ACT Basics

Contents

Overview

The ARM TRACER campaign collected a lot of very interesting data in Houston, TX from October 1, 2021 to September 30, 2022. One event that stands out is a dust event that occurred from July 16 to July 19, 2022. This notebook will give an introduction to basic features in ACT, using some relevant datastreams from this event

1. Intro to ACT
2. Downloading and Reading in Data
3. Quality Controlling Data
4. Aerosol Instrument Overview
5. Visualizing Data
6. Additional Features in ACT

Skip to main content

Prerequisites

This notebook will rely heavily on Python and the Atmospheric data Community Toolkit (ACT). Don't worry if you don't have experience with either, this notebook will walk you though what you need to know.

You will also need an account and token to download data using the ARM Live webservice. Navigate to the webservice information page and log in to get your token. Your account username will be your ARM username.

Concepts	Importance	Notes
ACT	Helpful	

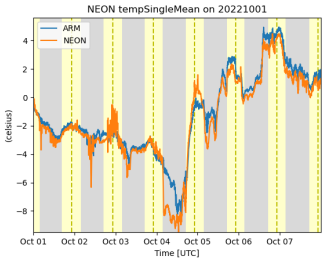
- **Time to learn**: 60 Minutes
- **System requirements**:
  - Python 3.11 or latest
  - ACT v1.5.0 or latest
  - numpy
  - xarray
  - matplotlib

Intro to ACT

The Atmospheric data Community Toolkit (ACT) is an open-source Python toolkit for exploring and analyzing atmospheric time-series datasets. Examples can be found in the ACT Example Gallery. The toolkit has modules for many different parts of the scientific process, including:

Data Discovery (act.discovery)

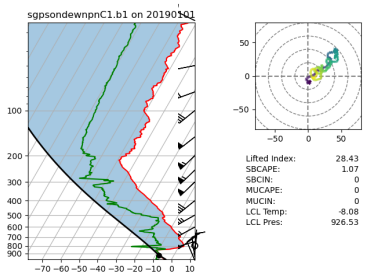
The discovery module houses functions to download or access data from different groups. Currently it includes function to get data for ARM, NOAA, EPA, NEON, and more!



ARM and NEON data from Utquivaik, AK

Input/Output (act.io)

io contains functions for reading and writing data from various sources and formats.



Enhanced Skew-T plot from ARM's Southern Great Plains Site (SGP)

Visualization (act.plotting)

plotting contains various routines, built on matplotlib, to help visualize and explore data. These include

1. Time-series plots
2. Distribution plots like histograms and heatmaps
3. Geographic plots for moving systems like radiosondes or aircraft
4. Skew-T plots for radiosonde data, built off MetPy
5. Wind rose plots for wind and data roses
6. Cross-section plots for working with 3-dimensional data

Corrections (act.corrections)

corrections apply different corrections to data based on need. A majority of the existing corrections are for lidar data.

Quality Control (act.qc)

The qc module has a lot of functions for working with quality control information, apply new tests, or filtering data based on existing tests. We will explore some of that functionality in this notebook.

Skip to main content

Retrievals (act.retrievals)

There are many cases in which some additional calculations are necessary to get more value from the instrument data. The retrievals module houses some functions for performing these advanced calculations.

Utilities (act.utils)

The utils module has a lot of general utilities to help with the data. Some of these include adding in a solar variable to indicate day/night (useful in filtering data), unit conversions, decoding WMO weather codes, performing weighted averaging, etc...

Imports

Let's get started with some data! But first, we need to import some libraries.

```
import act
import numpy as np
import xarray as xr
import matplotlib.pyplot as plt
```

Downloading and Reading ARM's NetCDF Data

ARM's standard file format is NetCDF (network Common Data Form) which makes it very easy to work with in Python! ARM data are available through a data portal called Data Discovery or through a webservice. If you didn't get your username and token earlier, please go back and see the Prerequisites!

Let's download some of the MPL data first but let's just start with one day.

```
# Set your username and token here!
username = 'YourUserName'
token = 'YourToken'

# Set the datastream and start/enddates
datastream = 'hou30smplcmask1zwangM1.c1'
startdate = '2022-07-16'
enddate = '2022-07-16'
```

```
# for ARM's instrument experts and cite their data if you use it in a publication
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
```

```
# Let's read in the data using ACT and check out the data
ds_mpl = act.io.armfiles.read_netcdf(result)

ds_mpl
```

```
ds_mpl['cloud_base'].plot()
```

Quality Controlling Data

ARM has multiple methods that it uses to communicate data quality information out to the users. One of these methods is through “embedded QC” variables. These are variables within the file that have information on automated tests that have been applied. Many times, they include Min, Max, and Delta tests but as is the case with the AOS instruments, there can be more complicated tests that are applied.

The results from all these different tests are stored in a single variable using bit-packed QC. We won’t get into the full details here, but it’s a way to communicate the results of multiple tests in a single integer value by utilizing binary and bits! You can learn more about bit-packed QC here but ACT also has many of the tools for working with ARM QC.

