots and extending to the use of R, RStudio, and knitr to produce manuscripts and reproducible and documented data-analysis projects. The emphasis here is on "Ranadu", an R package that is intended to facilitate use of the data archives produced by the data systems on the NSF/NCAR/EOL/RAF research aircraft. (NSF=National Science Foundation; NCAR=National Center for Atmospheric Research; EOL=Earth Observing Laboratory; RAF=Research Aviation Facility) All the recent data files are in netCDF format, so that is the format emphasized here. Those files contain measurements made in field campaigns that use the NSF research aircraft operated by NCAR, presently consisting of a C-130 and a Gulfstream V. A list of recent projects is available at this EOL web page, and data requests can be made via links on that page. Information regarding the instruments and the processing algorithms are available at these respective web sites: https://www.eol.ucar.edu/aircraft-instrumentation and ProcessingAlgorithms.pdf. The latter also provides references to the netCDF format, the variable names in common use, and algorithms used to calculate the variables in the data files.

This package can also be used to ingest the data into Python3 so that subsequent processing can use the tools of Python3. The plotting functions and all other functions can be called from Python3 via "rpy2" as described in Chapter 6.3.

If you prefer to see a PDF-format version of the information at this web site, you can download a copy at this URL.

Chapter 2

Getting Started

This chapter or "layer" summarizes a few key tools that will enable a new user to get started. Of course, it is certainly possible to work with the NCAR/EOL/RAF data files using R routines without reference to the "Ranadu" package featured here, but this discussion will describe use of that package. There are instructions for installing the package in the "RanaduManual.pdf", and there it is also recommended to use RStudio as the user GUI for working with R. Once R, RStudio and Ranadu are installed, it will be simple to use the functions highlighted in the remainder of this chapter to get started. The recommended way to install the Ranadu package is via the R command `devtools::install_github('NCAR/Ranadu')`. The key functions are getNetCDF(), for reading the netCDF file and producing an R data.frame with the measurements, and DataFileInfo(), for checking the properties of the netCDF file. These two functions provide a useful starting point for all data-analysis projects.

2.1 DataFileInfo()

A useful first look at a netCDF file is provided by Ranadu::DataFileInfo(), which returns characteristics like the project name, flight number, date/times, variable names, and the data rate. In addition, this function returns a set of "measurands" (measured properties of the atmosphere like air temperature) and the set of variables that provide measurements of that measurand. The measurands in a particular file (the one referenced above) are shown below:

Generating R code:

```
Project <- 'WECAN'
FlightNumber <- 6
fname <- sprintf ('%s%s/%srf%02d.nc', Ranadu::DataDirectory(),
         Project, Project, FlightNumber)
FI <- DataFileInfo(fname)
names (FI$Measurands)
```

```
##  [1] ""
##  [2] "altitude"
##  [3] "air_temperature"
##  [4] "air_pressure"
##  [5] "dew_point_temperature"
##  [6] "water_vapor_pressure"
##  [7] "latitude"
##  [8] "longitude"
##  [9] "height"
## [10] "humidity_mixing_ratio"
## [11] "barometric_altitude"
## [12] "platform_pitch_angle"
## [13] "atmosphere_cloud_liquid_water_content"
## [14] "air_pressure_at_sea_level"
## [15] "relative_humidity"
## [16] "platform_roll_angle"
## [17] "solar_azimuth_angle"
## [18] "solar_elevation_angle"
## [19] "solar_zenith_angle"
## [20] "platform_speed_wrt_air"
## [21] "platform_orientation"
## [22] "air_potential_temperature"
## [23] "equivelent_potential_temperature"
## [24] "platform_course"
## [25] "virtual_temperature"
## [26] "eastward_wind"
## [27] "northward_wind"
## [28] "wind_from_direction"
## [29] "upward_air_velocity"
## [30] "wind_speed"
```

The variables that provide redundant measurements of a specific measurand are named lists with the measurand name and can be displayed by printing the measurand name, as in the following example:

Generating R code:

```
FI$Measurands$air_temperature
```

```
## [1] "ATF1" "ATH1" "ATH2" "ATX"
```

In addition, the "long_name" describing a variable (e.g., here "ATX") can be found as follows:

Generating R code:

```
FI$LongNames[which('ATX' == FI$Variables)]
```

```
## [1] "Ambient Temperature, Reference"
```

To see all the lists of information contained in the DataFileInfo list, print the names as follows:

Generating R code:

```
names(FI)
```

```
##  [1] "Number"     "Project"    "Platform"   "DataFile"   "Start"
##  [6] "End"        "Rate"       "LatMin"     "LatMax"     "LonMin"
## [11] "LonMax"     "Variables"  "LongNames"  "Measurands"
```

Examining these can help a user understand what is included in a particular data file.

2.2 getNetCDF() and the resulting data.frame

A central component of the Ranadu structure is the Ranadu data.frame, produced by reading the netCDF data file. It has a structure similar to that of a spreadsheet, with rows corresponding to measurement times and columns corresponding to measurements. The data.frame has these features:

1. Each row corresponds to a unique time, and times are sequential (possibly with gaps). For data rates higher than 1 Hz, rows are produced for each time interval; i.e., 25 rows per second for 25-Hz files. When variables are present in the netCDF file at a slower rate, interpolation is used to produce the higher rate.

2. Each measurement corresponds to a single column. When there are multiple measurements of a given measurand (e.g., temperature), each individual measurement has its own column. There is a significant exception: For instruments producing size-distribution arrays, the entire array occupies one column of the data.frame.

3. Attributes describing the data.frame and the variables are carried with the data.frame. For example, variables often have "short_name" and "long_name" attributes, and these can be examined by looking at the variable attributes.

The data.frame is constructed by Ranadu::getNetCDF(fname, variables), which uses the ncdf4 package of routines to read the netCDF file. An example of a subset of the data.frame is shown here:

Generating R code:

```
Project <- 'WECAN'
FlightNumber <- 6
fname <- sprintf ('%s%s/%srf%02d.nc', Ranadu::DataDirectory(),
         Project, Project, FlightNumber)
Variables <- Ranadu::standardVariables(c('UXC', 'VYC'))
Data <- Ranadu::getNetCDF(fname, Variables)
print (sprintf ('Data from data file %s', fname))
```

```
## [1] "Data from data file /Data/WECAN/WECANrf06.nc"
```

```
print (tibble::as.tibble(Data))  # or print(head(Data))
```

```
## # A tibble: 23,701 x 18
##    Time                  ATX  DPXC   EWX GGALT  LATC  LONC   MACHX    MR
##    <dttm>              <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
##  1 2018-08-03 19:55:00  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  2 2018-08-03 19:55:01  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  3 2018-08-03 19:55:02  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  4 2018-08-03 19:55:03  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  5 2018-08-03 19:55:04  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  6 2018-08-03 19:55:05  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  7 2018-08-03 19:55:06  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  8 2018-08-03 19:55:07  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
##  9 2018-08-03 19:55:08  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
## 10 2018-08-03 19:55:09  29.9    NA    NA  865.  43.6 -116. 0.00125    NA
## # ... with 23,691 more rows, and 9 more variables: PALT <dbl>, PSXC <dbl>,
## #   QCXC <dbl>, TASX <dbl>, WDC <dbl>, WSC <dbl>, WIC <dbl>, UXC <dbl>,
## #   VYC <dbl>
```

Here is an explanation of some aspects of loading this data.frame:

1. Ranadu::DataDirectory() returns the location of the data directory on various systems, to avoid the necessity of changing this when moving among systems. It may return '/scr/raf_data/' or '/Data/' or '~/Data/' depending on the file system.

2. The function Ranadu::standardVariables() returns a set of commonly used variable names. Additional variables provided to the routine (here, 'UXC' and 'VYC') are added to the variable list.

3. Ranadu::getNetCDF() produces the data.frame. The special case where Variables <- 'ALL' will return all available variables.[1] Two additional optional arguments to getNetCDF() are "Start" and "End"; if set, the range of time values in the data.frame will be restricted to be between those two times. See "?Ranadu::getNetCDF" for complete information on this function.

4. The last statement, where the data.frame is converted to a tibble, is used here because the print function for tibbles produces a more concise and clearer format than that for a data.frame. "print (head (Data))" could have been used also. More information on tibbles is included later in this document. The resulting data.frame has 23,701 rows and 18 columns.

2.3 Using the data.frame

2.3.1 Simple Plots

It is straightforward to plot variables in the data.frame using standard R functions. For example, the following code plots the altitude vs. time using the data.frame loaded previously:

Generating R code:

```
plot(Data$Time, Data$GGALT, type='l')
```

Many of the Ranadu tools are aimed at making construction of such plots straightforward while supporting various manipulations of the style and content of the plots. Many of these are discussed in later chapters. However, at this point you will be able to conduct extensive data-analysis projects using only the standard tools provided by R.

[1] But use this cautiously because size-distribution variables are special and sometimes cause problems when manipulating the resulting data.frame. This is discussed later in association with "tibbles."

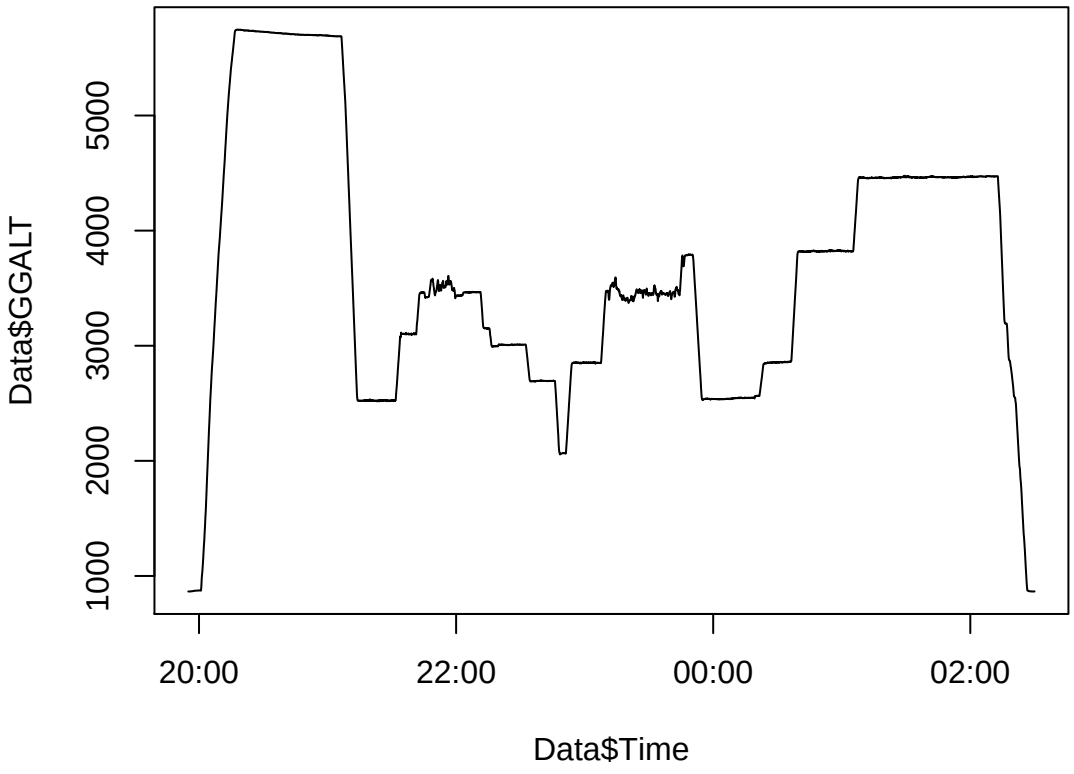

Figure 2.1: Geometric altitude vs time for WECAN research flight 6, 3 August 2018.

2.3.2 Using plotWAC() and ggplotWAC()

The function Ranadu::plotWAC() calls the standard R function "plot" with a particular set of conventions. Some reasons you may want to consider using it include the following:

time offset: The convention in the NCAR/EOL/RAF netCDF files is that the time variable represents the start of the interval over which measurements are averaged, so a 1-Hz variable with a specified time is actually an average where the mean time is 0.5 s later. Plots generated by plotWAC() adjust for this offset. For this same reason, you may want to use the routine Ranadu::lineWAC() to add lines to the plot, instead of the standard "lines" routine provided by R.

plot format: The set of conventions regarding time labels, axis formats, and legends may be preferable to those that are standard with "plot()" and will save you from making those tailoring adjustments.

pipe compatible: The function plotWAC() can be used in a pipe where the piped variable is a data.frame tailored to contain specified variables to construct multiple-variable plots. A similar pipe to "plot()" will produce a faceted plot of each variable vs. each other variable, which may not be what you want.

Figure 2.2 shows an example.

Generating R code:

```
plotWAC(Data[, c('Time', 'GGALT')])
```

Another option is provided by Ranadu::ggplotWAC(), as shown in Fig. 2.3:

Generating R code:

```
ggplotWAC(Data[, c('Time', 'GGALT')])
```

Additional examples showing the advantages of constructing plots with pipes will be presented in later chapters of this document. For more information on the use of these plotting routines, see "?Ranadu::plotWAC" and "?Ranadu::ggplotWAC".

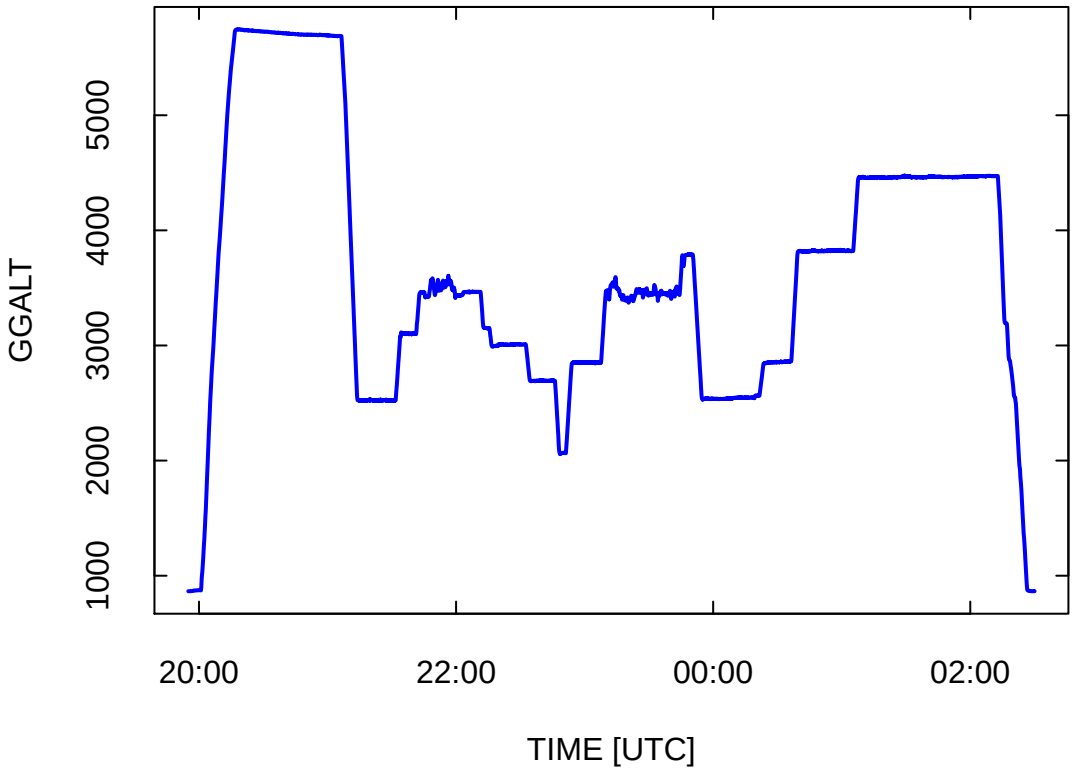

Figure 2.2: Example of the same plot as the preceding figure but generated with Ranadu::plotWAC().

