faampy Documentation

Release 0.1.5 (a)

Axel Wellpott

Feb 20, 2018

Contents

Contents:

CHAPTER 1

Motivation

The faampy repository contains a number of python modules and scripts that handle data from the FAAM aircraft. The goal of faampy is it to provide useful and stable tools that deal with common tasks. It focuses on general utilities rather than analysing specific scientific problems. From experience many tasks (data merging, mapping, etc.) are common no matter which background the user has (Aerosol, Chemistry, Cloud Physics, ...). faampy is meant to be for the FAAM community **by** the FAAM community. Therefore users are encouraged to report/fix bugs and send in suggestions for improvements.

CHAPTER 2

Installing faampy

Installation of faampy is done in the usual way using the setup script:

```
git clone https://github.com/ncasuk/faampy.git
python setup.py build
sudo python setup.py install
```

So far the module has only been tested on linux machines and most of the code development has been done under python 2.7. However the idea is to make faampy python3 compatible and platform independent.

2.1 Example flight data, databases, . . .

Example data and databases of flight tracks are available for download. After installing the faampy module you can run:

```
faampy data_download ZIP_PASSWORD
```

from the command line. This will download a zip file and copies its content to a 'faampy_data' directory in your $HOME directory. However, for the moment the zip file that you download is password protected. Please contact me if you think you need the data and I will give you the password.

2.2 Disclaimer

faampy is in its early stages and has not been thoroughly tested. There will more modules been added in the near future. A backlog of modules exists that have been written, but will need to be tidied up, before being added to the repository.

CHAPTER 3

FAAM_Dataset

The FAAM_Dataset class handles the core_faam*nc files and smoothes out the reading process of the data and ensures that older files are read in the same way as newer ones. The class copies the behaviour of netCDF4.Dataset class.

A nifty method of the class is the merge method, which allows you to merge data from other data sources. The data type that can be merged is a numpy.recarray. The index for the procedure is the timestamp, of the FAAM_Dataset. Care is taken off gaps in the recarray.

A convenient option is exporting the Dataset into a pandas DataFrame, which then gives you all the amazing features of pandas. Due to the fact that pandas can not deal with multidimensional arrays, only the first measurement within a row is used for the DataFrame.

class faampy.core.faam_data.**FAAM_Dataset**(*filename*)

Dataset class which has much in common with the netCDF4.Dataset. The class has methods that helps to perform common tasks like merging and can copy

as_dataframe(*varnames=[]*)

Returns the Dataset as a pandas.Dataframe using the timestamp as index, which opens the world of pandas to you. Only the first column of two dimensional data sets is grabbed, because pandas does not handle multidimensional data very well.

Parameters **varnames** (*list*) – list of variable names that should be exported as DataFrame. By default all are exported

Returns returns a pandas Dataframe with the Timestamp as index

as_kml(*extrude=1*, *tessellate=1*)

Returns a kml linestring which represents the flight track of the current dataset

Parameters

- **extrude** (*boolean*) – whether the linestring is extruded
- **tessellate** (*boolean*) – whether the linestring is tesselated

Returns kml string

close()

Closing dataset

merge(*recarray*, *index=''*, *varnames=[]*, *delay=0*)
Merges in a numpy recarray with the FAAM_Dataset using concurring timestamps

Parameters

- **recarray** (*numpy.recarray*) – A numpy numpy.recarray with named data
- **index** (*str*) – Name of the column/field that contains the timestamp. Note that the merging only works on timestamps. The maximum time resolution is 1sec.
- **varnames** (*list of strings*) – List of varnames from the input array that should be merged
- **delay** (*int*) – instruments have a time offset compared to the core data. For example the FGGA is aboute four seconds slower than the core temperature measurements. The delay keyword takes this care of this and shifts the data.

write(*outfilename*, *v_name_list=[]*, *as_1Hz=True*, *clobber=False*)
Writing dataset out as NetCDF

Parameters

- **outfilename** (*str*) – path for the new NetCDF
- **v_name_list** (*list*) – list of variables names that should be written. By default all variables are added to the NetCDF
- **as_1Hz** (*boolean*) – Writes only 1Hz data out. If the variable is avaiable in higher frequency only the first value within the second is used rather than the average from the number of data points
- **clobber** (*boolean*) – Overwrites the files if it exists

CHAPTER 4

Command line tools

The faampy module provides a number of command line tools. All those commands are called via:

```
faampy SUBCOMMAND [OPTIONS] ARGUMENTS
```

A list of available subcommands is shown by just typing "faampy" on the command line.

4.1 nimrod_to_nc

This script converts the NIMROD weather radar data format into a netCDF with the dimensions:

- Timestamp
- Longitude
- Latitude

The original array in the NIMROD data fits the OSGB 1936 spatial reference (EPSG:27700) system. However, to make the raster work with for example cis it is necessary to warp the array to WGS84 (EPSG:4326) so that longitude and latitude are available as dimensions.

The new netCDF results were tested with the cistools. If the netCDF stores more than one timestamp it is necessary to extract one layer using the subset command like this:

```
cis subset rain_intensity:nimrod.nc timestamp=['2012-03-04T00:50'] -o nimrod_
↪20120304T0050.nc
```

The above command extracts the data for the timestamp '2012-03-04T00:50' and writes a new netCDF with the filename "nimrod_20120304T0050.nc".

In a next step it is possible to plot the data as a heatmap:

```
cis plot rain_intensity:nimrod_20120304T0050.nc
```

Maybe there is a way to skip the subset step but so far I have not found it.

```
usage: faampy nimrod_to_nc [-h] [-n NUMBER_OF_PROCESSES] [-o OUTPATH]
                           rain_radar_tar_file
```

4.1.1 Positional Arguments

rain_radar_tar_file MetOffice compressed rain radar file

4.1.2 Named Arguments

-n, --number_of_processes Number of processes that can be used.

-o, --outpath Directory where the netCDF file will be stored. Default: $HOME.

4.2 nc_to_gpx

Creates a gpx file from a FAAM core netcdf data file.

GPX files are a standard xml-format, which is understood by many programs. The gpx file from a flight can for example be used to georeference photographs taken during a flight (e.g. gpscorrelate).