Other Sources of Quality Control

ARM also communicates problems with the data quality through Data Quality Reports (DQR). These reports are normally submitted by the instrument mentor when there’s been a problem with the instrument. The categories include:

- **Data Quality Report Categories**
  - Missing: Data are not available or set to -9999
  - Suspect: The data are not fully incorrect but there are problems that increases the uncertainty of the values. Data should be used with caution.
  - Bad: The data are incorrect and should not be used.
  - Note: Data notes are a way to communicate information that would be useful to the end user but does not rise to the level of suspect or bad data

Examples of ACT QC functionality

Additionally, data quality information can be found in the Instrument Handbooks, which are included on

```
# Let's take a look at the quality control information associated with a variable from the
variable = 'linear_depol_ratio'

# First, for many of the ACT QC features, we need to get the dataset more to CF standard a
# involves cleaning up some of the attributes and ways that ARM has historically handled Q
ds_mpl.clean.cleanup()

# Next, let's take a look at visualizing the quality control information
# Create a plotting display object with 2 plots
display = act.plotting.TimeSeriesDisplay(ds_mpl, figsize=(15, 10), subplot_shape=(2,))

# Plot up the variable in the first plot
display.plot(variable, subplot_index=(0,), cb_friendly=True)

# Plot up a day/night background
display.day_night_background(subplot_index=(0,))

# Plot up the QC variable in the second plot
display.qc_flag_block_plot(variable, subplot_index=(1,))
plt.show()
```

Filtering data

It's easy to filter out data failing tests with ACT. This will show you how to filter data by test or by assessment.

```
# Let's filter out test 5 using ACT.  Yes, it's that simple!
ds_mpl.qcfilter.datafilter(variable, rm_tests=[1, 2], del_qc_var=False)

# There are other ways we can filter data out as well.  Using the
# rm_assessments will filter out by all Bad/Suspect tests that are failing
# ds.qcfilter.datafilter(variable, rm_assessments=['Bad', 'Suspect'], del_qc_var=False)

# Let's check out the attributes of the variable
# Whenever data are filtered out using the datafilter function
# a comment will be added to the variable history for provenance purposes
print(ds_mpl[variable].attrs)

# And plot it all again!
# Create a plotting display object with 2 plots
display = act.plotting.TimeSeriesDisplay(ds_mpl, figsize=(15, 10), subplot_shape=(2,))

# Plot up the variable in the first plot
display.plot(variable, subplot_index=(0,), cb_friendly=True)

# Plot up a day/night background
display.day_night_background(subplot_index=(0,))

# Plot up the QC variable in the second plot
display.qc_flag_block_plot(variable, subplot_index=(1,))
plt.show()
```

ARM Data Quality Reports (DQR)!

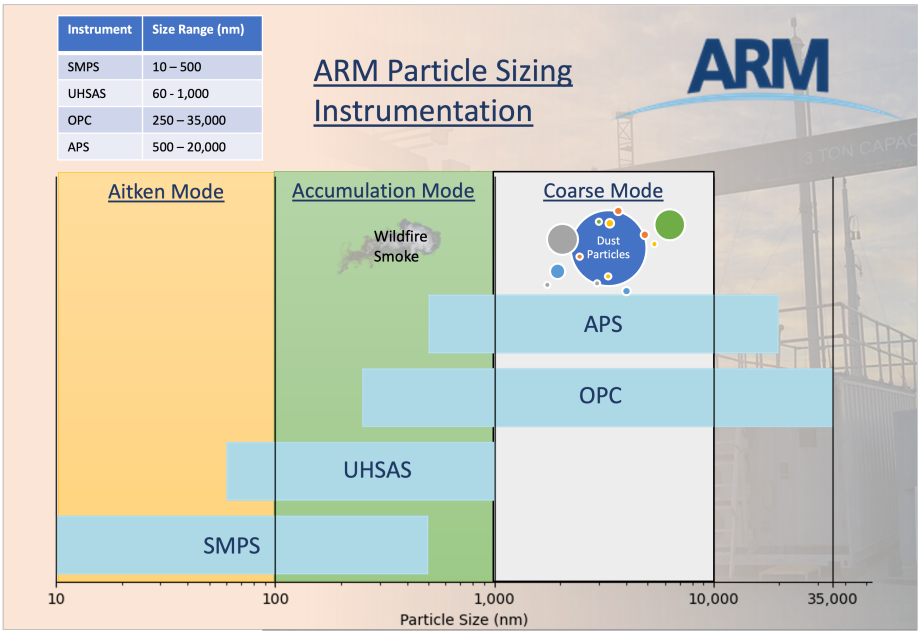
ARM's DQRs can be easily pulled in and added to the QC variables using ACT. We can do that with the below one line command. However, for this case, there won't be any DQRs on the data but let's visualize it just in case! Check out the ACT QC Examples for more use cases!

```
# Query the ARM DQR Webservice
ds_mpl = act.qc.add_dqr_to_qc(ds_mpl, variable=variable)

ds_mpl['qc_' + variable]
```

Aerosol Instrument Overview

Single Particle Soot Photometer (SP2)

The single-particle soot photometer (SP2) measures the soot (black carbon) mass of individual aerosol particles by laser-induced incandescence down to concentrations as low as ng/m^3. Learn more



ARM Aerosol Instrumentation Particle Size Ranges

Aerodynamic Particle Sizer (APS)

The aerodynamic particle sizer (APS) is a particle size spectrometer that measures both the particle aerodynamic diameter based on particle time of flight and optical diameter based on scattered light intensity. The APS provides the number size distribution for particles with aerodynamic diameters from 0.5 to 20 micrometers and with optical diameters from

Skip to main content

0.3 to 20 micrometers.
Learn more

Aerosol Chemical Speciation Monitor (ACSM)

The aerosol chemical speciation monitor is a thermal vaporization, electron impact, ionization mass spectrometer that measures bulk chemical composition of the rapidly evaporating component of sub-micron aerosol particles in real time. Standard measurements include mass concentrations of organics, sulfate, nitrate, ammonium, and chloride. Learn more

Downloading and QCing the Aerosol Data

Let's start pulling these data together into the same plots so we can see what's going on.

```
# Let's set a longer time period
startdate = '2022-07-10'
enddate = '2022-07-20'

# APS
datastream = 'houaosapsM1.b1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
ds_aps = act.io.armfiles.read_netcdf(result)

#ACSM
```