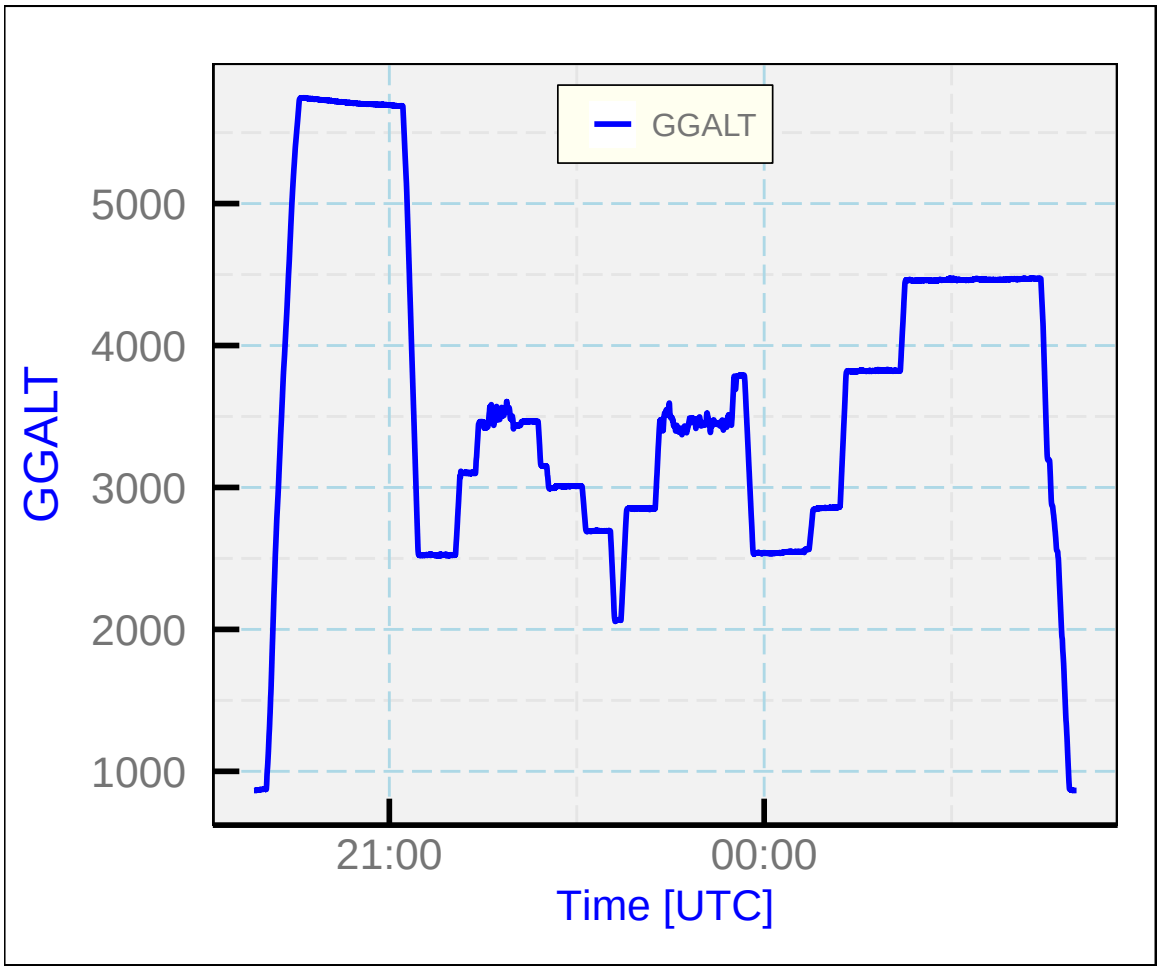

Figure 2.3: Example of the same plot as the preceding figure but generated with Ranadu::ggplotWAC().

Chapter 3

Overview of Ranadu functions

3.1 More about constructing data.frames

These Ranadu functions can be of use when constructing and modifying data.frames:

- `getNetCDF ( )`: loads a data.frame with requested variables
- `standardVariables ( )`: defines a common set of variables for input to getNetCDF(). This list is the default if no variables are specified in the call to getNetCDF. The optional argument is a list of character names that specify additional variables to include.
- `DataDirectory ( )`: This is system dependent and is intended to aid in portability by providing the usual location of the netCDF data files. It tries a set of likely names, but may need redefinition for some file systems.
- `getIndex ( )`: finds the index for a specified time
- `setRange ( )`: sets a range of indices covering a specified time interval.
- `selectTimes ( )`: produces a new data.frame with a limited time range.

In addition, some functions provided by the package "dplyr" can be very useful when working with data.frames. These actions can be implemented by normal R subsetting also (e.g., using "[[...]]" notation), but using these dplyr functions serves to clarify what the steps in the code are doing:

- `dplyr::select ( )`: produces a new data.frame with only the variables listed as arguments in the call to the function.
- `dplyr::filter ( )`: produces a subset data.frame where the argument is a logical test applied to the data.frame rows that must be met for the row to be included in the returned data.frame.
- `dplyr::mutate ( )`: adds new variables to the data.frame.

3.2 Getting information about the data

Some Ranadu functions provide information about the data set. A good starting point is DataFileInfo(), discussed in the previous chapter. In addition, these functions may be helpful:

1. `Ranadu::getAttributes():` The data.frames produced by getNetCDF() have assigned attributes that match those in the original netCDF file, and these can be retrieved via getAttributes(), where the argument can be either the data.frame (for which the returned attributes are those associated with the data file) or a variable name (for which the attributes are those associated with the variable).[1] The result of the function call is a printed listing of attributes and creation of a list (here in Z) that contains the attributes, so it is usually preferable to assign the returned value to something to avoid the double listing that would be produced otherwise.

2. `Ranadu::TellAbout()` provides some information about the characteristics of a data.frame or a variable in the data.frame.

3. `Ranadu::getStartEnd()` returns the start and end times of a data frame.

4. `Ranadu::ValueOf() and Ranadu::ValueOfAll()` for a specified time return the value of a specified variable or of all variables, respectively.

The standard R routines "dim()", "names()" and "str()" will also provide information about a data.frame or a variable.

3.3 Plotting routines

The routines plotWAC() and ggplotWAC() were introduced in the previous chapter. The following is a list of other Ranadu routines that produce plots. More detail about these routines is included in the next chapter.

1. `Ranadu::plotTrack()` plots a flight track on a background that shows geographic boundaries. In its simplest form it is called with a single argument of a data.frame that contains at least latitude and longitude in variables named LATC and LONC. This routine also has an option to plot the track in a reference frame that drifts with the wind. See ?plotTrack for more information on options.

2. `Ranadu::VSpec()` constructs a plot of the variance spectrum for a specified variable. See also `Ranadu::CohPhase()` for plots of the coherence and phase relationship between veriables.

[1] Subsetting will often lose some attributes so special steps are needed to preserve them if that is desired.

3. Ranadu::lineWAC() adds a line to a plot previously constructed by plotWAC(), with adjustment of the plotted time to correspond to the center of the interval averaged to obtain the measurements. Otherwise, like "lines()" in standard R.

4. Ranadu::plotSD() plots the size distribution measured by cloud or aerosol particle probes.

5. Ranadu::contourPlot() plots a scattergram-like display where the density of points is represented by colored areas.

6. Ranadu::DemingFit() calculates a Deming fit to two variables (in which the distance from the fitted line to the two-dimensional locations of the measurements is minimized) and plots the resulting fit.

7. Ranadu::SkewTSounding() plots a skew-T thermodynamic diagram with measurements from a data.frame averaged and superimposed on the diagram. Options are available to include a hodograph and the denote values of the LCL and CAPE.

3.4 Computational Algorithms

A set of functions provide calculations like those used to process the original files and, in some cases, to extend those calculations. A full list of Ranadu functions can be viewed by using the R command "?Ranadu", and clicking on the items in that list then will provide the standard "help" information for the functions. Some of these are listed below. For additional details, see the technical note titled "Processing Algorithms".

1. AdiabaticTandLWC( ): Calculate the temperature and liquid water content produced by adiabatic ascent.

2. AirTemperature( ): The standard calculation of temperature from the measured recovery temperature.

3. BoltonEquivalentPotentialTemperature( ): Equivalent potential temperature as calculated using the Bolton formula. In standard processing, this has been replaced now by the Davies-Jones representation; see "EquivalentPotentialTemperature( )" below.

4. ButterworthFilter( ): Implementation of a Butterworth low-pass filter, used in wind processing; see "ComplementaryFilter( )".

5. calcAttack( ): This function calculates the angle of attack from the pitch and rate of climb of the aircraft under the assumption that the vertical wind is zero. This algorithm may be useful when determining or checking sensitivity coefficients for the measured angle of attack.

6. `ComplementaryFilter( )`: The standard calculation of wind uses this complementary filter to combine the aircraft ground-speed components measured by the inertial system at high rate with the corresponding components measured by the global positioning system at lower rate.

7. `CorrectHeading( )`, `CorrectPitch( )` and `CorrectRoll( )`: Algorithms that use the observed Schuler oscillation of the inertial system to improve the measurements of pitch and roll, and that calculates a related correction to the heading based on a comparison of the observed acceleration vector and the velocity derivatives from the global positioning system.

8. `DPfromE( )`: Calculation of the dewpoint temperature corresponding to a specified vapor pressure.

9. `EquivalentPotentialTemperature( )`: The pseudo-adiabatic equivalent potential temperature calculated using the Davies-Jones (2009) formula. See also BoltonEquivalentPotentialTemperature( ) and RossbyEquivalentPotentialTemperature( ) for alternate values.

10. `GeoPotHeight( )`: Calculates the geopotential height associated with a specified geometric height above sea level and location.

11. `Gravity( )`: The acceleration of gravity as represented by the Somigliana formula.

12. `GV_AOAfromRadome( )`, `GV_YawFromRadome( )`: Algorithms used to find the angles of attack and sideslip or yaw from pressure differences measured on the radome of the GV.

13. `KingProbe( )`: Calculation of the liquid water content from the power measured by the CSIRO/King probe.

14. `LCL( )`: Find the lifted condensation level from the observed pressure, temperature, and water-vapor mixing ratio.

15. `MachNumber( )`: Calculate the Mach number from the measured dynamic pressure, ambient pressure, and humidity.

16. `memCoef( )`, `memEstimate( )`: Routines used to implement the "maximum entropy" method of spectral estimation; see "VSpec()".

17. `MixingRatio( )`: Calculates the water vapor mixing ratio.

18. `MurphyKoop( )`, `MurphyKoopIce( )`: Calculate the water vapor pressure from the Murphy-Koop formulas.

19. `PCORfunction( )`: The algorithm used to correct measured ambient and dynamic pressures for the static defect.

20. `PotentialTemperature( )`: Calculates the potential temperature.

21. `PressureAltitude( )`: Calculates the pressure altitude from the pressure.

22. `RAFdata`: This is a sample data.frame as might be constructed by getNetCDF(), containing a short period of measurements from an NSF/NCAR GV flight in a project called "IDEAS-4". The data.frame contains a set of measurements, one row per second, and a "Time" variable. This is provided for use in the Ranadu examples and can be used to test various Ranadu functions.

23. `RecoveryFactor( )`: Returns the recovery factor used to process measurements of air temperature from probes exposed to the airstream.

24. `RossbyEquivalentPotentialTemperature( )`: The Rossby formula for the equivalent potential temperature.

25. `SpecificHeats( )`: At the value of the ratio of water vapor pressure to total pressure provided as an argument, returns the values of the specific heat at constant pressure, the specific heat at constant volume, and the gas constant.

26. `Sqs( )`: The quasi-steady supersaturation that will exist in a cloud with specified droplet size distribution and updraft.

27. `StandardConstant( )`: Provides standardized values of some constants used in the processing algorithms. See "StandardConstant("?") for a list of available constants.

28. `TrueAirspeed( )`: Calculate the airspeed from the Mach number and temperature.

29. `VirtualTemperature( )` and `VirtualPotentialTemperature( )`: Functions to calculate these variables.

30. `WetEquivalentPotentialTemperature( )`: Wet-equivalent potential temperature.

31. `WindProcessor( )`: Calculates the wind vector from the basic measurements and adds new wind variables to the input data.frame.

3.5 Utility Functions

The following are some of the Ranadu functions that are provided to assist in various data-analysis tasks. The "getNetCDF()", "DataDirectory()" and "standardVariables()" functions were already described. Others include:

1. `binStats( )`: A function to calculate average values and standard deviations in bins of a specified variable, as might to useful when constructing error-bar or box-and-whisker plots.

2. `detrend( )`: Simple removal of the mean and trend from a time-series variable. Used in the spectral-analysis routines.

3. `df2tibble( )`: Convert a data.frame to a "tibble", a related data structure particularly suited to many data-analysis steps. `tibble2df( )` transforms from a tibble back to a data.frame.

4. `LagrangeInterpolate()` implements standard Lagrange interpolation.

5. `makeNetCDF( )`: A utility that writes a new netCDF-format file from the measurements and attributes contained in a data.frame.

6. `ncsubset( )`: Produce a subset netCDF file from an existing netCDF file.

7. `OpenInProgram( )`: View the variables in a data.frame in another program like ncplot or Xanadu.

8. `RAFdata`: This is a sample data.frame as might be constructed by getNetCDF(), containing a short period of measurements from an NSF/NCAR GV flight in a project called "IDEAS-4". The data.frame contains a set of measurements, one row per second, and a "Time" variable. This is provided for use in the Ranadu examples.

9. `removeSpikes( )`: Find "spikes" in a variable and replace them with linear-interpolated values. Useful in some studies of spectral variance where spikes distort the spectrum.

10. `RSubset( )`: Produce a subset of a data.frame while preserving attributes. This is useful because some standard R subsetting actions do not preserve attributes.

11. `selectTimes( )`: Produce a new data.frame with a subset of the time range included in the original data.frame. (Uses setRange.) This is of particular use in pipes.

12. `setRange( )`: Find the data.frame indices corresponding to a specified time range in a data.frame.

13. `setVariableList( )`: An interactive way of setting a list of variables.

14. `ShiftInTime( )`: Move a time-series variable forward or backward in time.