The geotagged images can then in a next step be overlaid on maps using the GPS information from the image metadata.

```
usage: faampy nc_to_gpx [-h] ncfile [outpath]
```

4.2.1 Positional Arguments

ncfile input netCDF-file

outpath file name of output file

4.3 ge_avaps

4.4 ge_flight_track

4.5 ge_ncas_airquality

Script downloads the model output from the NCAS air quality model and creates a kmz file that is viewable in google-earth. This allows for example FAAM flight tracks to be overlaid on top of the model images.

```
usage: faampy ge_ncas_airquality [-h] [-o OUTPATH] [-d DATE] [-l LIMIT]
```

4.5.1 Named Arguments

-o, --outpath outpath

-d, --date date in the format: %Y-%m-%d

-l, --limit maximum number of model images to be processed

4.6 ge_nimrod_to_kmz

Extracts all rain radar data from the tar file and:

- creates a 8bit png image file with a custom colour palette
- convertes the png to a gtiff and adds the projection ESPG:27700=OSGB1936
- warpes the gtiff image to EPSG:4326
- creates one kmz file with a folder which contains all gtiff-4326 files as groundoverlays

The original UKMO nimrod data files can be downloaded from CEDA.

Note: The script is very wasteful in terms of disk space. The temporary folder that is created for storing the image files can grow to several GB. The script deletes the temporary folder by default.

```
usage: faampy ge_nimrod_to_kmz [-h] [-o OUTPATH] [-k] rain_radar_tar_file
```

4.6.1 Positional Arguments

rain_radar_tar_file MetOffice compressed rain radar file

4.6.2 Named Arguments

-o, --outpath Directory where the kmz file will be stored. Default: $HOME.

-k, --keep-folder If option is set the temporary directory will *not* be deleted. Default: False

4.7 ge_photo_album

Photo album creator for google-earth from georeferenced photos. Script produces a kmz files for google-earth with all the photos in the input folder. The images need to contain GPS location information. This kind of information can be added to the image files using a tool like gpscorrelate and a gpx file which contains the track of the flight.

```
usage: faampy ge_photo_album [-h] path outfile
```

4.7.1 Positional Arguments

path directory which holds the photographs. All photographs in the directory will be added to the photo album.

outfile outfile name

4.8 ge_ncvar_to_kml

Creates a profile plot for a specfic netCDF variable that is viewable in google-earth.

```
usage: faampy ge_ncvar_to_kml [-h] [--offset OFFSET]
                              [--scale_factor SCALE_FACTOR]
                              [--time_lag TIME_LAG] [--fltsumm FLTSUMM]
                              ncvar faam_core_netcdf outpath
```

4.8.1 Positional Arguments

ncvar	FAAM core netCDF variable name used for the profile.
faam_core_netcdf	FAAM core netCDF data file
outpath	Path to where the kml file is written to.

4.8.2 Named Arguments

--offset	Offset value. Value is removed from variable before profiles are created
--scale_factor	Scaling factor, mulitplier for the netCDF variable.
--time_lag	time lag between variable and GIN measurement caused by inlets
--fltsumm	Path to flight summary file for the specific flight

4.9 ge_was_to_kmz

WAS (Whole Air Sample) log as google-earth overlay. The flight track covered during the time the bottle was filled is represented as a line.

```
usage: faampy ge_was_to_kmz [-h] was_log_file ncfile outpath
```

4.9.1 Positional Arguments

was_log_file	Input WAS log file or folder which contains log files
ncfile	input netCDF-file or path to netcdf files
outpath	outpath for kmz file

4.10 sat_tracker

More satellite track information can be found at:

http://www.n2yo.com/

Popular platforms:

Platform	Sensor	ID
CALIPSO		29108
ISS	CATS	25544
TERRA	MODIS	25994
LANDSAT8		39084
SENTINEL-2A		40697
SENTINEL-3A		41335

Example:

```
faampy sat_tracker track --show_map "-38 35 -20 43" 39084 13-02-2017 17-02-2017 60
```

```
usage: faampy sat_tracker [-h] {sat_list,track} ...
```

4.10.1 Sub-commands:

sat_list

Undocumented

```
faampy sat_tracker sat_list [-h] [sat_name]
```

Positional Arguments

sat_name shows list of available satellites and their IDs

track

Undocumented

```
faampy sat_tracker track [-h] [-w] [-m [SHOW_MAP]]
                         [sat_id [sat_id ...]] start_time end_time timestep
```

Positional Arguments

sat_id Satellite ID(s). If more than one satellite track should be calculated the ids should be separated by commas

start_time date in the format DD-MM-YYYY or dd-mm-YYYYTHH:MM:SS

end_time date in the format DD-MM-YYYY or dd-mm-YYYYTHH:MM:SS

timestep Timestep in seconds

Named Arguments

-w, --write_to_file If flag is set the output is stored to a file in the $HOME directory. Default: False

-m, --show_map Boundary for the map in the form "left_longitude bottom_latitude right_longitude top_latitude". The input has

4.11 plt_quicklooks

4.12 qa_report

CHAPTER 5

Recipe - FAAM core

FAAM core data have been available as NetCDF ever since the 1st FAAM science flight took place back in 2004. However, details of the NetCDF have changed over time and the FAAM_Dataset class helps to iron out those small annoyances. Maybe at some point all FAAM flights will be reprocessed to create an unified format.

```
import os
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
from faampy.core.faam_data import FAAM_Dataset


ifile = os.path.join(os.path.expanduser('~'),
                     'gdrive',
                     'core_processing',
                     '2017',
                     'c013-may-17',
                     'core_faam_20170517_v004_r0_c013_1hz.nc')

# read in the FAAM netCDF
ds = FAAM_Dataset(ifile)
# get a pandas dataframe
df = ds.as_dataframe()
# strip of data when aircraft is on the ground
df = df[df.WOW_IND == 0]
# Plot a time series of TAT_ND_R
df.TAT_ND_R.plot()
plt.legend()
plt.grid()
# create 2nd y-axis
plt.twinx()
df.ALT_GIN.plot(color='firebrick')
plt.legend()

# The FAAM_Dataset has a coords attribute which contains the lon, lat, alt
# values for the flight. Those values can be used for plotting a flight track
```