Skip to main content

```
ds_acsm = act.io.armfiles.read_netcdf(result)

#SP2
datastream = 'houaossp2bc60sM1.b1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
ds_sp2 = act.io.armfiles.read_netcdf(result)

# AOSMET - Just to get the wind data!
datastream = 'houmetM1.b1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
ds_met = act.io.armfiles.read_netcdf(result)

# MPL to get the full record
datastream = 'hou30smplcmask1zwangM1.c1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
ds_mpl = act.io.armfiles.read_netcdf(result)
```

```
# Before we proceed to plotting, let's reduce the MPL data down a little bit
# This will remove all data where heights are greater than 5
ds_mpl = ds_mpl.where(ds_mpl.height <= 3, drop=True)

# This will resample to 1 minute
ds_mpl = ds_mpl.resample(time='1min').nearest()
```

```
# Let's not forget about QCing the data!
# We can remove all the bad data from each aerosol dataset
ds_aps.clean.cleanup()
ds_aps = act.qc.arm.add_dqr_to_qc(ds_aps)
ds_aps.qcfilter.datafilter(rm_assessments=['Bad'], del_qc_var=False)

ds_acsm.clean.cleanup()
ds_acsm = act.qc.arm.add_dqr_to_qc(ds_acsm)
ds_acsm.qcfilter.datafilter(rm_assessments=['Bad'], del_qc_var=False)

ds_sp2.clean.cleanup()
ds_sp2 = act.qc.arm.add_dqr_to_qc(ds_sp2)
ds_sp2.qcfilter.datafilter(rm_assessments=['Bad'], del_qc_var=False)

ds_mpl.clean.cleanup()
ds_mpl = act.qc.arm.add_dqr_to_qc(ds_mpl)
ds_mpl.qcfilter.datafilter(rm_assessments=['Bad'], del_qc_var=False)
```

Visualizing Data

We have all the datasets downloaded, let's start to visualize them in different ways using ACT. If you ever need a place to start with how to visualize data using ACT, check out the ACT Plotting Examples

```
# We can pass a dictionary to the display objects with multiple datasets
# So let's plot all this up!
display = act.plotting.TimeSeriesDisplay({'aps': ds_aps, 'mpl': ds_mpl, 'acsm': ds_acsm, '
                                          subplot_shape=(4,), figsize=(10,18))
```

Skip to main content

```
# Variable names of interest linear_depol_ratio, linear_depol_snr, backscatter_snr
display.plot('linear_depol_ratio', dsname='mpl', subplot_index=(0,), cb_friendly=True)
display.set_yrng([0, 3], subplot_index=(0,))

# APS Plot
display.plot('total_N_conc', dsname='aps', subplot_index=(1,))
display.day_night_background(dsname='aps', subplot_index=(1,))

# ACSM plot
display.plot('sulfate', dsname='acsm', subplot_index=(2,), label='sulfate')
display.plot('nitrate', dsname='acsm', subplot_index=(2,), label='nitrate')
display.plot('ammonium', dsname='acsm', subplot_index=(2,), label='ammonium')
display.plot('chloride', dsname='acsm', subplot_index=(2,), label='chloride')
display.plot('total_organics', dsname='acsm', subplot_index=(2,), label='total_organics')

display.day_night_background(dsname='acsm', subplot_index=(2,))

# SP2 Plot
display.plot('sp2_rbc_conc', dsname='sp2', subplot_index=(3,))
display.day_night_background(dsname='sp2', subplot_index=(3,))

plt.subplots_adjust(hspace=0.3)
plt.legend()
plt.savefig('./images/output.png')
plt.show()
```

Data Rose Plots

These plots display the data on a windrose-like plot to visualize directional dependencies in the data.

```
# We already should have the data loaded up so let's explore with some data roses
# First we need to combine data and to do that, we need to get it on the same time grid
ds_combined = xr.merge([ds_met.resample(time='30min').nearest(), ds_acsm.resample(time='30

# Plot out the data rose using the WindRose display object
display = act.plotting.WindRoseDisplay(ds_combined)
display.plot_data('wdir_vec_mean', 'wspd_vec_mean', 'sulfate', num_dirs=15, plot_type='lin
plt.show()
```

```
# First we need to combine data and to do that, we need to get it on the same time grid
ds_combined = xr.merge([ds_met.resample(time='1min').nearest(), ds_sp2.resample(time='1min

# Plot out the data rose using the WindRose display object
display = act.plotting.WindRoseDisplay(ds_combined)

# Let's try a different type of data rose that will show the mean Black Carbon Concentrati
# depending on wind direction and speed
display.plot_data('wdir_vec_mean', 'wspd_vec_mean', 'sp2_rbc_conc', num_dirs=15, plot_type
plt.show()
```

Checkout the area

The AMF was deployed at La Porte Municipal Airport. Check out the google map and see if this mapes sense!

Back to the visualizations!

Let's get back to checking out the other visualization features in ACT!

Histograms

```
# We do the same thing as before but call the DistributionDisplay class
display = act.plotting.DistributionDisplay(ds_aps)

# And then we can plot the data!  Note that we are passing a range into the
# histogram function to set the min/max range of the data
display.plot_stacked_bar_graph('total_N_conc', bins=20, hist_kwargs={'range': [0, 60]})
plt.show()
```

```
# We can create these plots in groups as well but we need to know
# how many there will be ahead of time for the shape
display = act.plotting.DistributionDisplay(ds_aps, figsize=(15, 15), subplot_shape=(6, 4))
groupby = display.group_by('hour')

# And then we can plot the data in groups!  The main issue is that it doesn't automaticall
# Annotate the group on the plot.  We're also setting the titile to blank to save space
groupby.plot_group('plot_stacked_bar_graph', None, field='total_N_conc', set_title='', bin

# We want these graphs to have the same axes, so we can easily run through
# each plot and modify the axes.  Right now, we can just hard code these in
for i in range(len(display.axes)):
    for j in range(len(display.axes[i])):
        display.axes[i, j].set_xlim([0, 60])
        display.axes[i, j].set_ylim([0, 15000])

plt.subplots_adjust(wspace=0.35)

plt.show()
```