15. `SmoothInterp( )`: Fill missing values with linear interpolation and optionally smooth the resulting time series.

16. `theme_WAC( )`: An optional theme available for use with "ggplot2" plots.

17. `XformLA( )`: The transformation from the "local" Earth-relative reference frame to the aircraft or body reference frame, or the reverse. Used extensively in the Kalman processor and other GPS-updating functions.

Chapter 4

More Details About Plotting

This chapter includes additional details about the plot functions available in Ranadu.

4.1 Pipes

When constructing plots, the use of "pipes" makes the logic clear and is recommended, so that is described first. All the code sequences described here can be implemented by saving the result from each step and then providing it to the next step, but pipes support the transmission of the result of a calculation to the next stage in the calculation without the need for intermediate storage. They are supported using the "%>%" argument, which is enabled by the "magrittr" package for R. Perhaps the strongest argument for using pipes is that they make the logic of plot construction clear. You start with a data.frame, optionally construct new variables, make appropriate selection of variables and the time interval, apply filters to accept only data meeting particular tests, and then construct the plot using the resulting tailored data.frame. Here is an example, where the data.frame is piped to "select()" (part of the dplyr package that passes on only the listed variables) and where the result is then piped to "Ranadu::selectTime()" where only the specified time range is transmitted forward. The result is finally piped to Ranadu::plotWAC(), where the first argument is a data.frame. That is supplied by the pipe. The result is shown in Fig. 4.1. Alternately, ggplotWAC() could be used to produce a similar result. In addition to showing the explicit steps in the processing chain, code like this ensures that the plot will be constructed the same way if the code is re-used or moved.

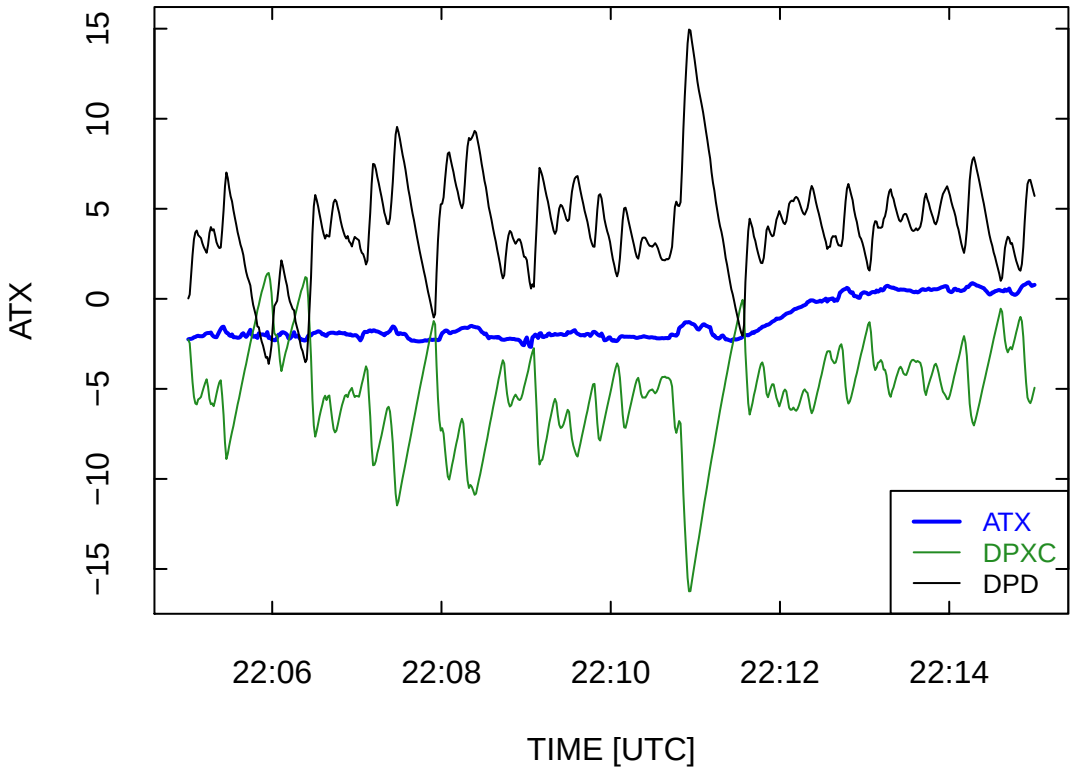

Figure 4.1: Example of a figure generated using pipes. The variables are air temperature (ATX), dew point temperature (DPXC), and a new generated variable representing the dew point depression (DPD=ATX-DPXC). From WECAN research flight 6, 3 August 2018.

Generating R code:

```
library(magrittr)
Ranadu::getNetCDF(fname, Variables) %>%     ## load the data.frame
  dplyr::filter(TASX > 90) %>%              ## limit based on airspeed
  dplyr::select(Time, ATX, DPXC) %>%        ## select the variables to plot
  Ranadu::Rmutate(DPD = ATX - DPXC) %>%       ## add the dewpoint-depression DPD
  Ranadu::selectTime(220500, 221500) %>%    ## set the time range
  Ranadu::plotWAC(col=c('blue', 'forestgreen', 'black'))  ## construct the plot
```

More information on some of the utility functions used or available when constructing plots is provided in the following list:

1. `dplyr::filter()`: This function is used to limit the range of accepted values. The arguments are a data.frame (provided above by the pipe) and a logical statement. Only rows for which the specified test is true are included in the resulting data.frame. An example where a statement like this might be useful is when fitting to determine the sensitivity coefficients for angle of attack, because it is useful to exclude slow flight when the gear and/or flaps might be deployed. Be sure to use the version from dplyr; the filter functions from the packages "stats" or "signal" have different behavior. An alternative method of creating a subset is to use the notation "Data[Data$TASX > 90,]". The disadvantage of this method and of the "select(Data, Data$TASX > 90)" function provided by base-R is that variable attributes are lost.[1]

2. `dplyr::select()`: This function creates a subset data.frame with only the desired variables. The desired list of names can be specified either as character names (with quotes) or variable names (without quotes). This also has the advantage over the "[]" or "[[]]" methods of subsetting that attributes of the data.frame and the variables are preserved.

3. `Ranadu::Rmutate( )`: This function adds new variables to the data.frame according to formulas specified in the second argument. In this processing chain, the first argument is the data.frame provided by the pipe. This calls the routine `dplyr::mutate()` but then, because that function does not preserve variable attributes, it transfers attributes from the input to the output data.frame. New variables, however, have no attributes (even the "Dimension" attribute) so the resulting data.frame has some limitations, notably not being accepted by "makeNetCDF( )".

4. `Ranadu::selectTime( )`: This function limits the time range of the resulting data.frame to be between the times that are specified in HHMMSS format (hours, minutes, seconds). This is equivalent to using "dplyr::filter( )" with limits on the accepted times, but it avoids the need to provide those times in the POSIXct format used by Ranadu data.frames. It preserves attributes and is suitable for use in pipes.

5. `Ranadu::Rsubset( )`: This is not used in the present example but could be. It accepts start and end times like "selectTime", selects variables like "dplyr::select", and imposes limitations on the data like "dplyr::filter( )", so several functions could be combined in one step: Ranadu::getNetCDF(fname, Variables) %>% Ranadu::Rsubset(220500, 221500, c('ATX', 'DPXC')) %>% plotWAC( ). This function also preserves attributes in the modified data.frame.

"Ranadu::plotWAC( )" is designed primarily for time-series plots, but scatterplots can also be generated. In that case, the first two variables in the data.frame should be the variables for the

[1] Preserving attributes is desirable because the attributes are often used by Ranadu routines. The bin assignments for size distributions are carried in an attribute, as is the data rate (used in spectral analysis). Variables with the same "short_name" attributes are redundant measurements of the same quantity, so it is often useful to plot all with matching short_names together. Finally, the function makeNetCDF( ) makes a new netCDF file with the variables and attributes in the data.frame, and it requires some attributes (like the "Dimension" attribute) to function.

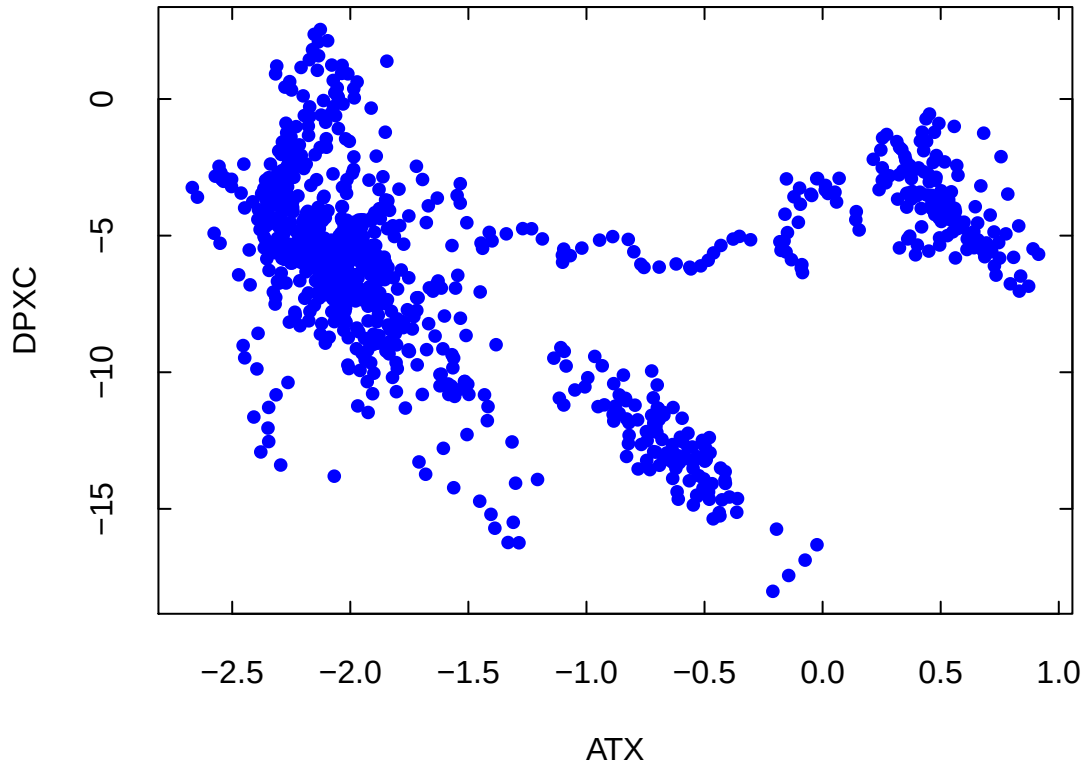

Figure 4.2: Example scatterplot.

scatterplot, not the Time variable, and an explicit label "xlab=xxx" should be supplied. Here is an example:

Generating R code:

```
Data %>% selectTime(220000, 221500) %>%
         dplyr::select(ATX, DPXC) %>%
         plotWAC(xlab='ATX', type='p')
```

4.2 Multi-Frame Plots

It is often desirable to combine several plots into a single plot. There are several ways to do this with Ranadu routines:

4.2.1 Multiple plotWAC( ) plots:

The "layout()" function in base-R can be used. Here is an example. The matrix layout can also be used to display plots in multiple columns or in multiple rows and columns.

Generating R code:

```
layout(matrix(1:2, ncol=1), widths=c(8,8), heights=c(5.5,8))
op <- par (mar=c(2,4,1,1)+0.1, oma=c(1.1,0,0,0))
Data %>% dplyr::select(Time, ATX) %>%
         plotWAC(ylab=expression(paste("T [", degree, "C]")))
op <- par (mar=c(5,4,1,1)+0.1)
Data %>% dplyr::select(Time, WIC) %>%
         plotWAC(ylab=expression(paste('W [m/s]')))
```

4.2.2 Panels and Facets in ggplotWAC( ):

The function `Ranadu::ggplotWAC( )` is based on the ggplot2 package for R, which provides extensive plotting capabilities and is highly recommended. What is provided via ggplotWAC( ) is a very simplified and restricted approach, but it might be useful in preliminary applications. See this URL for information on ggplot2. The Ranadu routine provides two approaches to multiple plots:

- "facets": If the argument "panels" is supplied (e.g., panels=N), ggplotWAC( ) will construct N vertically aligned panels. All will contain time-series plots. The first-argument to ggplotWAC(), the data.frame, should contain the variable "Time" and N*M variables, where the first M variables will be plotted in the first panel, the next M in the second panel, etc. A set of M character-mode labels should be supplied via labelN.
- "viewports": An optional argument "position=c(i, j)" can be used to place a plot in the ith of j vertically aligned viewports.

Examples are shown in Figs. 4.4 and 4.5. An advantage of the faceted plot is that vertical alignment of the plots is ensured; this can be a problem with other plots if the axis labels are of different size in the different plots. In the second case, the viewports are positioned so that the abscissa labels and title are obscured for the top plot. The intent is that this should be used for identical time scales for each plot, so that it is not necessary to duplicate the axis labels and title.