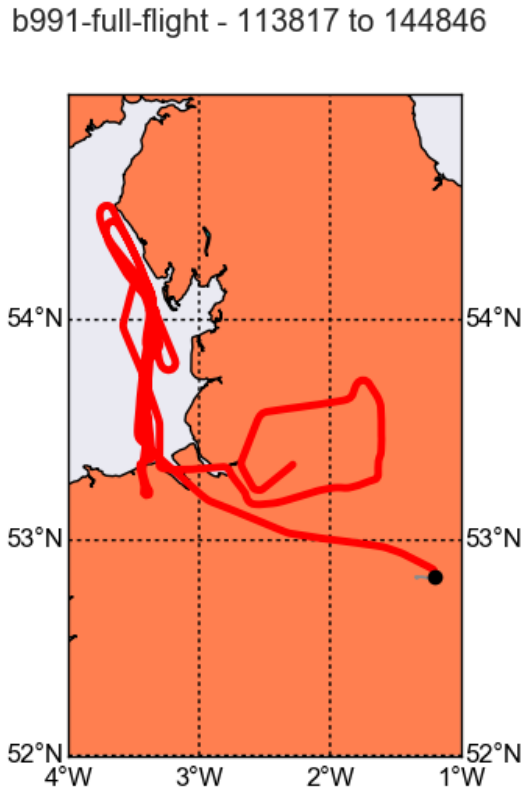
```
# opening a new plotting window
plt.figure()
# Setting up the map
ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines(resolution='10m')
# extract longitude and latitude from the coords
lon, lat, alt = zip(*ds.coords)
# plot the flight track
ax.plot(lon[::10], lat[::10], lw=2)
```

CHAPTER 6

Recipe - Post Flight Analysis

This is an example for what the post flight analysis for a typical FAAM chemistry flight could look like.

The data we are using are from the "Into the Blue" flight b991 on the 24th October 2016. This flight took us up and down the west coast between Morecambe and Wales. On that stretch some "plumes" were sampled, that originated from the Manchester/Liverpool area.



Warning: All the provided chemistry data in the example dataset are preliminary and uncalibrated. Therefore the data are not suitable for scientific publication.

6.1 Getting Started

At the start we need to import a number of modules and define a few variables that we need in later steps.

```
import datetime
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import faampy
from faampy.core.faam_data import FAAM_Dataset

year, month, day = 2016, 10, 24
FID = 'b991'

core_file = os.path.join(faampy.FAAMPY_EXAMPLE_DATA_PATH,
                         'b991',
                         'core',
                         'core_faam_20161024_v004_r0_b991.nc')

fltsumm_file = os.path.join(faampy.FAAMPY_EXAMPLE_DATA_PATH,
                            'b991',
                            'core',
                            'flight-sum_faam_20161024_r0_b991.txt')
```

Reading in data from the different chemistry instruments.

```
# define the input data file
nox_file = os.path.join(faampy.FAAMPY_EXAMPLE_DATA_PATH,
                        'b991',
                        'chem_data',
                        'NOx_161024_090507')

# defining the function that calculates the timestamp
nox_dateparse = lambda x: pd.datetime(year, month, day) + \
      datetime.timedelta(seconds=int(float(float(x) % 1)*86400.))

df_nox = pd.read_csv(nox_file, parse_dates=[0], date_parser=nox_dateparse)
df_nox = df_nox.set_index('TheTime')  # Setting index
t = df_nox.index.values
df_nox['timestamp'] = t.astype('datetime64[s]')  # Converting index data type
df_nox = df_nox[['timestamp', 'no_conc', 'no2_conc', 'nox_conc']]
df_nox[df_nox < 0] = np.nan

# Now the FGGA data
from faampy.data_io.chem import read_fgga
fgga_file = os.path.join(faampy.FAAMPY_EXAMPLE_DATA_PATH,
                         'b991',
                         'chem_data',
                         'FGGA_20161024_092223_B991.txt')
```

```
df_fgga = read_fgga(fgga_file)

# Using the valve states for flagging out calibration periods
df_fgga.loc[df_fgga['V1'] != 0, 'ch4_ppb'] = np.nan
df_fgga.loc[df_fgga['V2'] != 0, 'co2_ppm'] = np.nan
df_fgga.loc[df_fgga['V2'] != 0, 'ch4_ppb'] = np.nan


# Last but not least: Reading in the FAAM core data file using the FAAM_Dataset
# object from the faampy module

ds = FAAM_Dataset(core_file)
```

Merge the data different datasets.

```
# merge chemistry data with the core data set
# The delay keyword is used to set off the chemistry measurements. Due to the
# fact that the air has to travel through tubings in the cabine those
# instruments are slower than e.g compared to the temperature measurements
ds.merge(df_nox.to_records(convert_datetime64=False),
         index='timestamp', delay=3)
ds.merge(df_fgga.to_records(convert_datetime64=False),
         index='timestamp', delay=4)

# define variable list, that we like to extract
var_list = ['Time', 'LAT_GIN', 'LON_GIN', 'ALT_GIN', 'HGT_RADR',
            'CO_AERO', 'U_C', 'V_C', 'W_C', 'U_NOTURB', 'V_NOTURB',
            'WOW_IND', 'TAT_DI_R', 'TDEW_GE', 'PS_RVSM', 'ch4_ppb', 'co2_ppm',
            'no_conc', 'no2_conc', 'nox_conc', 'TSC_BLUU', 'TSC_GRNU',
            'TSC_REDU', 'BSC_BLUU', 'BSC_GRNU', 'BSC_REDU', 'IAS_RVSM']

# write the netcdf out to you HOME directory
outfile = os.path.join(os.path.expanduser('~'), '%s_merged.nc' % (FID.lower()))
print('Written ... %s' % (outfile,))
ds.write(outfile,
         clobber=True,
         v_name_list=var_list)
```

6.2 Google-Earth overlays

The commands in this section are run from the konsole. To keep the filenames short we move into the directory where the data for b991 are located:

```
cd ~/faampy_data/example_data/b991
```

We create a gpx (GPS Exchange Format) file:

```
faampy nc_to_gpx core/core_faam_20161024_v004_r0_b991.nc .
```

We use the gpx data file to geotag a few photographs that were taking during the flight. The gpscorrelate utility can be installed from the linux distribution package manager:

```
gpscorrelate --gps b991_20161024.gpx --photooffset -3600 photos/*jpg
```

Now that the photos are geotagged it is possible to create a photo album:

```
faampy ge_photo_album ./photos ./ge_photo_album_20161024_b991.kmz
```