Scatter Plots and Heatmaps

Let's plot up a comparison of the APS total concentration and the ACSM sulfates. Feel free to change the variables from the ACSM to experiment!

```
ds_combined = xr.merge([ds_aps.resample(time='30min').nearest(), ds_acsm.resample(time='30

# Plot out the data rose using the Distribution display object
display = act.plotting.DistributionDisplay(ds_combined)
display.plot_scatter('total_N_conc', 'sulfate', m_field='time')

plt.show()
```

```
# Let's try a heatmap with this as well!
display = act.plotting.DistributionDisplay(ds_combined, figsize=(12, 5), subplot_shape=(1,

display.plot_scatter('total_N_conc', 'sulfate', m_field='time', subplot_index=(0, 0))
display.plot_heatmap('total_N_conc', 'sulfate', subplot_index=(0, 1), x_bins=50, y_bins=50

plt.show()
```

```
# Let's try one last plot type with this dataset
# Violin plots!
display = act.plotting.DistributionDisplay(ds_acsm)

# And then we can plot the data!
display.plot_violin('sulfate', positions=[1.0])
display.plot_violin('nitrate', positions=[2.0])
display.plot_violin('ammonium', positions=[3.0])
display.plot_violin('chloride', positions=[4.0])
display.plot_violin('total_organics', positions=[5.0])

# Let's add some more information to the plots
# Update the tick information
display.axes[0].set_xticks([0.5, 1, 2, 3, 4, 5, 5.5])
display.axes[0].set_xticklabels(['',
                                 'Sulfate',
                                 'Nitrate',
                                 'Ammonium',
                                 'Chloride',
                                 'Total Organics',
                                 '']
                                )

plt.show()
```

Additional Features in ACT

If there's time to explore more features or if you want to on your own time, these are some of the many additional features that you might find useful in ACT

Skew-T Plots

```
# Let's set a longer time period
```

```
# SONDE
datastream = 'housondewnpnM1.b1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
result.sort()
ds_sonde = act.io.armfiles.read_netcdf(result[-1])

# Plot enhanced Skew-T plot
display = act.plotting.SkewTDisplay(ds_sonde)
display.plot_enhanced_skewt(color_field='alt')

plt.show()
```

Wind Roses

```
# Now we can plot up a wind rose of that entire month's worth of data
windrose = act.plotting.WindRoseDisplay(ds_met, figsize=(10,8))
windrose.plot('wdir_vec_mean', 'wspd_vec_mean', spd_bins=np.linspace(0, 10, 5))
windrose.axes[0].legend()
plt.show()
```

Present Weather Codes

See this example of how to plot up these present weather codes on your plots!

```
# Pass it to the function to decode it along with the variable name
ds_met = act.utils.inst_utils.decode_present_weather(ds_met, variable='pwd_pw_code_inst')

# We're going to print out the first 10 decoded values that weren't 0
# This shows the utility of also being able to use the built-in xarray
# features like where!
print(list(ds_met['pwd_pw_code_inst_decoded'].where(ds_met.pwd_pw_code_inst.compute() > 0,
```

Accumulating Precipitation

This example shows how to accumulate precipitation using the ACT utility and then overplot the PWD present weather codes

```
# Let's accumulate the precipitation data from the three different sensors in the MET Syst
# These instruments include a tipping bucket rain gauge, optical rain gauge, and a present
variables = ['tbrg_precip_total', 'org_precip_rate_mean', 'pwd_precip_rate_mean_1min']
for v in variables:
    ds_met = act.utils.data_utils.accumulate_precip(ds_met, v)
```

Skip to main content

```
display = act.plotting.TimeSeriesDisplay(ds_met, figsize=(8, 6))
for v in variables:
    display.plot(v + '_accumulated', label=v)

# Add a day/night background
display.day_night_background()

# Now we can decode the present weather codes (WMO codes)
ds_met = act.utils.inst_utils.decode_present_weather(ds_met, variable='pwd_pw_code_1hr')

# We're only going to plot up the code when it changes
# and if we plot it up, we will skip 2 hours so the plot
# is not busy and unreadable
ct = 0
ds = ds_met.where(ds_met.pwd_pw_code_1hr.compute() > 0, drop=True)
wx = ds['pwd_pw_code_1hr_decoded'].values
prev_wx = None
while ct < len(wx):
    if wx[ct] != prev_wx:
        # We can access the figure and axes through the display object
        display.axes[0].text(ds['time'].values[ct], -7.5, wx[ct], rotation=90, va='top')
    prev_wx = wx[ct]
    ct += 120
plt.subplots_adjust(bottom=0.20)
plt.legend()
plt.show()
```

Doppler Lidar Wind Retrievals

This will show you how you can process the doppler lidar PPI scans to produce **wind profiles** based on [illegible]

```
# We're going to use some test data that already exists within ACT
# Let's set a longer time period
startdate = '2022-07-16T21:00:00'
enddate = '2022-07-16T22:00:00'

# SONDE
datastream = 'houdlppiM1.b1'
result = act.discovery.download_data(username, token, datastream, startdate, enddate)
result.sort()


ds = act.io.armfiles.read_netcdf(result)
ds
# Returns the wind retrieval information in a new object by default
# Note that the default snr_threshold of 0.008 was too high for the first profile
# Reducing it to 0.002 makes it show up but the quality of the data is likely suspect.
ds_wind = act.retrievals.compute_winds_from_ppi(ds, snr_threshold=0.0001)

# Plot it up
display = act.plotting.TimeSeriesDisplay(ds_wind)
display.plot_barbs_from_spd_dir('wind_speed', 'wind_direction', invert_y_axis=False)

#Update the x-limits to make sure both wind profiles are shown
```

```
plt.show()
```

Mimic ARM Data Files

ARM's NetCDF files are based around what we call a data object definition or DOD. These DOD's essentially create the structure of the file and are what you see in the NetCDF file as the header. We can use this information to create an xarray object, filled with missing value, that one can populated with data and then write it out to a NetCDF file that looks exactly like an ARM file.