Generating R code:

```
Project <- 'WECAN'    ## faceted plot with ggplotWAC()
Flight <- 5
```

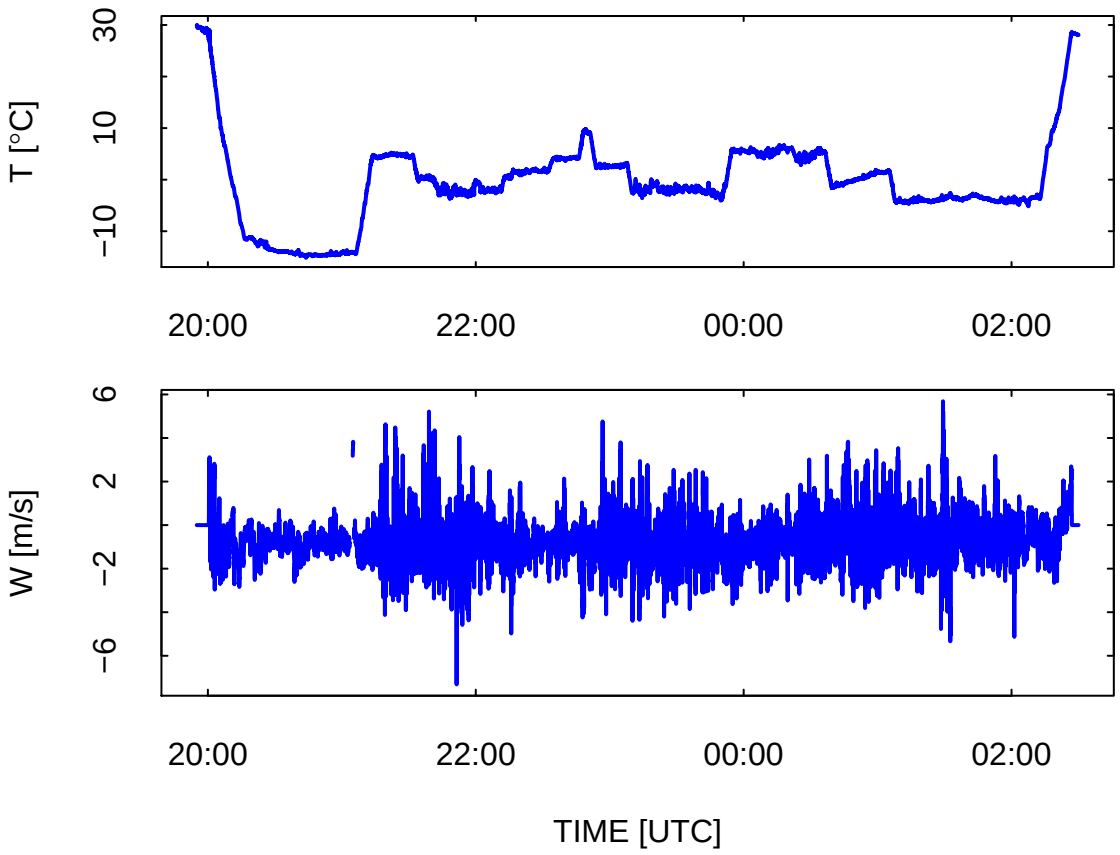

Figure 4.3: Two-panel figure.

```
V <- c('ATH1', 'ATH2', 'ATF1', 'RTH1', 'RTH2', 'RTF1')
fname <- sprintf('%s%s/%srf%02d.nc', DataDirectory(), Project, Project,
                 Flight)
getNetCDF(fname, V, 200000, 201500) %>%
    ggplotWAC(panels=2,
              col=c('blue', 'darkorange', 'forestgreen'),
              ylab=expression(paste('temperature [', degree,'C]')),
              lwd=c(1.5, 0.8, 1), lty=c(1,2,1),
              labelP=c('   air temperature', ' recovery temperature'),
              labelL=c('H1', 'H2', 'F1'),
              legend.position=c(0.5,0.95)
)
```

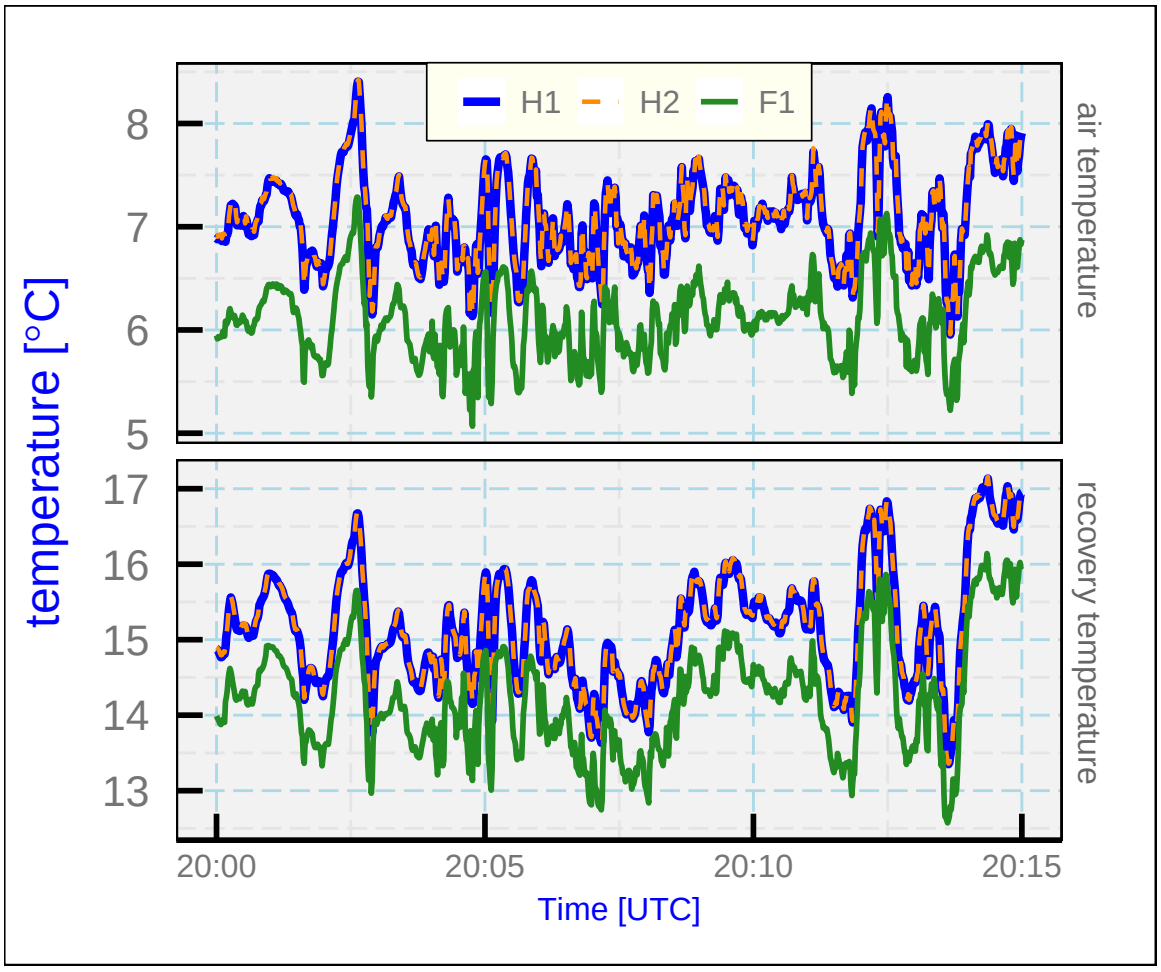

Figure 4.4: Example of a faceted ggplotWAC plot. Three temperature measurements are shown from probes identified as H1, H2, and F2. The top panel shows the air temperature, and the bottom panel the directly measured recovery temperature before correction for dynamic heating.

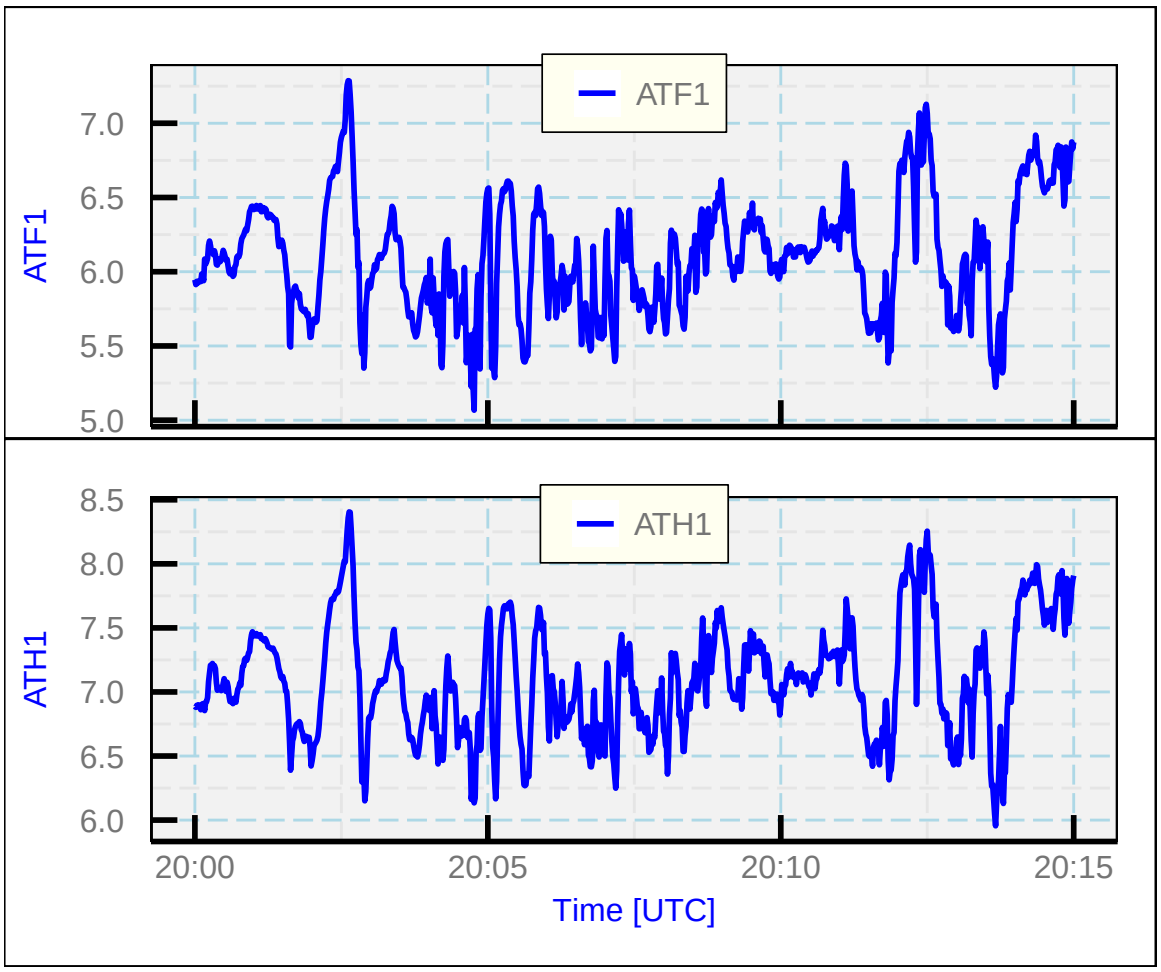

Figure 4.5: An example of selecting the figure position using viewports and the "position" argument to ggplotWAC().

Generating R code:

```
## viewport-plot with ggplotWAC()
DG <- getNetCDF(fname, V, 200000, 201500)
with(DG, ggplotWAC(data.frame(Time, ATH1), position=c(1,2)))
with(DG, ggplotWAC(data.frame(Time, ATF1), position=c(2,2)))
```

4.3 Plotting the density of events

A scatterplot like that in Fig. 4.2 often is used to show a two-dimensional display of where events occur. Such plots are useful when the number of events is small, but for large numbers of events the overlap of points can obscure relationships. The base-R function "smoothScatter()" is one option for plotting the density of points in such cases. Another is the "filled.contour()" function. Ranadu provides a third option in the function Ranadu::contourPlot(). The number of bins, colors, and linear vs. logarithmic density intervals can be provided as arguments to this function, although the defaults often work acceptably. The following figure and code illustrates the use of this plot. Additional more elegant solutions are provided by the ggplot2 package.

Generating R code:

```
getNetCDF(fname) %>% dplyr::select(ATX, DPXC) %>%
    contourPlot(title='WECAN flight #5')
```

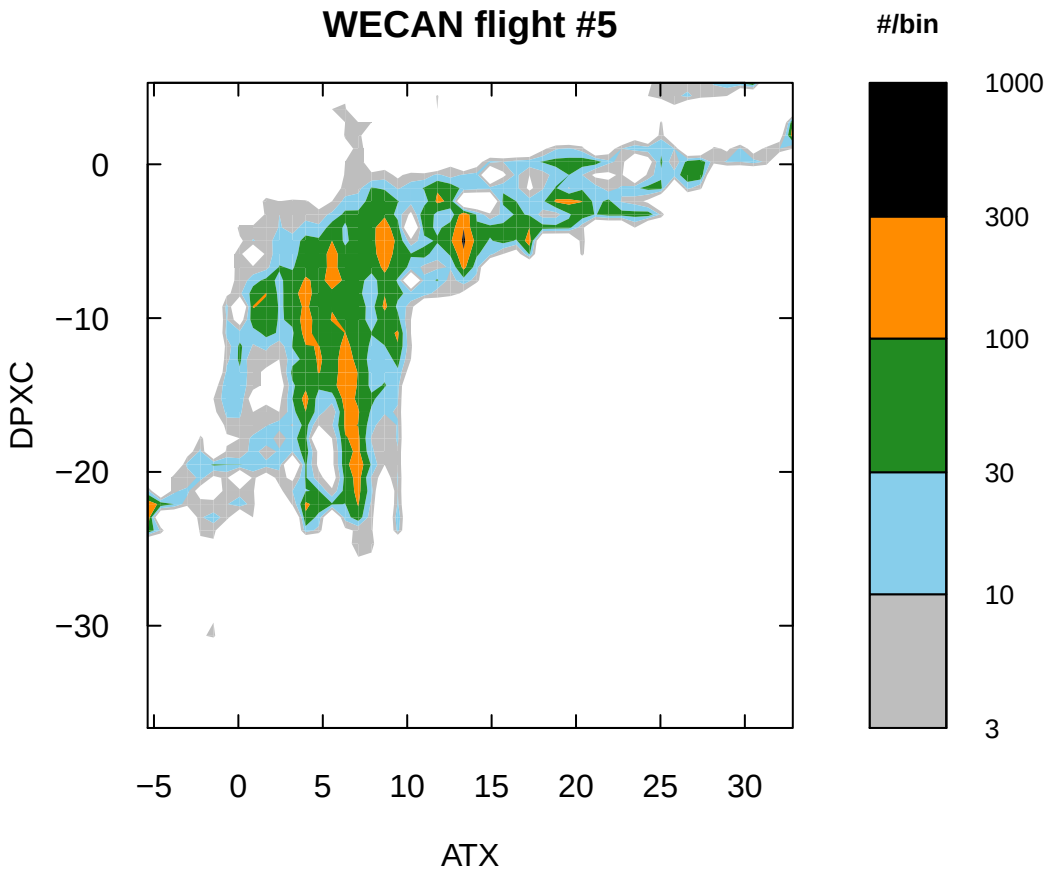

Figure 4.6: Example of a density plot generated using Ranadu::contourPlot().

4.4 Error bars and box-and-whisker plots

Ranadu provides the "binStats()" function to compile variable characteristics needed to generate plots like error-bar plots and box-and-whisker plots. The input should be a data.frame whose first two columns specify the expected respective ordinate and abscissa variables. Each row in the data.frame is assigned to a bin on the basis of the value of the second variable, and for each bin the mean, standard deviation, and number of events are accumulated for values of the first variable in the data.frame. The output from binStats() is a new four-column data.frame where the respective columns are the mean value of the abscissa for each bin, the mean value of the ordinate for all events in the bin, and corresponding standard deviation, and the number of events in the bin.