WAS (Whole Air Sample) bottle overlay:

```
faampy ge_was_to_kmz ./chem_data/B991.WAS ./core/core_faam_20161024_v004_r0_b991_1hz.
↪nc .
```

Now make profiles for some of the variables in the created merged file.

```
from faampy.mapping import ge_ncvar_to_kml

# We are now continuing to work with the merged data file, that was produced
# in the previous step

core_file2 = os.path.join(os.path.expanduser('~'),
                          '%s_merged.nc' % (FID.lower()))

opath = os.path.expanduser('~')

ge_ncvar_to_kml.process(core_file2, 'CO_AERO', 0, -100, 500, opath)
ge_ncvar_to_kml.process(core_file2, 'co2_ppm', 0, -435, 1500, opath)
ge_ncvar_to_kml.process(core_file2, 'ch4_ppb', 0, -2115, 500, opath)
```

6.3 Quicklook Figures

faampy provides a tool to create quicklook figures using information from the flight summary. According to the event name (e.g. Profile, Run, ...) either a time series or a profile plot is produced. Maps are created for every event and tephigrams for every profile.

```
import faampy.plotting.quicklooks as q

Plot_Config = [[['TSC_BLUU', 'TSC_GRNU', 'TSC_REDU'],
                ['BSC_BLUU', 'BSC_GRNU', 'BSC_REDU']],
               [['CO_AERO'],],
               [['ch4_ppb'], ['co2_ppm']],
               [['no_conc'], ['no2_conc'], ['nox_conc']]]

# define the outpath, where all the figures should be saved to
quicklooks_outpath = os.path.join(os.path.expanduser('~'), 'b991_quicklooks')

# Check if directory exists; if not create it
if not os.path.exists(quicklooks_outpath):
    os.makedirs(quicklooks_outpath)
    print('Directory created: %s ...' % (quicklooks_outpath))

b991_qlooks = q.Quicklooks(fltsumm_file,
                           core_file2,
                           quicklooks_outpath)

# Set-up the plot configuration
b991_qlooks.Plot_Config = Plot_Config
b991_qlooks.process()
```

Make the output directory for the quicklook figure files:

```
mkdir ~/b991_quicklooks
```

The quicklook tool is also available from the command line. First create a quicklooks configuration file (quicklooks.cfg) which defines the figure layout:

```
touch quicklooks.cfg
```

Add the following text to the quicklooks.cfg file using a text editor:

```
[['TSC_BLUU', 'TSC_GRNU' , 'TSC_REDU'], ['BSC_BLUU', 'BSC_GRNU', 'BSC_REDU']]
[['CO_AERO'],]
[['ch4_ppb'], ['co2_ppm']]
[['no_conc'], ['no2_conc'], ['nox_conc']]
```

Every line defines one figure and the number of subplots. For example the first line ([['TSC_BLUU', 'TSC_GRNU' , 'TSC_REDU'], ['BSC_BLUU', 'BSC_GRNU', 'BSC_REDU']]) will create two subplots. In the 1st of these the total scatter values from the Nephelometer will be plotted and in the 2nd subplot the backscatter values will be plotted.

We will use the merged data file, which we produced in the previous section. This file contains the NOx and FGGA data. The command for creating the quicklooks is:

```
faampy plt_quicklooks --config_file quicklooks.cfg b991_merged.nc \
  ./core/flight-sum_faam_20161024_r0_b991.txt ~/b991_quicklooks/
```

If the above commmand was successful the figures should have been created in the b991_quicklooks directory in your home directory.

6.4 Coloured Line Figure

```
import numpy as np
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import os
from matplotlib.collections import LineCollection

from faampy.core.faam_data import FAAM_Dataset

cities = [('Liverpool',  53.410565, -2.992983),
          ('Manchester', 53.484538, -2.242493),
          ('Blackpool',  53.817770, -3.035729),
          ('Preston',    53.763530, -2.703519),
          ('Southport',  53.648671, -3.007051),
          ('Blackburn',  53.748788, -2.488308)]

### SETTINGS ##########################################################################

# Define map extent
ymin = 53.2
ymax = 54.0
xmin = -3.5
xmax = -2.0

FID = 'b991'
year, month, day = 2016, 10, 24
```

```
_RUNS = [('Run 2@100ft',  '121217', '122142'),
         ('Run 3@500ft',  '122257', '122937'),
         ('Run 4@1000ft', '123134', '124152'),
         ('Run 5@1000ft', '124305', '125133')]

_run = _RUNS[0]

# The Variable name that we want to plot
VARIABLE_NAME = 'nox_conc'
# Range for the chemical that we are interested in
rng = (0, 30000)

################################################################################


#http://nbviewer.jupyter.org/github/dpsanders/matplotlib-examples/blob/master/
↪colorline.ipynb
def colorline(x, y, z=None, cmap=plt.get_cmap('copper'),
              norm=plt.Normalize(0.0, 1.0), linewidth=3, alpha=1.0):
    """
    Plot a colored line with coordinates x and y
    Optionally specify colors in the array z
    Optionally specify a colormap, a norm function and a line width
    """

    # Default colors equally spaced on [0,1]:
    if z is None:
        z = np.linspace(0.0, 1.0, len(x))

    z = np.asarray(z)

    segments = make_segments(x, y)
    lc = LineCollection(segments, array=z, cmap=cmap, norm=norm, \
                        linewidth=linewidth, alpha=alpha)
    ax = plt.gca()
    ax.add_collection(lc)
    return lc


def make_segments(x, y):
    """
    Create list of line segments from x and y coordinates, in the correct
    format for LineCollection:
    an array of the form   numlines x (points per line) x 2 (x and y) array
    """
    points = np.array([x, y]).T.reshape(-1, 1, 2)
    segments = np.concatenate([points[:-1], points[1:]], axis=1)
    return segments


# Read in the merged data file
core_file2 = os.path.join(os.path.expanduser('~'), '%s_merged.nc' % (FID.lower()))
ds = FAAM_Dataset(core_file2)
# Get the data as a pandas dataframe
df = ds.as_dataframe()

# set up the map
```

```
wgs84 = ccrs.PlateCarree() #  we need this for transforming data points
proj = ccrs.OSGB()  # this is the standard map projection for Great Britain
ax = plt.axes(projection=proj)
ax.set_extent([xmin, xmax, ymin, ymax])
ax.coastlines(resolution='10m', color='black', linewidth=1)

# add the cities as dots to the map
for city in cities:
    x, y = proj.transform_point(city[2], city[1], wgs84)
    ax.scatter(x, y, color='grey')
    ax.text(x, y, city[0], horizontalalignment='right')

# convert from GPS coordinates (WGS84) to OSGB
coords = proj.transform_points(wgs84,
                               df['LON_GIN'].values,
                               df['LAT_GIN'].values)
x = coords[:,0]
y = coords[:,1]

z = df[VARIABLE_NAME].values

# extract the data for the run
start_time = datetime.datetime(year, month, day,
                               int(_run[1][0:2]),
                               int(_run[1][2:4]),
                               int(_run[1][4:6]))
end_time = datetime.datetime(year, month, day,
                             int(_run[2][0:2]),
                             int(_run[2][2:4]),
                             int(_run[2][4:6]))

ixs = [np.where(df.index == start_time)[0][0],
       np.where(df.index == end_time)[0][0]]

x = x[ixs[0]:ixs[1]]
y = y[ixs[0]:ixs[1]]
z = z[ixs[0]:ixs[1]]

#http://matplotlib.org/examples/pylab_examples/multicolored_line.html
lc = colorline(x, y, z=z,
               cmap=plt.get_cmap('gnuplot2'),
               norm=plt.Normalize(rng[0], rng[1]),
               linewidth=5)

# No the wind barbs
# Reduce the number of data points by averaging the data; otherwise
# the plot will look messy
df_30s = df[ixs[0]:ixs[1]].resample('30s').mean()

# calculate the coords for the figure
coords = proj.transform_points(wgs84,
                               df_30s['LON_GIN'].values,
                               df_30s['LAT_GIN'].values)
x = coords[:,0]
y = coords[:,1]

u, v = (df_30s['U_C'].values, df_30s['V_C'].values)
```

```
# finally plot the wind barbs
plt.barbs(x, y, u, v,
          length=7,
          barbcolor='k',
          flagcolor='k',
          linewidth=0.5,
          zorder=2)

# add a colorbar to the plot
cb = plt.colorbar(lc, label="NOx (ppt)")

ax.set_title(_run[0])
```