The user is able to set up the size of the datasets ahead of time by passing in the dimension sizes as shown below with `{'time': 1440}`

This could greatly streamline and improve the usability of PI-submitted datasets.

Note, that this does take some time for datastreams like the MET that have a lot of versions.

```
ds = act.io.armfiles.create_ds_from_arm_dod('ld.b1', {'time': 1440}, scalar_fill_dim='time

# Create some random data and set it to the variable in the obect like normal
ds['precip_rate'].values = np.random.rand(1440)
ds
```

```
ds['precip_rate'].plot()
```

Py-ART Basics

Contents

Overview

Within this notebook, we will cover:

1. General overview of Py-ART and its functionality
2. Reading data using Py-ART
3. An overview of the `pyart.Radar` object
4. Create a Plot of our Radar Data

Skip to main content

Prerequisites

Concepts	Importance	Notes
Intro to Cartopy	Helpful	Basic features
Matplotlib Basics	Helpful	Basic plotting
NumPy Basics	Helpful	Basic arrays

- **Time to learn**: 30 minutes

Imports

```
import os
import warnings

import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np


import pyart

warnings.filterwarnings('ignore')
```

```
## You are using the Python ARM Radar Toolkit (Py-ART), an open source
## library for working with weather radar data. Py-ART is partly
## supported by the U.S. Department of Energy as part of the Atmospheric
## Radiation Measurement (ARM) Climate Research Facility, an Office of
## Science user facility.
##
## If you use this software to prepare a publication, please cite:
##
##     JJ Helmus and SM Collis, JORS 2016, doi: 10.5334/jors.119
```

```
/Users/mgrover/miniforge3/envs/pyart-docs/lib/python3.10/site-packages/tqdm/auto.py:22: Tq
  from .autonotebook import tqdm as notebook_tqdm
```

An Overview of Py-ART

History of the Py-ART

- Development began to address the needs of ARM with the acquisition of a number of new scanning cloud and precipitation radar as part of the American Recovery Act.
- The project has since expanded to work with a variety of weather radars and a wider user base including radar researchers and climate modelers.
- The software has been released on GitHub as open source software under a BSD license. Runs on Linux, OS X. It also runs on Windows with more limited functionality.

What can PyART Do?

Py-ART can be used for a variety of tasks from basic plotting to more complex processing pipelines. Specific uses for Py-ART include:

- Reading radar data in a variety of file formats.
- Creating plots and visualization of radar data.
- Correcting radar moments while in antenna coordinates, such as:
  - Doppler unfolding/de-aliasing.
  - Attenuation correction.
  - Phase processing using a Linear Programming method.
- Mapping data from one or multiple radars onto a Cartesian grid.
- Performing retrievals.
- Writing radial and Cartesian data to NetCDF files.

Reading in Data Using Py-ART

The Sample Data - SAIL!

Our Radar 📡

Reading data in using `pyart.io.read`

Skip to main content

`pyart.io.read` can read a variety of different radar formats, such as Cf/Radial, LASSEN, and more. The documentation on what formats can be read by Py-ART can be found here:

- Py-ART IO Documentation

For most file formats listed on the page, using `pyart.io.read` should suffice since Py-ART has the ability to automatically detect the file format.

Let's check out what arguments arguments `pyart.io.read()` takes in!

```
pyart.io.read?
```

```
Signature: pyart.io.read(filename, use_rsl=False, **kwargs)
Docstring:
Read a radar file and return a radar object.

Additional parameters are passed to the underlying read_* function.

Parameters
----------
filename : str
    Name of radar file to read.
use_rsl : bool
    True will use the TRMM RSL library to read files which are supported
    both natively and by RSL. False will choose the native read function.
    RSL will always be used to read a file if it is not supported
    natively.

Other Parameters
----------------
field_names : dict, optional
    Dictionary mapping file data type names to radar field names. If a
    data type found in the file does not appear in this dictionary or has
    a value of None it will not be placed in the radar.fields dictionary.
    A value of None, the default, will use the mapping defined in the
    metadata configuration file.
additional_metadata : dict of dicts, optional
    Dictionary of dictionaries to retrieve metadata from during this read.
    This metadata is not used during any successive file reads unless
    explicitly included. A value of None, the default, will not
    introduct any addition metadata and the file specific or default
    metadata as specified by the metadata configuration file will be used.
file_field_names : bool, optional
    True to use the file data type names for the field names. If this
    case the field_names parameter is ignored. The field dictionary will
    likely only have a 'data' key, unless the fields are defined in
    `additional_metadata`.
exclude_fields : list or None, optional
    List of fields to exclude from the radar object. This is applied
    after the `file_field_names` and `field_names` parameters.
delay_field_loading : bool
    True to delay loading of field data from the file until the 'data'
    key in a particular field dictionary is accessed. In this case
    the field attribute of the returned Radar object will contain
    LazyLoadDict objects not dict objects. Not all file types support this
```

Skip to main content

```
Returns
-------
radar : Radar
    Radar object. A TypeError is raised if the format cannot be
    determined.
File:      ~/git_repos/pyart/pyart/io/auto_read.py
Type:      function
```

Let's use a sample data file from `pyart` - which is **cfradial** format.