Generating R code:

```
getNetCDF(fname) %>%
  Rmutate(DPD=ATX-DPXC) %>%       ## define dew-point-depression variable
  dplyr::select(DPD, GGALT) %>%
  binStats() %>%
  ggplot(aes(x=xc)) + geom_point(aes(y=ybar), color='blue') +
  geom_errorbar(aes(ymin=ybar-sigma, ymax=ybar+sigma)) +
  xlab('geometric altitude [m]') +
  ylab(expression(paste('dew point depression [', degree, 'C]'))) +
  theme_WAC()
```

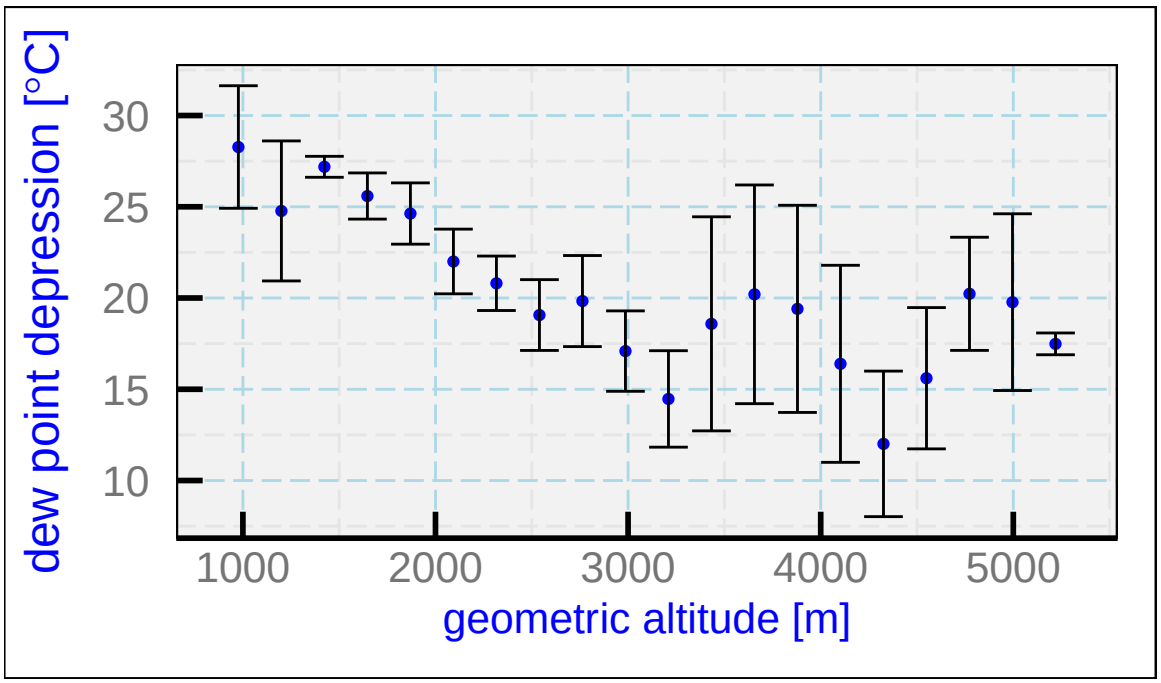

Figure 4.7: Example of an error-bar plot generated using binStats().

The following code creates Fig. 4.8, an example of creating a box-and-whisker plot. When used with the argument "addBin = TRUE", binStats instead returns a modified data.frame with a variable "BIN" added that is suitable to use when grouping in ggplot aesthetics.

Generating R code:

```
getNetCDF(fname) %>%
  dplyr::select(DPXC, GGALT) %>%
  binStats(addBin = TRUE) %>%
  ggplot() + geom_boxplot(aes(GGALT, DPXC, group=BIN),
                          color='blue', na.rm=TRUE) +
  theme_WAC()
```

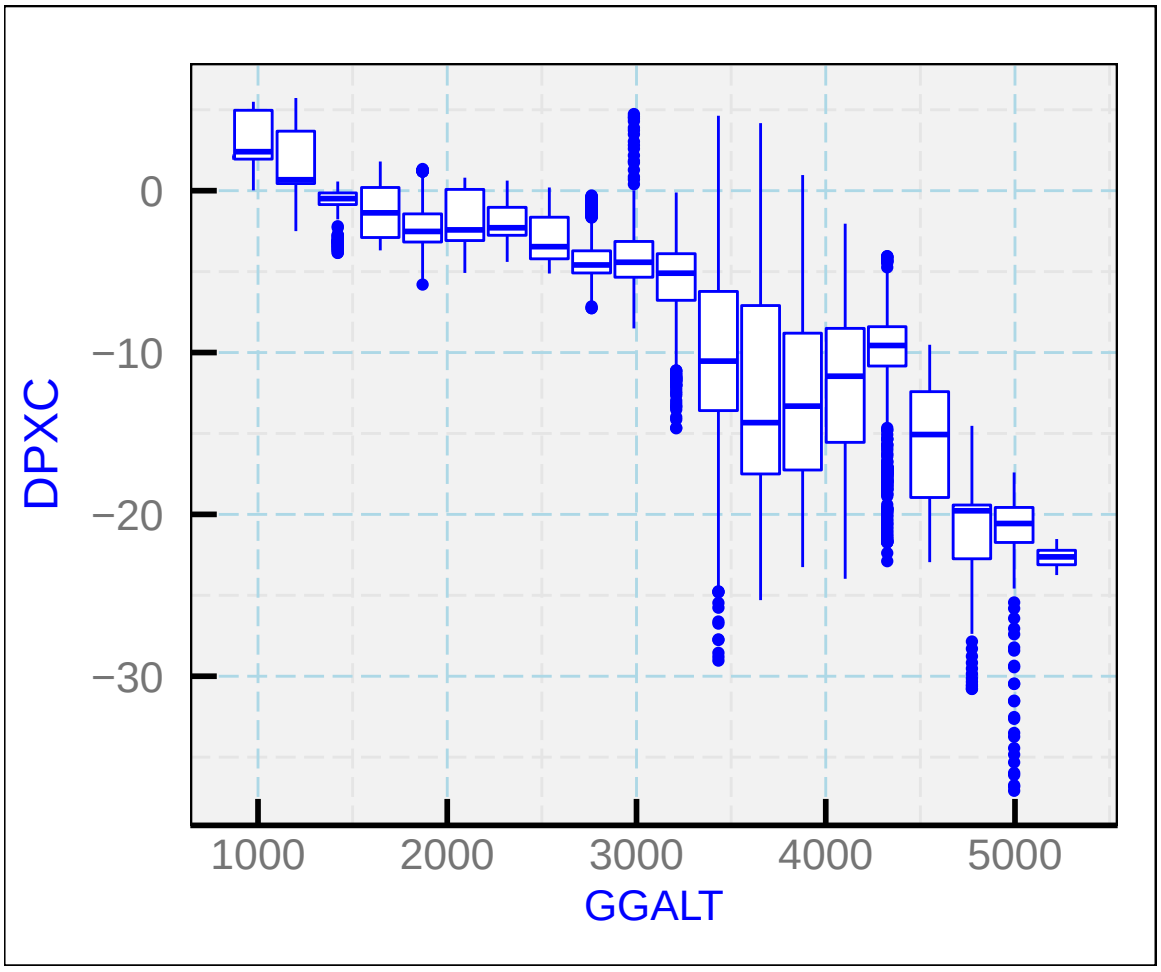

Figure 4.8: An example of a box-and-whisker plot.

4.5 The skew-T thermodynamic diagram

Ranadu incorporates the ability to plot a set of measurements on the background of a skew-T diagram. The background is non-standard and is described in detail in this document. The "`Ranadu::SkewTSounding()`" function should be called with a data.frame containing measurements of pressure, temperature and dewpoint, which may be named either ("PSXC", "ATX", "DPXC") or ("Pressure", "Temperature", "DewPoint"). A skew-T background is generated using the Ranadu data file "skewTDiagram.Rdata" and the values from the input data.frame are optionally averaged in pressure intervals and then plotted on this background. Figure 4.9 is an example of this plot.

Generating R code:

```
Project <- 'PREDICT'
Flight <- 11
fname <- sprintf ('%s%s/%srf%02d.nc', DataDirectory(), Project,
                  Project, Flight)
getNetCDF(fname) %>%
    selectTime(0, 150000) %>%
    SkewTSounding(AverageInterval=10)
```

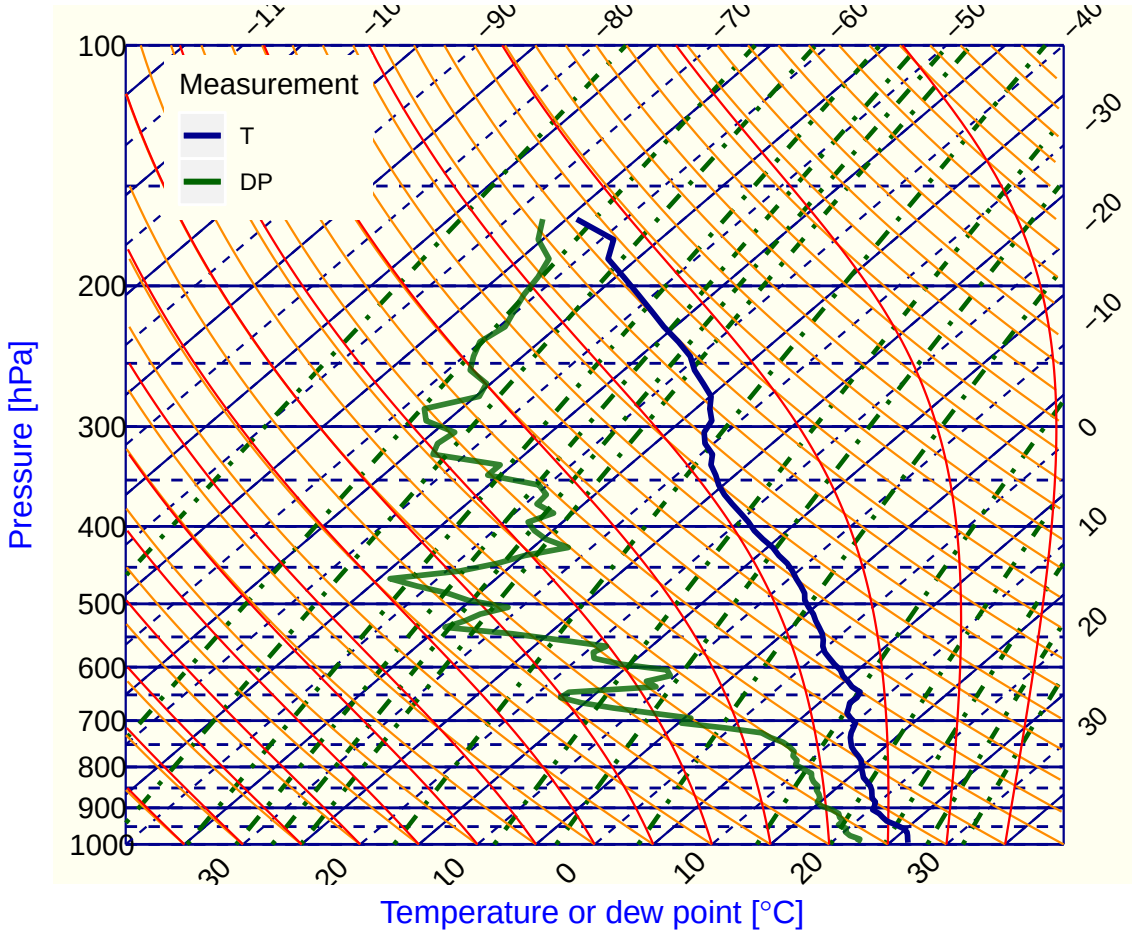

Figure 4.9: Sounding from PREDICT flight 11.

4.6 Aerosol- and hydrometeor-size distributions

The Ranadu function "plotSD()" displays the size distribution measured by various probes that produce arrays of measurements. The data.frame containing these measurements is special in that the column corresponding to a variable name like "CCDP_RPC" is a two-dimensional vector. This makes the data.frame inconsistent with the "tidy" structure and with the structure required for a "tibble", so some special considerations are required if an analyst wants to use only tidy data. In this section, those considerations are not discussed further because the "plotSD()" function assumes Ranadu-style data.frame conventions.

"Ranadu::getNetCDF()" accepts variable names like "CCDP_", in which case it searches for the first variable starting with that name. This avoids the need to know the location of the CDP probe in various projects. However, if there are multiple probes with the same prefix name, they need to be specified in the variable list used to construct the data.frame.

An example, Fig. 4.10, shows code that will construct a plot of the size distribution from several probes. The "exceedance" is added when the argument "CDF=TRUE" is used; it is the complement to a cumulative distribution and shows the fraction of particles that exceed the plotted size. The four numbers returned from the function are the mean concentration, mean diameter, standard deviation in the diameter, and liquid water content under the assumption that all particles are liquid. It is also possible to construct a plot of the distribution in liquid water content, as shown in Fig. 4.11. See "?Ranadu::plotSD" for more options including alternate specification of the size limits, bins to include, and log vs linear axes.

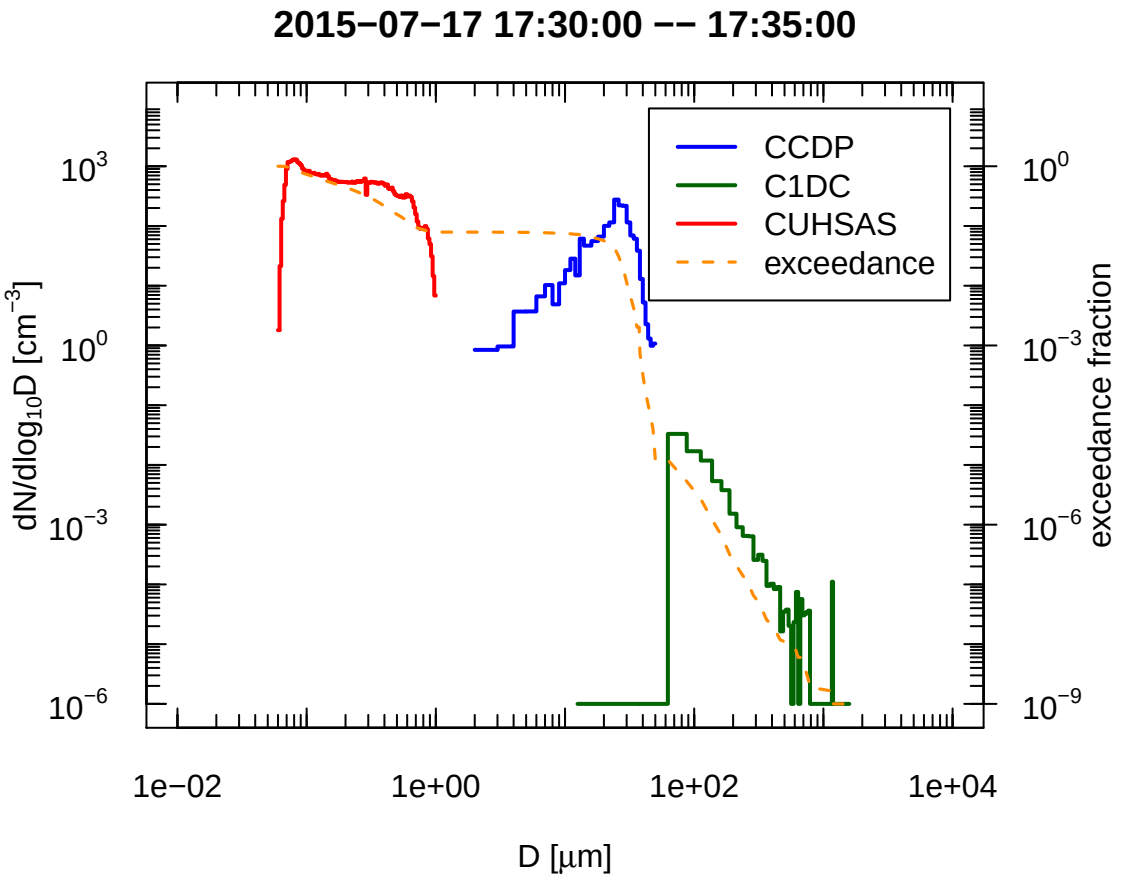

Figure 4.10: Example of a size distribution.