6.5 Transect Figure

```
import matplotlib.gridspec as gridspec

# seaborn changes the plot layout
try:
    import seaborn
except:
    pass

# Figure layout set up
# 4 rows 1 column and all figures should share the x-axis
gs = gridspec.GridSpec(4, 1)
fig = plt.figure()
fig.add_subplot(gs[3,:])
fig.add_subplot(gs[2,:], sharex=fig.get_axes()[0])
fig.add_subplot(gs[1,:], sharex=fig.get_axes()[0])
fig.add_subplot(gs[0,:], sharex=fig.get_axes()[0])


# Loop over the runs
for i, _run in list(enumerate(_RUNS)):
    print('%i: %s' % (i, _run[0]))
    ixs = (datetime.datetime(year, month, day,
                             int(_run[1][0:2]),
                             int(_run[1][2:4]),
                             int(_run[1][4:6])),
           datetime.datetime(year, month, day,
                             int(_run[2][0:2]),
                             int(_run[2][2:4]),
                             int(_run[2][4:6])))

    df_extracted=df[list(df.index.to_pydatetime()).index(ixs[0]):\
                    list(df.index.to_pydatetime()).index(ixs[1])]

    plt.sca(fig.get_axes()[0])
    ax = plt.gca()
    plt.plot(df_extracted.LAT_GIN.values,
             df_extracted.nox_conc.values,
             label=_run[0], lw=3)
    plt.xlabel('Latitude')
    plt.ylabel(r"NOx (ppt)")
```

```
    plt.grid()

    plt.sca(fig.get_axes()[1])
    ax = plt.gca()
    plt.setp(ax.get_xticklabels(), visible=False) # remove the xtick labels
    plt.plot(df_extracted.LAT_GIN.values,
             df_extracted.ch4_ppb.values,
             label=_run[0], lw=3)
    plt.ylabel(r"CH4 (ppb)")

    plt.sca(fig.get_axes()[2])
    ax = plt.gca()
    plt.setp(ax.get_xticklabels(), visible=False)
    plt.plot(df_extracted.LAT_GIN.values,
             df_extracted.co2_ppm.values,
             label=_run[0], lw=3)
    plt.ylabel(r"CO2 (ppm)")

    plt.sca(fig.get_axes()[3])
    ax = plt.gca()
    plt.setp(ax.get_xticklabels(), visible=False)
    plt.plot(df_extracted.LAT_GIN.values,
             df_extracted.CO_AERO.values,
             label=_run[0], lw=3)
    plt.ylabel(r"CO (ppb)")
    ax.set_ylim(150, 260)
    plt.legend()

# adjust the x-axis limits
plt.xlim(53.3, 53.9)

plt.tight_layout()
```

CHAPTER 7

Recipe - Spatial Analysis

FAAM core data are stored as netCDF and come with *Time* as dimension. However, since the FAAM aircraft is a moving plaform location is obviously also a relevant dimension and spatial queries of the FAAM data can add useful functionality. To provide this feature the FAAM flight tracks are inserted as *linestring* into a database with spatial capabilities. Such a database allows queries like:

- "Give me all the flights that have crossed Scotland"
- "On which flights did we fly closer than 10nm miles pass the Queen's palace"
- "What length was the flight track"

7.1 DB description

The spatialite DB is stored in one single file, which is very convenient and does not require the setup of an advanced database, which can come with a lots of obstacles. In direct comparison spatialite is less powerful but has all the features that we need. For more information see:

https://www.gaia-gis.it/fossil/libspatialite/index
http://www.gaia-gis.it/gaia-sins/spatialite-sql-4.4.0.html
https://www.gaia-gis.it/gaia-sins/spatialite-tutorial-2.3.1.html
http://postgis.net/docs/

The example database has currently only one table and three colums:

fid - Flight id
date - Start date of flight
the_geom - Holds the linestring geometry

For the below examples python is our tool of choice, which has all the necessary modules to interact with the database.