When we read this in, we get a `pyart.Radar` object!

```
file = '../data/sample_sail_ppi.nc'
radar = pyart.io.read(file)
radar
```

```
<pyart.core.radar.Radar at 0x283fa3160>
```

Investigate the `pyart.Radar` object

Within this `pyart.Radar` object object are the actual data fields.

This is where data such as reflectivity and velocity are stored.

To see what fields are present we can add the fields and keys additions to the variable where the radar object is stored.

```
radar.fields.keys()
```

```
dict_keys(['corrected_velocity', 'corrected_reflectivity', 'corrected_differential_reflect
```

Extract a sample data field

The fields are stored in a dictionary, each containing coordinates, units and more. All can be accessed by just adding the fields addition to the radar object variable.

For an individual field, we add a string in brackets after the fields addition to see the contents of that field.

Let's take a look at `'corrected_reflectivity'`, which is a common field to investigate.

```
print(radar.fields['corrected_reflectivity'])
```

```
{'_FillValue': 1e+20, 'long_name': 'Corrected reflectivity', 'units': 'dBZ', 'standard_nam
  data=[[--, --, --, ..., --, --, --],
        [--, --, --, ..., --, --, --],
        [--, --, --, ..., --, --, --],
        ...,
        [12.25, 9.84000015258789, 14.210000038146973, ..., --, --, --],
        [11.5, 9.729999542236328, 11.75999927520752, ..., --, --, --],
        [11.069999694824219, 10.329999923706055, 10.050000190734863, ...,
         --, --, --]],
  mask=[[ True,  True,  True, ...,  True,  True,  True],
        [ True,  True,  True, ...,  True,  True,  True],
        [ True,  True,  True, ...,  True,  True,  True],
        ...,
        [False, False, False, ...,  True,  True,  True],
        [False, False, False, ...,  True,  True,  True],
        [False, False, False, ...,  True,  True,  True]],
  fill_value=1e+20)}
```

We can go even further in the dictionary and access the actual reflectivity data.

We use add `'data'` at the end, which will extract the **data array** (which is a masked numpy array) from the dictionary.

```
reflectivity = radar.fields['corrected_reflectivity']['data']
print(type(reflectivity), reflectivity)
```

```
<class 'numpy.ma.core.MaskedArray'> [[-- -- -- ... -- -- --]
 [-- -- -- ... -- -- --]
 [-- -- -- ... -- -- --]
 ...
 [12.25 9.84000015258789 14.210000038146973 ... -- -- --]
 [11.5 9.729999542236328 11.75999927520752 ... -- -- --]
 [11.069999694824219 10.329999923706055 10.050000190734863 ... -- -- --]]
```

Lets' check the size of this array...

```
reflectivity.shape
```

```
(9013, 668)
```

This reflectivity data array, numpy array, is a two-dimensional array with dimensions:

- Gates (number of samples away from the radar)
- Rays (direction around the radar)

```
print(radar.nrays, radar.ngates)
```

Skip to main content

```
9013 668
```

If we wanted to look the 300th ray, at the second gate, we would use something like the following:

```
print(reflectivity[300, 2])
```

```
9.369999885559082
```

Plotting our Radar Data

An Overview of Py-ART Plotting Utilities

Now that we have loaded the data and inspected it, the next logical thing to do is to visualize the data! Py-ART's visualization functionality is done through the objects in the pyart.graph module.

In Py-ART there are 4 primary visualization classes in pyart.graph:

- RadarDisplay
- RadarMapDisplay
- AirborneRadarDisplay

Plotting grid data

- GridMapDisplay

Use the RadarMapDisplay with our data

For the this example, we will be using `RadarMapDisplay`, using Cartopy to deal with geographic coordinates.

We start by creating a figure first.

```
fig = plt.figure(figsize=[10, 10])
```

```
<Figure size 1000x1000 with 0 Axes>
```

Once we have a figure, let's add our `RadarMapDisplay`

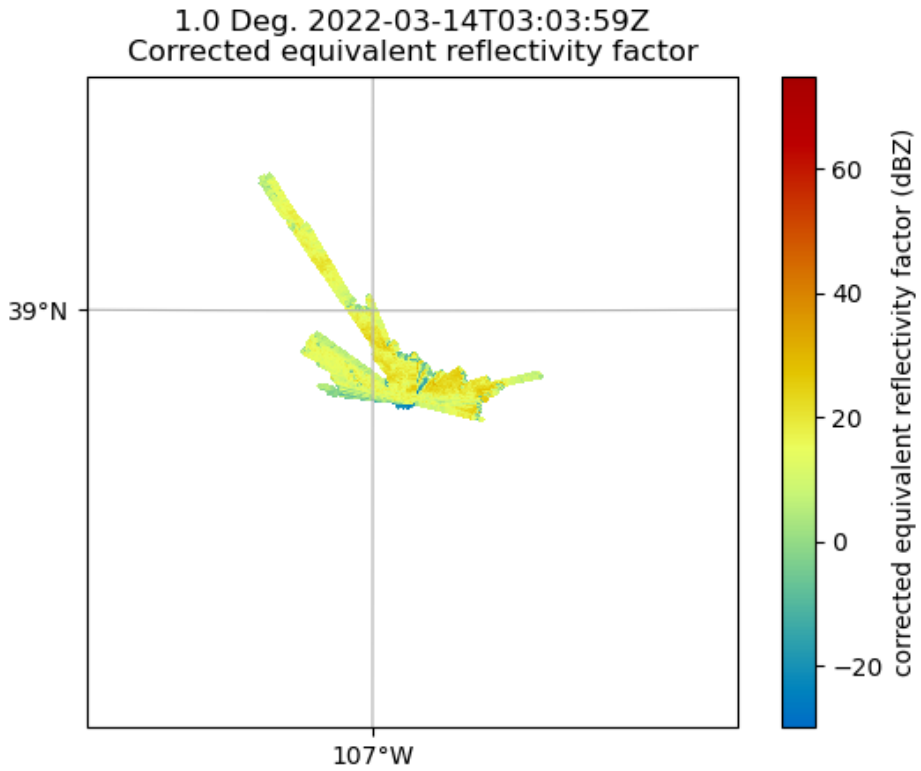
Skip to main content

```
fig = plt.figure(figsize=[10, 10])
display = pyart.graph.RadarMapDisplay(radar)
```

```
<Figure size 1000x1000 with 0 Axes>
```