Generating R code:

```
# assignment to A just suppresses printing of the list returned from plotSD
A <- getNetCDF('/Data/CSET/CSETrf06.nc', c('CCDP_', 'C1DC_', 'CUHSAS_')) %>%
  selectTime(173000, 173500) %>%
  plotSD(CellLimits=NA, logAxis='xy', CDF=TRUE)
```

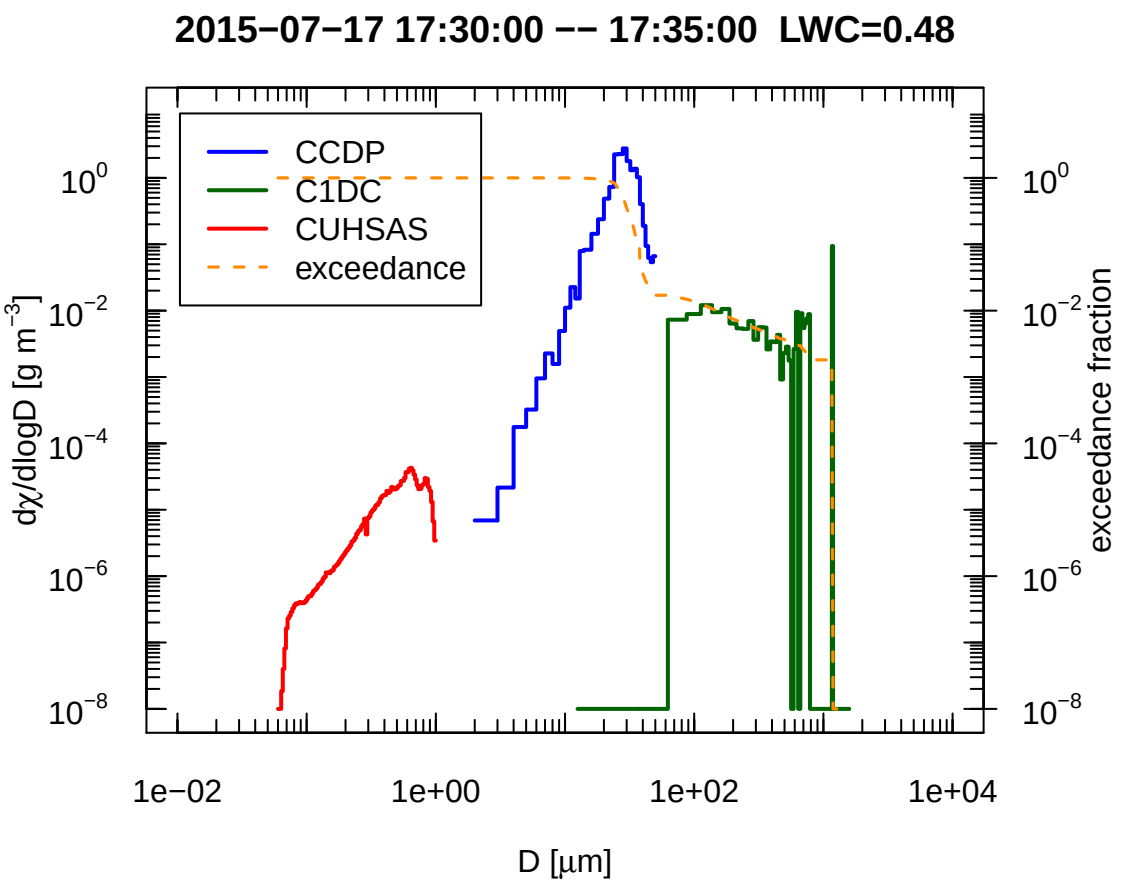

Figure 4.11: Example of a distribution in liquid water content.

Generating R code:

```
A <- getNetCDF('/Data/CSET/CSETrf06.nc', c('CCDP_', 'C1DC_', 'CUHSAS_')) %>%
  selectTime(173000, 173500) %>%
  plotSD(CellLimits=NA, logAxis='xy', LWC=TRUE, CDF=TRUE)
```

Chapter 5

Variance Spectra

Variance spectra are also plots and perhaps belong in the preceding chapter, but they are discussed here in greater detail than the preceding plots and so seemed to fit better in a separate chapter.

5.1 Background regarding spectral variance

5.1.1 Terminology

The plots discussed here as "variance spectra" are often referred to as "power spectra." That term is not used here because the spectra representing variance in the data.frame measurements from NCAR/EOL/RAF netCDF files are not power. Even in the case of wind (with variance dimensions m^2s^{-2}/Hz), the variance spectrum is better described as a kinetic-energy spectrum. For this reason, the plots discussed in this chapter will be called "variance spectra" and the plotted quantity will be called the spectral variance.

5.1.2 Transformations among spectra; "proper" spectra

Consider the cumulative distribution function for variance $C(\nu)$, the fraction of the variance that is contributed by frequencies smaller than ν. The differential distribution function with respect to frequency is then

$$P(\nu) = \frac{dC(\nu)}{d\nu} \quad .$$

Consider how this distribution function would change if defined in terms of a new variable $x(v)$ starting with the specific variable $x = \ln \nu$. The differential distribution function would then be

$T(x)$ specified by

$$T(x) = \frac{dC(x)}{dx} = \frac{dC(\nu)}{d\nu}\frac{d\nu}{dx} = \nu P(\nu)$$

$\nu P(\nu)$ thus gives the spectral density in terms of $\ln \nu$ and so in terms of $\ln(10)\log_{10}\nu \approx 2.30\log_{10}\nu$.

In the following, two options for variance spectra are emphasized: $P(\nu)$ vs ν on a linear plot and $\nu P(\nu)$ vs $\log_{10}\nu$ with a logarithmic abscissa and either a linear or logarithmic ordinate scale. These are regarded here as "proper" displays because the area under segments of the plotted curves represent contributions to the variance so it is possible to estimate the contributions to variance from various intervals in frequency by using the areas on the plot. This direct representation is compromised in the case where the variable $\nu P(\nu)$ is plotted on a logarithmic scale because then it is necessary to consider the logarithmic ordinate when evaluating areas. This minor inconvenience nevertheless is less significant than the problems that arise from using a linear ordinate scale, in which case the ordinate range obscures relationships and the common "$-5/3$" slope seen in logarithmic plots becomes a parabolic line that is difficult to interpret. For that reason, plots of spectral variance here will emphasis plots of $\nu P(\nu)$ vs $\log_{10}(\nu)$ on log-log scales. It is suggested that plots of $P(\nu)$ vs ν on log-log scales should be avoided because the connection between area on the plot and variance is lost, making the plot harder to interpret. In addition, "$-5/3$" spectra are steep, the range of ordinate values is higher, and the plots are therefore more difficult to interpret than those plotting $\nu P(\nu)$ vs ν on a log-log scale.

5.1.3 Methods used to estimate spectral variance

The function "Ranadu::VSpec()" includes three methods that can be selected to estimate the spectral variance:

1. The function "spectrum()" from the "stats" R package, which estimates a periodogram using Fourier-series estimation and then optionally smooths the spectrum using modified Daniell smoothers. The amount of smoothing can be controlled with the "spans" parameter in the function call.

2. The "Welch" method as implemented by the R package "bspec". In this method, the time series is divided into subsets and the spectra resulting from the subsets are averaged to reduce the variance in the result. The degree of smoothing is controlled by the "segLength" parameter that specifies the length of the segments to use.

3. The "maximum entropy" method in which the spectrum is represented by "poles" and evaluated from the contributions of those poles across the frequency spectrum. A higher number of poles and a smaller value of the "resolution" will lead to more structure in the result.

In addition, it is possible to smooth the resulting spectrum further by specifying a value for the parameter "smoothBins", in which case the frequency range will be partitioned into the specified number of logarithmically spaced bins and the values of the spectral density will be averaged in each bin.

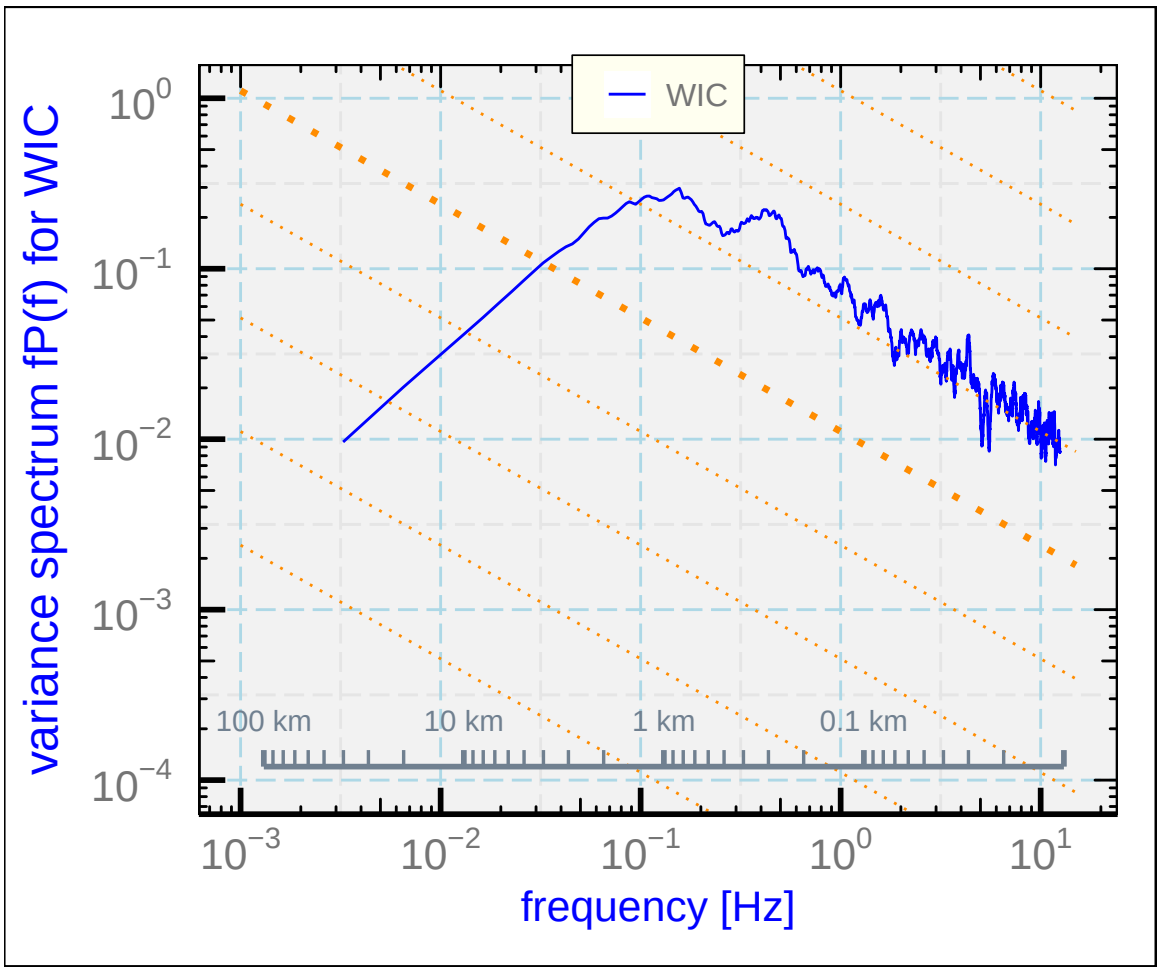

Figure 5.1: Variance spectrum for measurements of vertical wind during SOCRATES flight 8, 4:56:00 – 5:01:00.

5.2 Generating plots of variance spectra

See "?Ranadu::VSpec" for details regarding using this function. The essential inputs are a data.frame that includes at least the variables "Time", "TASX", and the variable for which the variance spectrum is desired. TASX is needed to interpret the scale both in terms of frequency and wavelength. An example is shown in Fig. 5.1, using the default specifications.

Generating R code:

```
getNetCDF('/Data/SOCRATES/SOCRATESrf08h.nc', Start=45600, End=50100) %>%
  VSpec('WIC') + theme_WAC()
```

The following demonstrates how to combine plotted spectra. The three lines on this plot were generated using the three methods of spectral estimation available in VSpec(), all for the WIC variable:

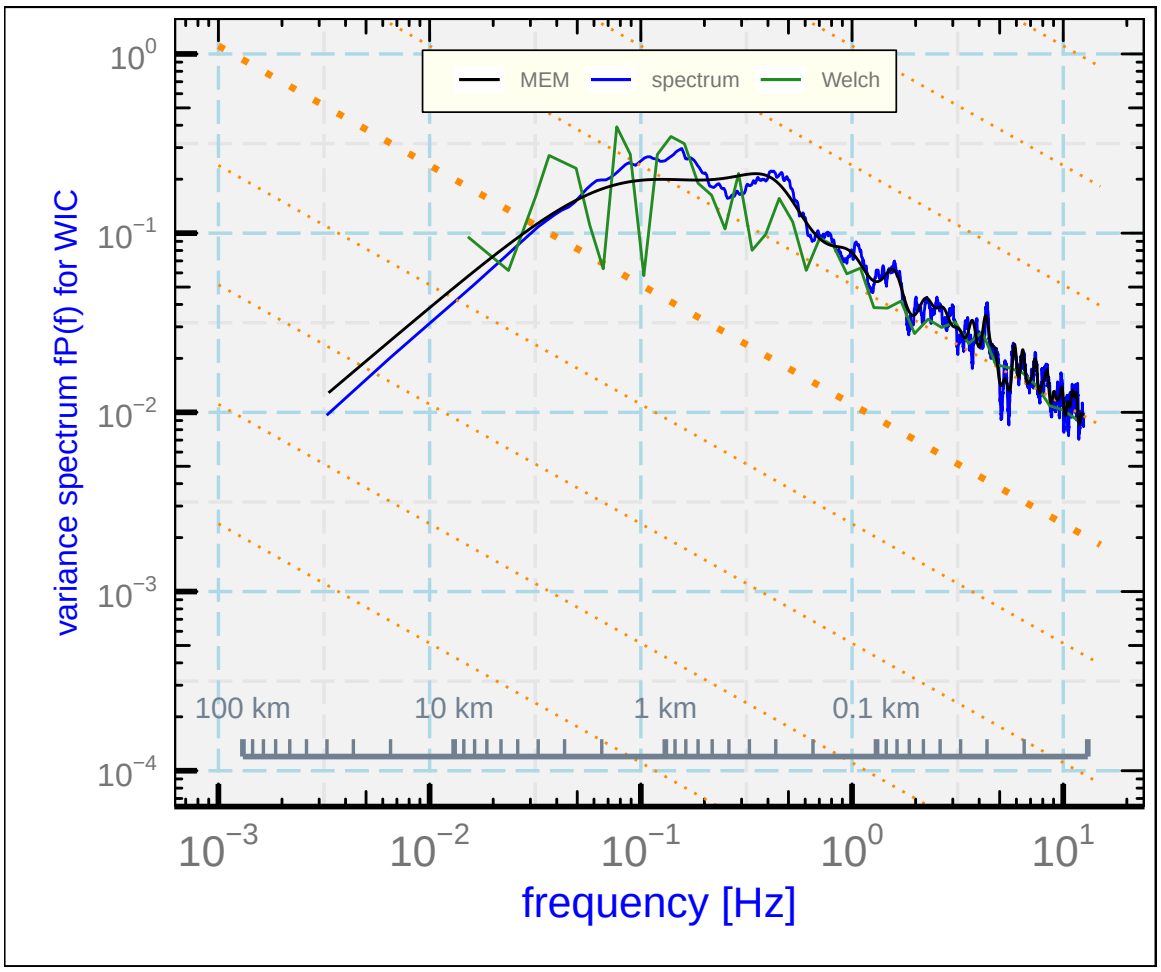

Figure 5.2: Variance spectra for the same data shown in the preceding plot but generated by the three methods indicated in the legend.