7.2 Imports and DB connection

For the examples below to work we need to import some common modules and connect to the database.

```
import json
import numpy as np
import os
import osgeo.ogr
import simplekml

import faampy
from faampy.core.faam_spatial import FAAM_Spatial_DB

DB_FILE = os.path.join(faampy.FAAMPY_DATA_PATH, 'db', 'faam_spatial_db.sqlite')

db = FAAM_Spatial_DB(DB_FILE)
```

7.3 Example 1: Get some db info

Just get some basic information from the database.

```
print '\n'*3
print '=' * 40
print '\n'*3

print 'Some DB info'

# Count how many records are in the DB
sql="""SELECT COUNT(*) FROM flight_tracks;"""
cur = db.conn.cursor()
cur.execute(sql)
cnt = cur.fetchone()[0]

print 'Number of flights in the DB: %i' % (cnt,)
print ''

sql="""SELECT fid FROM flight_tracks ORDER BY fid;"""
cur = db.conn.cursor()
cur.execute(sql)
fids = sorted([i[0] for i in cur.fetchall()])
print('Latest fid in db: %s' % fids[-1])
all_fids = set(['b%0.3i' % i for i in range(1, 1000)] + \
               ['c%0.3i' % i for i in range(1, int(fids[-1][1:]))])
missing_fids = sorted(all_fids.difference(fids))

print 'Number Missing flights: %i' % (len(missing_fids),)
print 'Missing flights ids: %s' % (','.join(missing_fids),)
```

7.4 Example 2: Find flights that go over the North Sea

The goal is to find all FAAM flights that go over the North Sea. To do this we need the boundaries for the North Sea. A shapefile with the Polygon can be downloaded from the marineregions website:

```
print '\n'*3
print '=' * 40
print '\n'*3
print 'TASK: Finding all flights that go over the North Sea'
print '\n'

# The shape (Polygon from the North Sea was downloaded from the web
# http://www.marineregions.org/gazetteer.php?p=details&id=2350

print 'Reading in the shape file for the North Sea'
shp_file = os.path.join(faampy.FAAMPY_DATA_PATH, 'shp', 'north_sea.shp')
sf = osgeo.ogr.Open(shp_file)
layer = sf.GetLayer()
ns = layer.GetFeature(0)          # there is only one feature in the layer
geometry = ns.GetGeometryRef()
ns_wkt = geometry.ExportToWkt()   # Getting a Well-known text representation

print 'Give me all flights where the track intersects the North Sea Polygon'

# Give me all flights where the track intersects the North Sea Polygon. Now that
# we have the Geometry in wkt format we can use it to create a sql query that we
# can send to the spatialite DB.

sql = "SELECT FT.fid FROM flight_tracks FT where "
sql += "ST_Intersects( GeomFromText('%s'), FT.the_geom) " % (ns_wkt,)
sql += "ORDER BY FT.fid;"
cur = db.conn.cursor()                          # connect
cur.execute(sql)                                # execute
fids = [i[0] for i in cur.fetchall()]           # flatten the result

print ''
print 'Number of flights that intersect the North Sea: %i' % (len(fids),)
print ''
print 'List flights that intersect the North Sea: %s\n' % (','.join(fids),)


# Now that we have all the fids that intersected the North Sea, we want
# to look at them using google-earth. Spatialite has the capability of
# formatting the geometries into a kml string (askml)

sql = "SELECT askml(Simplify(FT.the_geom, 0.01)) FROM flight_tracks FT WHERE"
sql += " FT.fid IN (%s)" % (str(','.join(["'%s'" % fid for fid in fids])))
cur.execute(sql)
flight_tracks_kml = cur.fetchall()


#Create a new kml file
kml=simplekml.Kml()
folder=kml.newfolder(name='Spatialite result')
lines=kml.kml().split('\n')
lines.insert(-4, '<Placemark>')
lines.insert(-4, '<name>North Sea</name>')
```

```
lines.insert(-4, geometry.ExportToKML())
lines.insert(-4, '</Placemark>')
for i, flight_track in enumerate(flight_tracks_kml):
    lines.insert(-4, '<Placemark>')
    lines.insert(-4, '<name>%s</name>' % (fids[i],))
    lines.insert(-4, flight_track[0])
    lines.insert(-4, '</Placemark>')


ofilename = os.path.join(os.environ['HOME'], 'fids_crossing_ns.kml')
print 'Writing North Sea Polygon and the flight track linestrings as kml'
print 'kml written to: %s' % (ofilename,)
ofile = open(os.path.join(os.path.expanduser('~'), ofilename), 'w')
ofile.write('\n'.join(lines))
ofile.close()
```

7.5 Example 3: Get the length of a flight track

Get the length of a flight track. The database can do this for us using the *GreatCircleLength* function.

```
print '\n'*3
print '=' * 40
print '\n'*3
# Find the length of flight track for a specific flight
fid = 'b659'
print 'TASK: Give me the length of %s' % (fid,)
print '\n'
sql = "SELECT GreatCircleLength(the_geom) from "
sql += "flight_tracks where fid = '%s';" % (fid, )
cur = db.conn.cursor()                          # connect
cur.execute(sql)                                # execute
length = cur.fetchone()[0]/1000.
print 'Flight %s was %.2f km long.' % (fid, length)
```

7.6 Example 4: Get all flights when the ARA climbed above a certain altitude

We are trying to find all the flights where we climbed above a certain gps altitude. For this we loop over all individual flight tracks. The steps are: 1. Get flight track from DB in json format 2. Use the 'coordinates' key from the json and extract the z-coordinate 3. Check if the maximum z-value is greater than the MAX_ALT and store the fid in the result list if that's the case

```
print '\n'*3
print '=' * 40
print '\n'*3
# Print give me all flights when we climbed above 11000m
# There does not seem to be a way to do this directly in spatialite, so we
# do some simple data crunching in python
```

```
#
# To do this we need to get the xyz coordinates for each flight first and check
# those. I did not find a way to query the linestring directly
# Spatialite can return the geometry in json format which can then easily
# converted into a dictionary with 'coordinates' being one of the keys
MAX_HEIGHT = 11000
print 'TASK: Finding flights exceeding %i m altitude' % (int(MAX_HEIGHT,))
sql = """SELECT fid, AsGeoJSON(the_geom) from flight_tracks;"""
cur = db.conn.cursor()                          # connect
cur.execute(sql)                                # execute
result = cur.fetchall()
fid_max_alt_list = []
for r in result:
    fid = r[0]
    # get the coordinates from the geojson
    coords = np.array(json.loads(r[1])['coordinates'])
    # the alt coordinate is the 3rd column
    alt_max = np.nanmax(coords[:,2])
    fid_max_alt_list.append((fid, alt_max))

fids = sorted([i[0] for i in fid_max_alt_list if i[1] > MAX_HEIGHT])
print 'N fids with gps height > %i: %i' % (int(MAX_HEIGHT), len(fids),)
print ''
print 'List of flight ids: %s\n' % (','.join(fids),)
```