Adding our map display without specifying a field to plot **won't do anything** we need to specifically add a field to field using `.plot_ppi_map()`, which creates a Plan Position Indicator (PPI) plot.

```
display.plot_ppi_map('corrected_reflectivity')
```



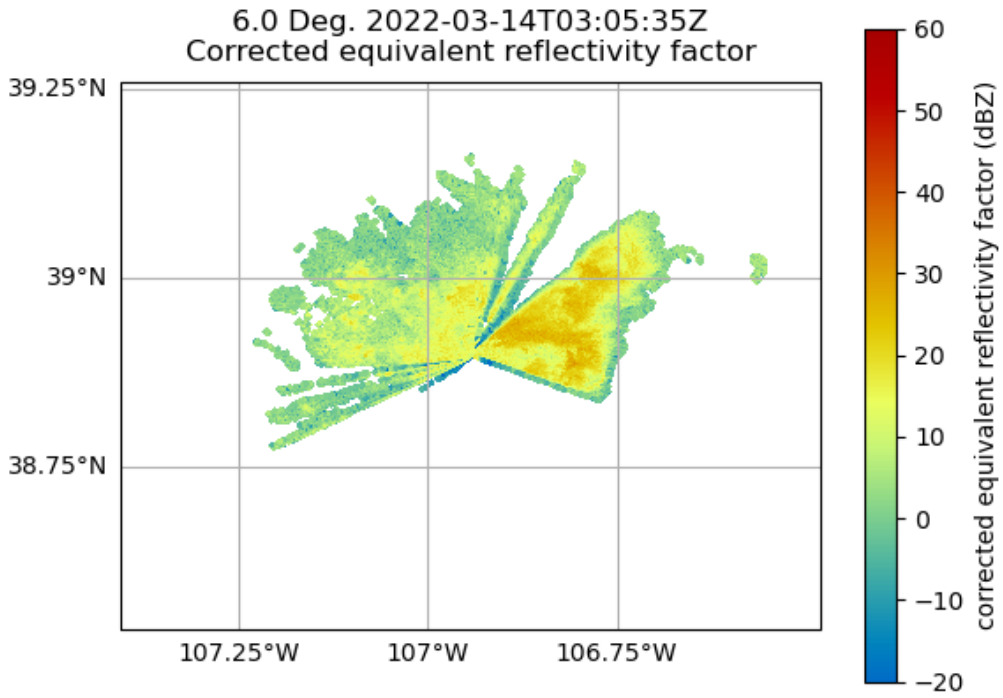
By default, it will plot the elevation scan, the the default colormap from `Matplotlib`... let's customize!

We add the following arguements:

- `sweep=3` - The fourth elevation scan (since we are using Python indexing)
- `vmin=-20` - Minimum value for our plotted field/colorbar
- `vmax=60` - Maximum value for our plotted field/colorbar
- `projection=ccrs.PlateCarree()` - Cartopy latitude/longitude coordinate system
- `cmap='pyart_HomeyerRainbow'` - Colormap to use, selecting one provided by PyART of
- `lat_lines` - Which lines to plot for latitude
- `lon_lines` - Which liens to plot for longitude

Skip to main content

```
###### fig = plt.figure(figsize=[12, 8])
display = pyart.graph.RadarMapDisplay(radar)
display.plot_ppi_map('corrected_reflectivity',
                     sweep=3,
                     vmin=-20,
                     vmax=60,
                     lat_lines = np.arange(38, 39.5, .25),
                     lon_lines = np.arange(-107.5, -106.5, .25),
                     projection=ccrs.PlateCarree(),
                     cmap='pyart_HomeyerRainbow')
plt.savefig("sample-ppi-map.png", dpi=300)
```



You can change many parameters in the graph by changing the arguments to plot_ppi_map. As you can recall from earlier. simply view these arguments in a Jupyter notebook by typing:

```
display.plot_ppi_map?
```

```
Signature:
display.plot_ppi_map(
    field,
    sweep=0,
    mask_tuple=None,
    vmin=None,
    vmax=None,
    cmap=None,
    norm=None,
    mask_outside=False,
    title=None,
    title_flag=True,
    colorbar_flag=True,
    colorbar_label=None,
```