Generating R code:

```
D <- getNetCDF('/Data/SOCRATES/SOCRATESrf08h.nc', Start=45600, End=50100) %>%
     Rmutate(WIC2=WIC, WIC3=WIC) ## duplicate the variable
g <- VSpec(D, 'WIC', VLabel='spectrum')
g <- VSpec(D, 'WIC2', method='Welch', VLabel='Welch',
           segLength=128, smoothBins=50, add=g)
VSpec(D, 'WIC3', method='MEM', VLabel='MEM', add=g) +
  theme_WAC(1)
```

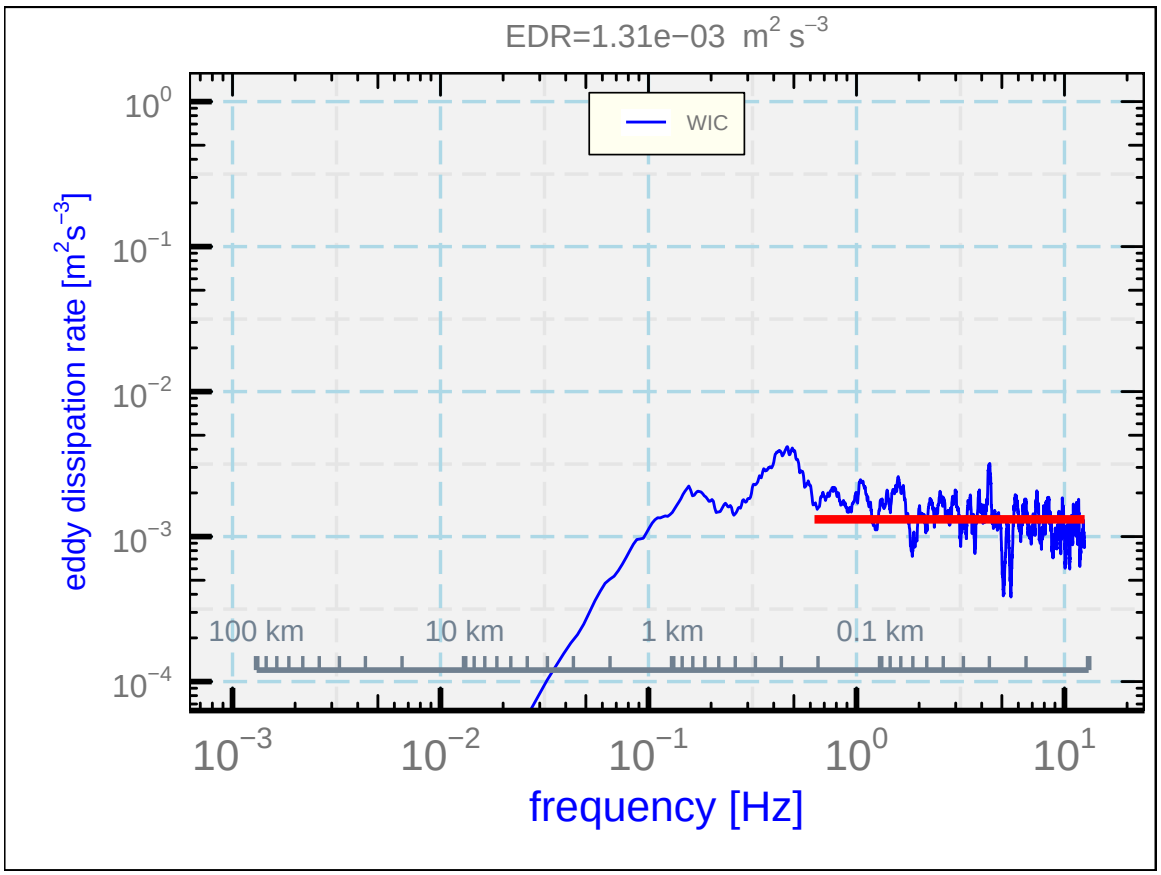

Figure 5.3: An eddy-dissipation-rate plot for the same data shown in the preceding plot.

Another option that may be of use, although the result is not a "proper" spectrum in the sense used above, is to plot with weighting by $\nu^{5/3}$ and additional change of variables so that the resulting ordinate matches the eddy dissipation rate in a case where the measurements are indeed from an inertial subrange. Figure 5.3 illustrates this plot. The variable plotted is $(2\pi/V)(CP(\nu)\nu^{5/3})^{3/2}$, with V the airspeed and $C = 1.5$ for lateral spectra; this quantity should equal the eddy dissipation rate in an inertial subrange.

Generating R code:

```
VSpec(D, 'WIC', EDR=TRUE) + theme_WAC(1)   ## theme_WAC(1) => smaller title
```

Generating R code:

```
## to suppress the title, add "+ ggtitle(' ')"
```

Chapter 6

Miscellaneous

6.1 Combining flights

It is sometimes useful to have a data.frame that spans a whole project. Individual data.frames can be combined using the R function "rbind", provided the individual data.frames have the same structure. The argument "F" to "getNetCDF()" can be used to add a variable named "RF" with the value specified by "F", so that individual flights can be identified and easily separated in the combined data.frame.

Here are some examples that illustrate uses of the combined data set:

Generating R code:

```
VarList <- c("ADIFR", "PITCH", "QCF", "PSF", "AKRD", "WIC",
  "TASF", "GGALT", "ROLL", "PSXC", "ATX", "DPXC", "QCXC",
  "EWX", "ACINS","GGLAT")
## add variables needed to recalculate wind
VarList <- c(VarList, "TASX", "ATTACK", "SSLIP",
  "GGVEW", "GGVNS", "VEW", "VNS", "THDG")
Data <- data.frame()
Project <- 'CSET'
Fl <- sort (list.files ( ## get list of available flights
  sprintf ("%s%s/", DataDirectory(), Project),
  sprintf ("%srf...nc$", Project)))
for (flt in Fl) {
    fname = sprintf("%s%s/%s", DataDirectory(), Project, flt)
    fno <- as.numeric(sub('.*f([0-9]*).nc', '\\1', flt))
    D <- getNetCDF (fname, VarList, F=fno)
    Data <- rbind(Data, D)
}
```

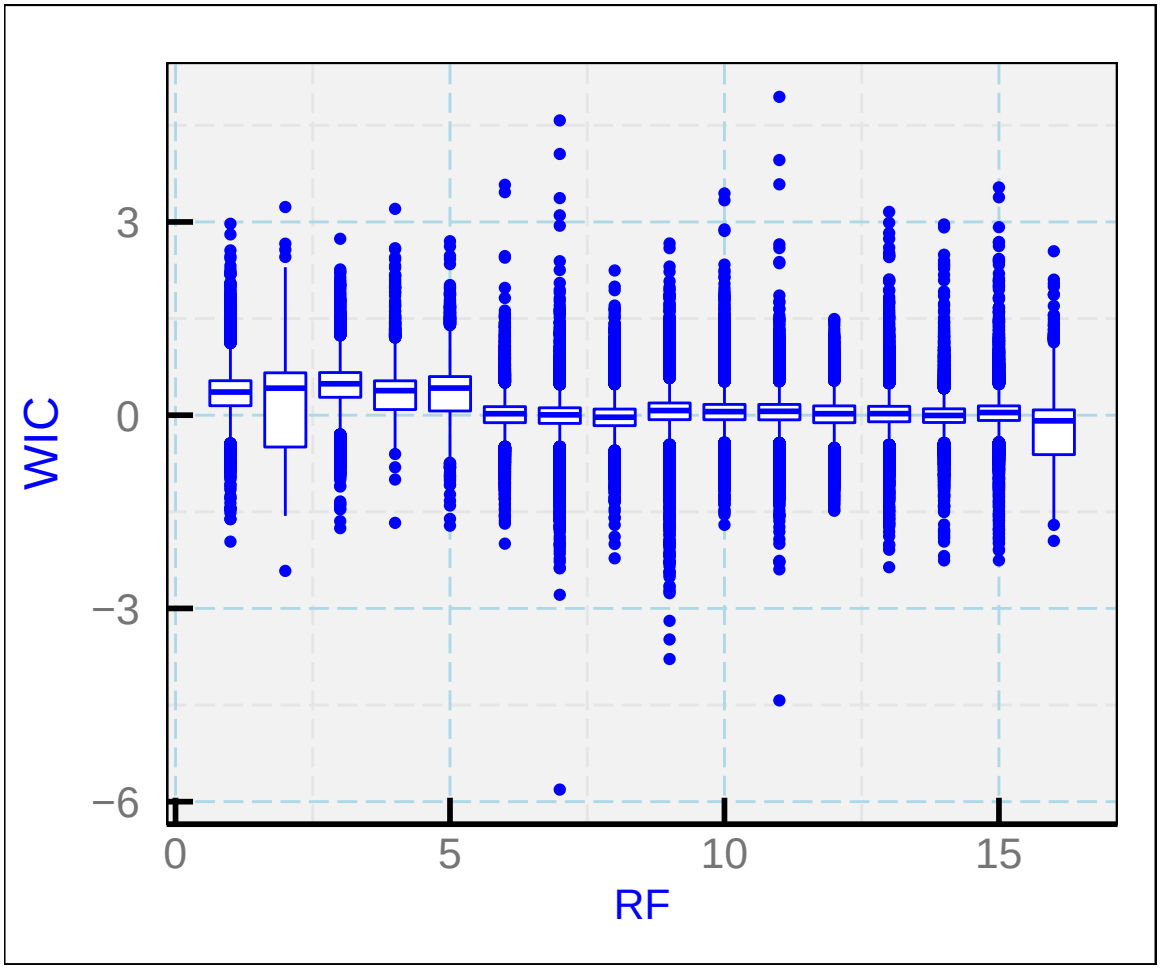

Figure 6.1: Distribution of values of the vertical wind for each research flight number.

```
## restrict to where good vertical wind is expected
dplyr::filter(Data, TASX > 90, abs(ROLL) < 2) %>%
    dplyr::select(Time, WIC, ATX, DPXC, EWX, GGALT, RF) %>%
    ggplot() +
    geom_boxplot(aes(RF, WIC, group=RF),
                 color='blue', na.rm=TRUE) +
    theme_WAC()
```

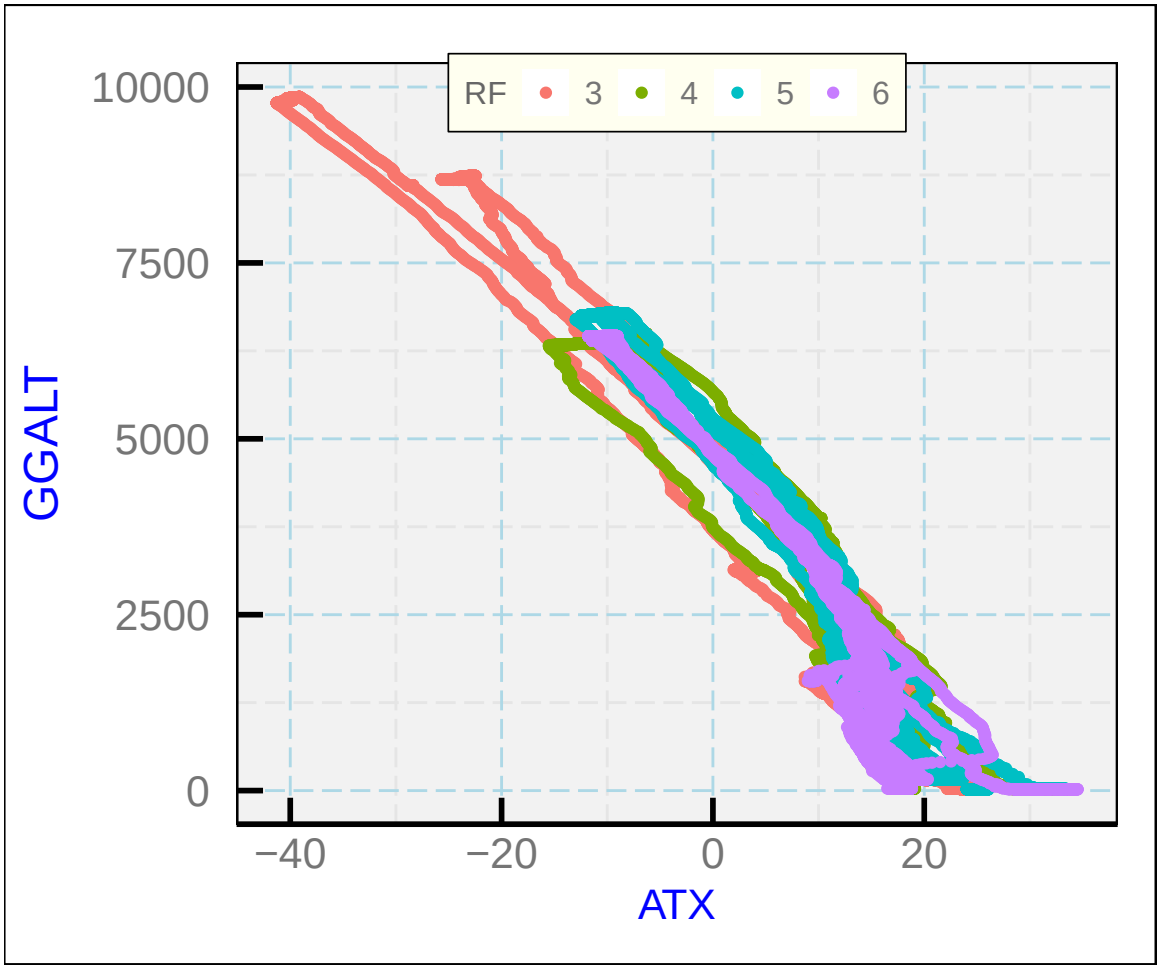

Figure 6.2: Measurements of temperature vs. altitude during research flights 3 to 6.