7.7 Example 5: Get all flights that took off from Cranfield

```
# http://stackoverflow.com/questions/19412462/getting-distance-between-two-points-
↪based-on-latitude-longitude-python
def calc_distance(lat1, lon1, lat2, lon2):
    from math import sin, cos, sqrt, atan2, radians
    # approximate radius of earth in m
    R = 6373000.0
    lat1 = radians(lat1)
    lon1 = radians(lon1)
    lat2 = radians(lat2)
    lon2 = radians(lon2)
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    distance = R * c
    return distance

print('TASK: Finding flights that took off in Cranfield in every year')
Cranfield_Coords = (52.072222, -0.616667)    # Cranfield Airport coordinates
# in m; the distance is rather large to cover flights
# when the GIN didn't work straight away
MAX_DISTANCE = 15000
sql = """SELECT fid, date, AsGeoJSON(the_geom) from flight_tracks order by date;"""
cur = db.conn.cursor()                          # connect
cur.execute(sql)                                # execute
```

```
result = cur.fetchall()
# get a list of all years for which we do the analysis
years = list(set([r[1].split('-')[0] for r in result]))
dist_dict = {}
for y in years:
    dist_dict[y] = []

for r in result:
    fid = r[0]
    # get the coordinates from the geojson
    coords = np.array(json.loads(r[2])['coordinates'])
    # extract year string from sql result
    year = r[1].split('-')[0]
    lat1, lon1 = Cranfield_Coords
    # pull coordinates form the very first array
    lon2 = coords[0, 0]
    lat2 = coords[0, 1]
    dist = calc_distance(lat1, lon1, lat2, lon2)
    if dist < MAX_DISTANCE:
        dist_dict[year].append((fid, dist))

# print summary
total = 0
# print the number for every year
for year in sorted(dist_dict.keys()):
    n = len(dist_dict[year])
    total += n
    print('%7s: %3s' % (year, n))
print('%7s: %3s' % ('total', total))
```

CHAPTER 8

Recipe - FAAM meets cis

8.1 CIS Installation

Information about installing anaconda2 and cis:

anaconda2: https://www.continuum.io/downloads

cis: http://cistools.net/get-started#installation

Please note that the cis instructions say that you should install python 2.7 and **not** 3.x. If you are new to python you might be irritated why you would not install the very latest version of some software. In brief: The two versions are not fully compatible and many people decided to stick with 2.7 for the time being, because the benefits are just too small to move on.

8.2 FAAM netCDF preparation

The FAAM core data do not work with the cis tool straight away. The netCDF data product need a little tweaking to make them fully CF compliant, so that cis interprets the data correctly. The global attributes "Conventions" and "Coordinates" need to be added. A small bash snippet can do the changes using nc utilities. The example netCDF has already been edited and works with cis.

```
#!/bin/bash

ofile=`echo $1 | sed s/.nc/_editted.nc/`

ncatted \
 -a 'Version,global,c,c,1.5' \
 -a 'conventions,global,m,c,NCAR-RAF/nimbus' \
 -a 'Coordinates,global,c,c,LON_GIN LAT_GIN ALT_GIN Time' \
 -a '_FillValue,LAT_GIN,m,f,0' \
 -a '_FillValue,LON_GIN,m,f,0' \
 -a '_FillValue,ALT_GIN,m,f,0' \
 -o $ofile $1
```

```
ncwa -a sps01 -O $ofile $ofile
ncrename -d data_point,Time -O $ofile
```

Save the above code as faam_edit.sh and make it executable:

```
chmod u+x faam_edit.sh
./faam_edit.sh core_faam_20161024_v004_r0_b991_1hz.nc
```

The example data (core_faam_20161024_v004_r0_b991_1hz_editted.nc) are for flight b991 (24-10-2016), when the aircraft was flying downwind of Manchester and Liverpool measuring emissions from the two cities ahead of the *Into the Blue* event.

8.3 Starting cis

The next thing to do is to start the cis environment that we installed earlier. Go to the bin directory of your conda installation:

```
cd ~/anaconda2/bin/
```

and activate the environment:

```
source activate cis_env
```

From now on the shell should have the string '(cis_env)' in front indicating that we are working in the cis environment.

8.4 Working with cis and FAAM data

Below are several one line examples that show the functionality of the cis tools. Most of the examples have been taken and adapted from the cis online documentation.

Note: All the commands below go on **one** line in your shell. The page is just too small to get it all printed on one line.

Get information about the netCDF:

```
cis info TAT_ND_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc
```

and the true air temperature form the non-deiced sensor:

```
cis info TAT_ND_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc
```

Create scatter plot to compare the deiced (TAT_DI_R) and non-deiced (TAT_ND_R) temperature measurements on the ARA:

```
cis plot TAT_ND_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc \
  TAT_DI_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc \
  --type comparativescatter --grid \
  --title "True air temperature comparison" \
  --xlabel "non deiced sensor (K)" --ylabel "deiced sensor (K)"
```

And print some statistics about the TAT_DI_R variable:

```
cis stats TAT_ND_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc \
  TAT_DI_R:core_faam_20161024_v004_r0_b991_1hz_editted.nc
```

Make a coloured line plot, showing the CO concentration (variable name is CO_AERO) on a map:

```
cis plot CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc \
  --xaxis longitude --yaxis latitude --xmin -5 --xmax -2 --ymin 52.2 --ymax 55
```

Calculate mean, min, max for 1min time intervals for the CO_AERO data for the time interval 11:45 to 14:45. The results are written out to a new NetCDF:

```
cis aggregate CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc:kernel=mean \
  t=[2016-10-24T11:45,2016-10-24T14:45,PT1M] -o b991_co_aero_1min_mean.nc

cis aggregate CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc:kernel=max \
  t=[2016-10-24T11:45,2016-10-24T14:45,PT1M] -o b991_co_aero_1min_max.nc

cis aggregate CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc:kernel=min \
  t=[2016-10-24T11:45,2016-10-24T14:45,PT1M] -o b991_co_aero_1min_min.nc
```