Generating R code:

```
Data %>% dplyr::select(ATX, GGALT, RF) %>% dplyr::filter(RF >= 3 & RF <= 6) %>%
  Rmutate(RF = as.character(RF)) %>%
  ggplot() + geom_point(aes(ATX, GGALT, color=RF), na.rm=TRUE) +
  theme_WAC()
```

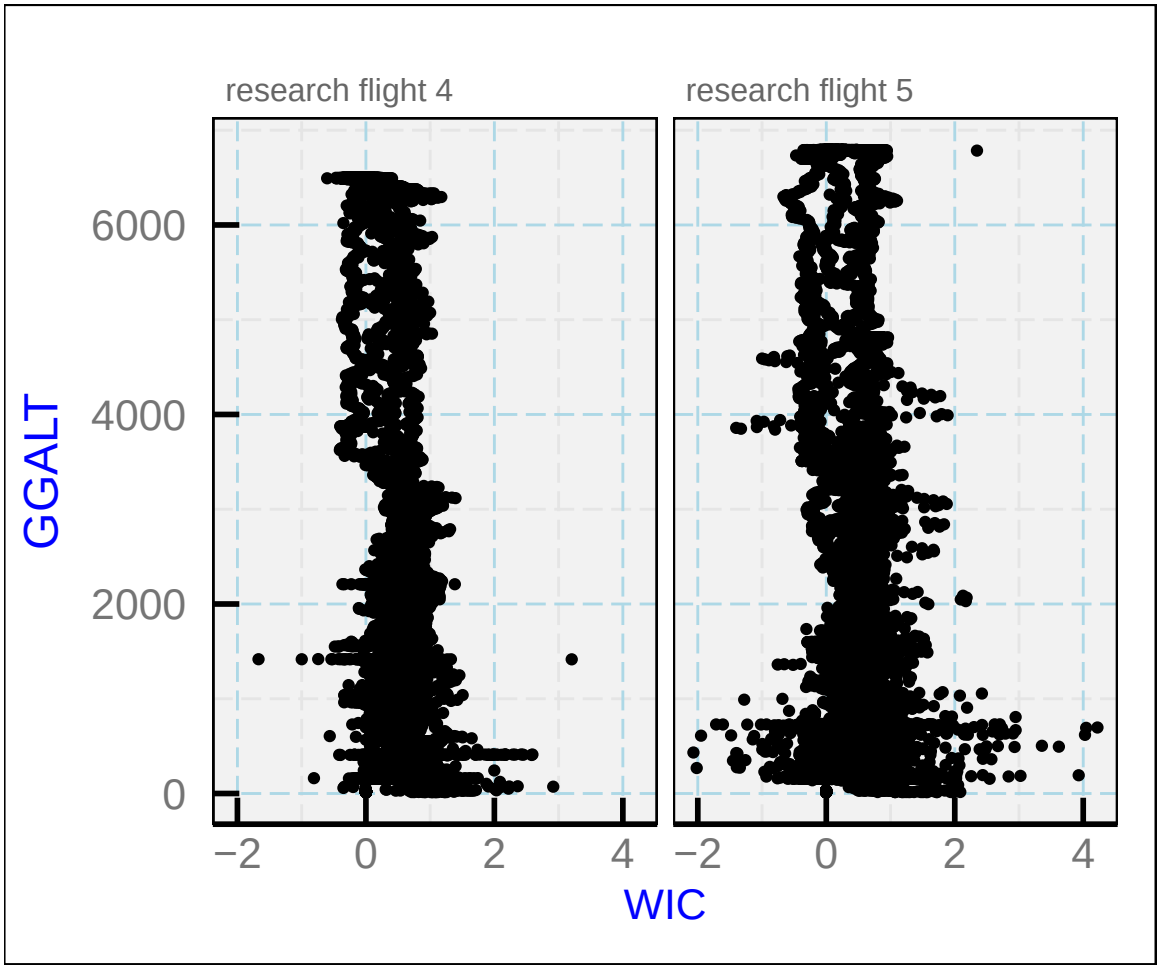

Figure 6.3: Example that uses faceted plots to show results from different research flights.

Generating R code:

```
Data %>% dplyr::select(WIC, GGALT, RF) %>%
         dplyr::filter(RF == 4 | RF == 5) %>%
         Rmutate(RF = sprintf('research flight %d', RF)) %>%
         ggplot() + geom_point(aes(WIC, GGALT), na.rm=TRUE) +
         facet_wrap(~ RF, nrow=1) + ## see also facet_grid()
         theme_WAC()
```

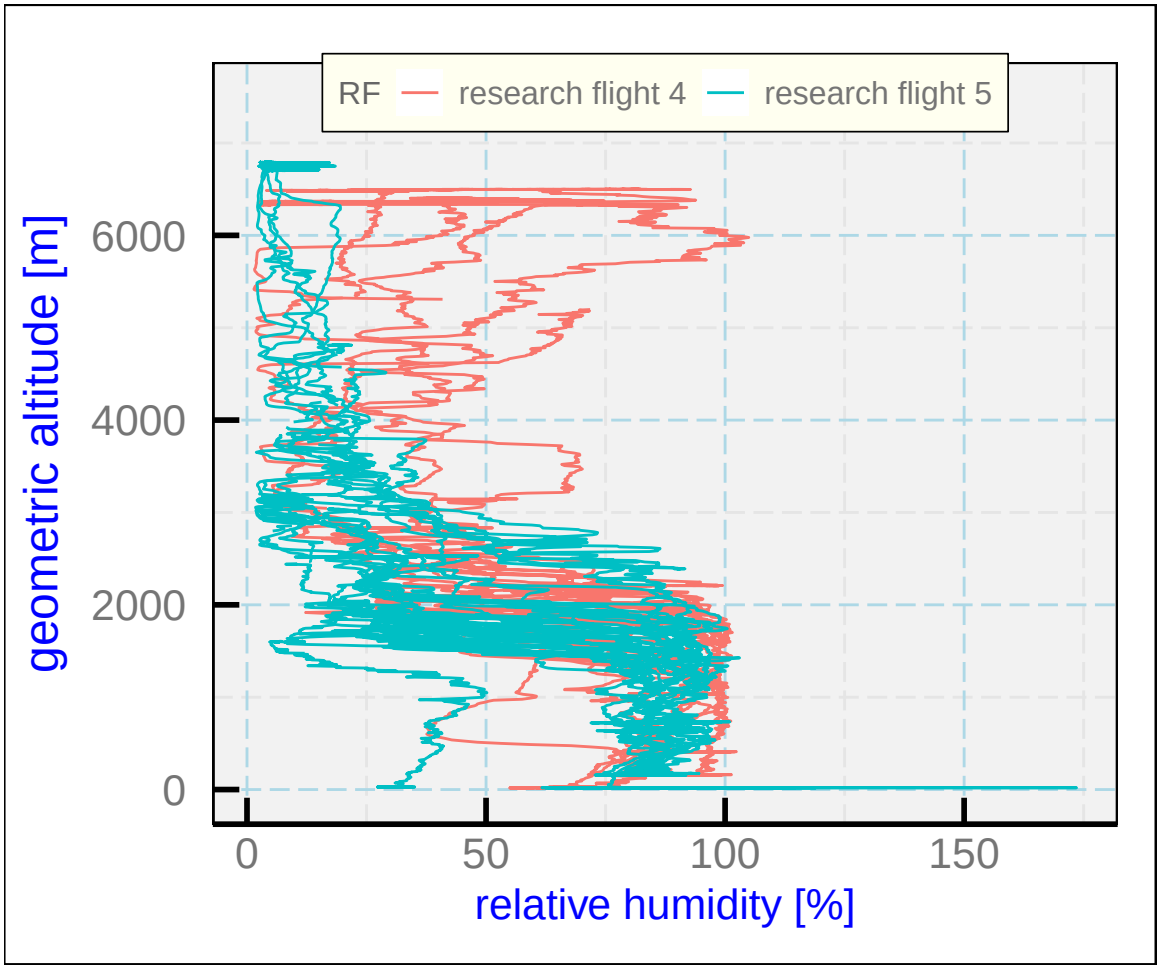

Figure 6.4: Example that uses faceted plots to show results from different research flights.

Generating R code:

```
Data %>% dplyr::select(EWX, ATX, GGALT, RF) %>%
         dplyr::filter(RF == 4 | RF == 5) %>%
         Rmutate(RF = sprintf('research flight %d', RF)) %>%
         Rmutate(RH = 100 * EWX / MurphyKoop(ATX)) %>%  ## new variable
         ggplot() + geom_path(aes(RH, GGALT, color=RF), na.rm=TRUE) +
         ylim(c(0, 7500)) +
         xlab('relative humidity [%]') +
         ylab('geometric altitude [m]') +
         theme_WAC()
```

Generating R code:

```
Data %>% dplyr::group_by(RF) %>% summarise(mean = mean(WIC, na.rm=TRUE))
```

```
## # A tibble: 16 x 2
##       RF     mean
##    <dbl>    <dbl>
##  1     1  0.329
##  2     2  0.150
##  3     3  0.419
##  4     4  0.311
##  5     5  0.326
##  6     6  0.00121
##  7     7 -0.0337
##  8     8 -0.0271
##  9     9  0.0543
## 10    10  0.0533
## 11    11  0.0468
## 12    12  0.00970
## 13    13  0.0220
## 14    14 -0.00305
## 15    15  0.0375
## 16    16 -0.205
```

6.2 Comments re "tibbles"

The data.frames used by convention in Ranadu are inconsistent with the "tidy" structure discussed in "R for Data Analysis" by H. Wickham because, for size-distribution variables such as those produced by the CDP or UHSAS the column consists of a two-dimensional vector where the first dimension is the row and the second is the concentration or count of particles in each bin. Data.frames not containing such variables are "tidy" and can be converted to tibbles using the function as.tibble(). This will fail, however, for data.frames that contain size-distribution variables. The function Ranadu::df2tibble() will convert such data.frames to tibbles by converting the two-dimensional vectors into lists. However, then the tibbles won't work with functions like Ranadu::plotSD(). Otherwise, the resulting tibbles are consistent with the Ranadu functions including plotting and algorithm calculations.

6.3 Using Ranadu from Python

The interface between Python and R is "rpy2", which is available only for Python3. To install rpy2 and prepare for its use in Python, use commands like these after installing Ranadu in your local R installation:[1]

```
sudo python3 -m pip install rpy2
# optional...
# from numpy import *
# import scipy as sp
# from pandas import *
# needed:
from rpy2.robjects.packages import importr
import rpy2.robjects as r
from rpy2.robjects import pandas2ri
pandas2ri.activate()
from rpy2 import robjects
from rpy2.robjects import Formula, Environment
from rpy2.robjects.vectors import IntVector, FloatVector
from rpy2.robjects.lib import grid
from rpy2.robjects.packages import importr, data
from rpy2.rinterface import RRuntimeError
import warnings
# The R 'print' function
rprint = robjects.globalenv.get('print')
stats = importr('stats')
grdevices = importr('grDevices')
base = importr('base')
datasets = importr('datasets')
grid.activate()
import math, datetime
import rpy2.robjects.lib.ggplot2 as ggplot2
Ranadu = importr('Ranadu')
magrittr = importr('magrittr') # this is required to use ``%>%'' pipes
```

You can then import data.frames generated by Ranadu::getNetCDF() using Python commands like the following:

```
Data = robjects.r(``"'
```

[1] In my case compilation initially failed because "readline.h" wasn't found. This is likely a defect of my configuration, but if that occurs for you try copying "readline.h" from whereever it exists (e.g., "~/miniconda2/include/readline" to somewhere in your include path like "/usr/include/python3.7m".

```
                    # get data.frame from R
                    Project = 'WECAN'
                    Flight = 1
                    filename = sprintf('%s%s/%srf%02d.nc',
                        DataDirectory(), Project, Project, Flight)
                    Data = getNetCDF(filename)
                    ‘“”’)
Data.head()  # use this to see the structure of Data
```

You can now reference the columns in "Data" using the syntax "Data['Time']", "Data['ATX']", etc. For example, to plot the time series of the measurement ATX use:

```
from matplotlib import pyplot as plt
plt.plot(Data['Time'], Data['ATX'])
plt.show()
#
# another example in Python, relying on the ggplot2 import above:
g = ggplot2.ggplot(Data) + ggplot2.aes_string(x='Time', y='ATX') + \
    ggplot2.geom_path()
g.plot()
```

A subsequent command like the following will generate and display a plot using the Ranadu function "plotWAC()" instead:

```
robjects.r('Data %>% dplyr::select(Time, WIC) %>% plotWAC()')
# to dismiss the R window that appears, use ‘‘robjects.r('dev.off()')’’
```

The "Time" variable in the R data.frame is a POSIX-format time, but the resulting Python object uses units of seconds since 1970. You may have to deal with this difference when constructing plots using Python.

It is also possible to place data generated using Python back into the local R instance to use R routines for analysis. See the "rpy2" documentation for further information regarding this.

6.4 Reproducible research

With the tools now available, it is possible to document analysis projects to a degree that others can duplicate them using archived information. Steps toward that goal are the topic of this section. It is suggested that proper documentation of a project should include these components:

1. The project report, documenting the analysis steps, data used, results and interpretation.

2. Any code used.

3. Enough information on the underlying programming language (version number, operating system, etc.) that someone else can use the same code interpreter if necessary.

4. Locations of data files used, if in maintained archives, or copies of the data sufficient to reproduce the results.

5. A discussion of the workflow required to reproduce the research. This may include discussion of aspects of the code that may not be evident to an inexperienced reader, documentation of investigations not included in the report, reasons for choices made, and other advice to a person seeking to reproduce the research that might not be appropriate in the project report. The workflow document can be less formal and more wordy or chatty that the project report if that material might be useful to another analyst.

Often, analysis steps are stutter-steps producing scattered material that is hard to assemble, with different steps used to generate plots, manipulate data, perform fits, construct derived data, combine multiple and supplementary data sets, etc. Reproducibility does not mean necessarily following that original path, but a logical path using the successful steps should be documented. Essential but not adequate steps toward reproducibility include making the code available in some repository and ensuring that the data as used is archived where it is accessible. The project report should indicate where these components of the analysis are saved. The additional component that will usually be needed by a reproducing analyst is a workflow document, which can be thought of as guidance to a person wishing to verify or extend the results.

R tools are available that are of great utility in performing reproducible research. The "knitr" package (see references) makes it possible to assemble the text and code in the same file and to use knitr functions to reference results from the code in the text or embed graphics in the document as generated in the code. The "Rnw" format (and other alternative formats) support this approach, and running that program can generate the project report while running the specified code. This avoids ad hoc assembly of figures, tables, and text from different sources, which often obscures efforts to reproduce the work. A suggested documentation package can then include the Rnw-format (or equivalent) file, the report in text form, links to archives where the data are available or alternately inclusion of the data in the archived project package, a workflow discussion, and documentation of the version of various programs and computer systems used. Some more information on using knitr is included in the "RSessions" shinyApp tutorial, in the "reproducibility" tab.

6.5 The Ranadu Shiny app

A shiny app that uses the Ranadu package to examine data files is documented here.