Plot the three lines in one figure:

```
cis plot CO_AERO:b991_co_aero_1min_max.nc \
  CO_AERO:b991_co_aero_1min_mean.nc \
  CO_AERO:b991_co_aero_1min_min.nc
```

Reproducing an aggregation example from the documentation: http://cis.readthedocs.io/en/stable/aggregation.html#aircraft-track

The results from the aggregation will be saved to a netCDF (option -o). The following line aggregates over 5 minutes and over altitude in 200 meter steps in the range of 0 to 1000m:

```
cis aggregate CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc \
  t=[2016-10-24T11:45,2016-10-24T14:45,PT5M],z=[0,1000,200] \
  -o b991_co_aero_alt_time.nc
```

Plot a curtain (transect) using the netCDF that we just created:

```
cis plot CO_AERO:b991_co_aero_alt_time.nc --xaxis time --yaxis altitude
```

Make a grid plot from the mean, where each grid cell is 0.2 degrees in size. The results are written to a netCDF:

```
cis aggregate CO_AERO:core_faam_20161024_v004_r0_b991_1hz_editted.nc:kernel=mean \
  x=[-5,0,0.2],y=[52,55,0.2] -o b991_co_aero_grid_mean.nc
```

Now plot the grid on a map using the netcdf that we just created:

```
cis plot CO_AERO:b991_co_aero_grid_mean.nc
```

CHAPTER 9

Recipe - Data Mining

The code snipped below loops over all FAAM core data files and extracts the CO and O3 data plus coordinates. All the data are concatenated and written out into one large csv file.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

"""
Scripts finds all FAAM core data files and filters for the latest revision.
The FAAM_Dataset object from the faampy module is used, which makes processing
easier, by synchronicing variables names and adding an artificial WOW FLag for
old flights.

"""

import os
import netCDF4
import numpy as np

from faampy.core.faam_data import FAAM_Dataset
from faampy.utils.file_list import File_List

#=================================================================

ROOT_DATA_PATH = '/mnt/faamarchive/badcMirror/data/'
OUTFILENAME = os.path.join(os.path.expanduser('~'), 'chemistry_spatial.csv')

#=================================================================

# Get all hires FAAM core data that are in the ROOT_DATA_PATH directory
fl = File_List(ROOT_DATA_PATH)
fl.filter_by_data_type('core-hires')
fl.filter_latest_revision()


def extract(core_netcdf):
```

```
    """
    Extracts all CO and O3 data from a FAAM core netCDF.

    """

    ncfilename = os.path.join(core_netcdf.path, core_netcdf.filename)
    ds = FAAM_Dataset(ncfilename)
    _ds_index = ds.index.ravel()
    units = 'seconds since %s 00:00:00 +0000' % str(_ds_index[0])[:10]
    timestamp = netCDF4.num2date(ds.variables['Time'][:].ravel(), units)
    n = timestamp.size

    if 'CO_AERO' in ds.variables.keys():
        co_aero = ds.variables['CO_AERO'][:]
        co_aero_flag = ds.variables['CO_AERO_FLAG'][:]
        co_aero[co_aero_flag != 0] = -9999.0
    else:
        co_aero = np.zeros(n)-9999.0

    if 'O3_TECO' in ds.variables.keys():
        o3_teco = ds.variables['O3_TECO'][:]
        o3_teco_flag = ds.variables['O3_TECO_FLAG'][:]
        o3_teco[o3_teco_flag != 0] = -9999.0
    else:
        o3_teco = np.zeros(n)-9999.0

    # Old FAAM files didn't have the GIN instrument fitted
    if 'LAT_GIN' in ds.variables.keys():
        lon_var_name = 'LON_GIN'
        lat_var_name = 'LAT_GIN'
        alt_var_name = 'ALT_GIN'
    elif 'LAT_GPS' in ds.variables.keys():
        lon_var_name = 'LON_GPS'
        lat_var_name = 'LAT_GPS'
        alt_var_name = 'GPS_ALT'

    if len(ds.variables[lon_var_name][:].shape) > 1:
        x = ds.variables[lon_var_name][:, 0].ravel()
        y = ds.variables[lat_var_name][:, 0].ravel()
        z = ds.variables[alt_var_name][:, 0].ravel()
    else:
        x = ds.variables[lon_var_name][:].ravel()
        y = ds.variables[lat_var_name][:].ravel()
        z = ds.variables[alt_var_name][:].ravel()

    wow = ds.variables['WOW_IND'][:].ravel()

    timestamp_string = [t.strftime('%Y-%m-%dT%H:%M:%S') for t in timestamp]
    fid = [core_netcdf.fid,]*n
    result = zip(list(np.array(timestamp_string)[wow == 0]),
                 list(np.array(fid)[wow == 0]),
                 list(x[wow == 0]),
                 list(y[wow == 0]),
                 list(z[wow == 0]),
                 list(co_aero[wow == 0]),
                 list(o3_teco[wow == 0]))
    return result
```

```
# open the output file and write the column labels out
ofile = open(OUTFILENAME, 'w')
ofile.write('timestamp,fid,lon,lat,alt,co,o3\n')

# loop over all core files
for core_netcdf in fl:
    print('Working on ... %s' % core_netcdf.fid)
    try:
        data = extract(core_netcdf)
    except:
        print('  Issue with %s ...' % core_netcdf.fid)
        continue

    out_txt = ['%s,%s,%f,%f,%f,%f,%f\n' % l for l in data]
    out_txt = ''.join(out_txt)
    ofile.write(out_txt)
    ofile.flush()

ofile.close()
```

CHAPTER 10

Full List

10.1 faampy.utils

class faampy.utils.file_info.**File_Info**(*filename*)

Holds all file specific information for a FAAM data file:

- filename
- path
- Flight Number (fid)
- date
- revision
- datatype

class faampy.utils.file_list.**File_List**(**args*)

A list of File_Info objects. The list can be sorted and filtered which can be useful for batch processing.

For example it is possible to (i) get all DECADES rawdlu and flight constant files from a path, (ii) filter those for the latest revisions and reprocess them.

filter_by_data_type(*dtype*)

Filtering by data type.

filter_latest_revision()

Compresses the list and keeps only the latest revision for a FID.

get_filenames()

Returns the file names for each entry.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index