Skip to main content

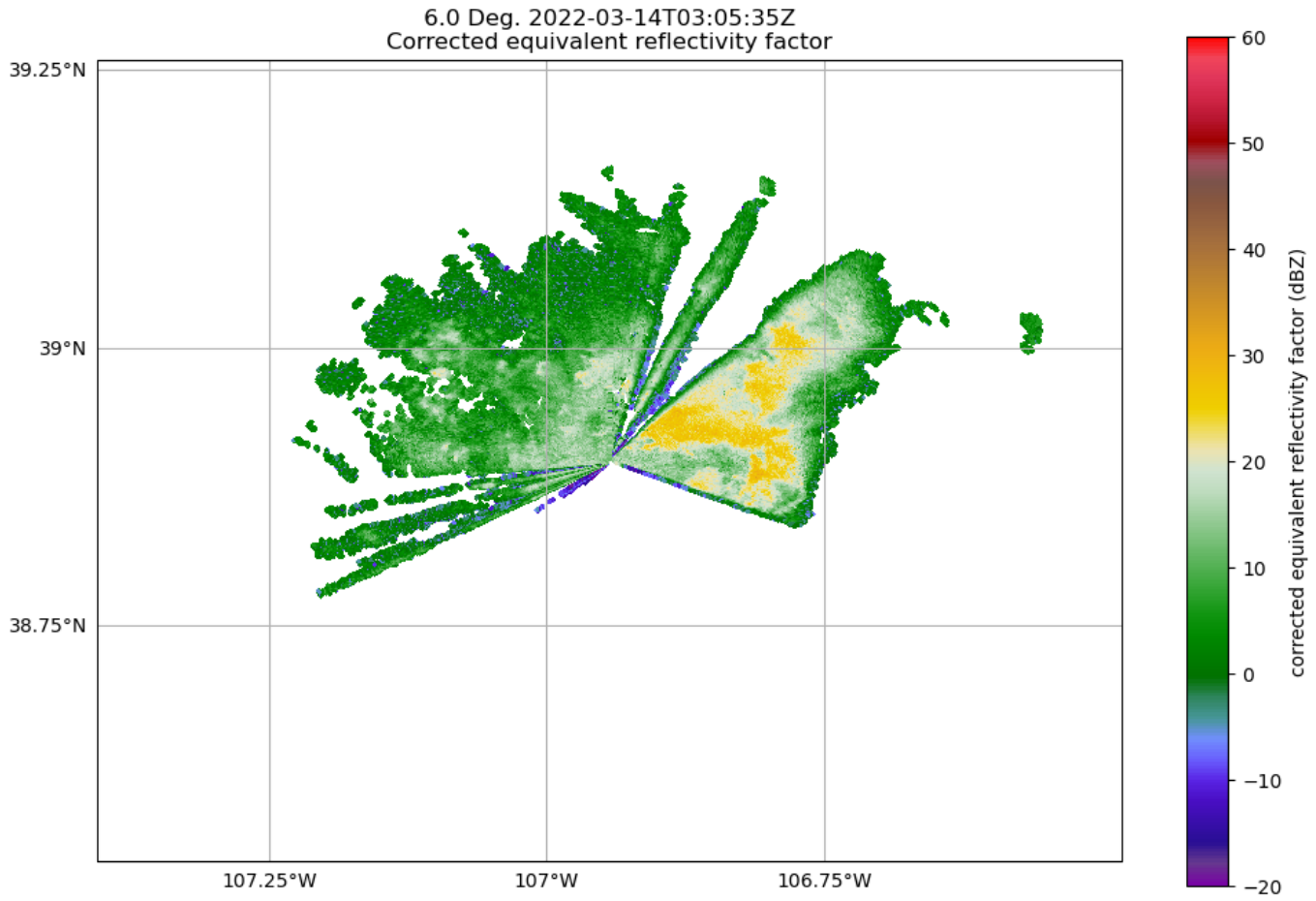
```
    lat_lines=None,
    lon_lines=None,
    projection=None,
    min_lon=None,
    max_lon=None,
    min_lat=None,
    max_lat=None,
    width=None,
    height=None,
    lon_0=None,
    lat_0=None,
    resolution='110m',
    shapefile=None,
    shapefile_kwargs=None,
    edges=True,
    gatefilter=None,
    filter_transitions=True,
    embellish=True,
    raster=False,
    ticks=None,
    ticklabs=None,
    alpha=None,
    edgecolors='face',
    **kwargs,
)
Docstring:
Plot a PPI volume sweep onto a geographic map.

Parameters
----------
field : str
    Field to plot.
sweep : int, optional
    Sweep number to plot.

Other Parameters
----------------
mask_tuple : (str, float)
    Tuple containing the field name and value below which to mask
    field prior to plotting, for example to mask all data where
    NCP < 0.5 set mask_tuple to ['NCP', 0.5]. None performs no masking.
vmin : float
    Luminance minimum value, None for default value.
    Parameter is ignored is norm is not None.
vmax : float
    Luminance maximum value, None for default value.
    Parameter is ignored is norm is not None.
norm : Normalize or None, optional
    matplotlib Normalize instance used to scale luminance data. If not
    None the vmax and vmin parameters are ignored. If None, vmin and
    vmax are used for luminance scaling.
cmap : str or None
    Matplotlib colormap name. None will use the default colormap for
    the field being plotted as specified by the Py-ART configuration.
mask_outside : bool
    True to mask data outside of vmin, vmax. False performs no
    masking.
title : str
    Title to label plot with, None to use default title generated from
    the field and tilt parameters. Parameter is ignored if title_flag
    is False.
```

```
colorbar_flag : bool
    True to add a colorbar with label to the axis. False leaves off
    the colorbar.
ticks : array
    Colorbar custom tick label locations.
ticklabs : array
    Colorbar custom tick labels.
colorbar_label : str
    Colorbar label, None will use a default label generated from the
    field information.
ax : Cartopy GeoAxes instance
    If None, create GeoAxes instance using other keyword info.
    If provided, ax must have a Cartopy crs projection and projection
    kwarg below is ignored.
fig : Figure
    Figure to add the colorbar to. None will use the current figure.
lat_lines, lon_lines : array or None
    Locations at which to draw latitude and longitude lines.
    None will use default values which are resonable for maps of
    North America.
projection : cartopy.crs class
    Map projection supported by cartopy. Used for all subsequent calls
    to the GeoAxes object generated. Defaults to LambertConformal
    centered on radar.
min_lat, max_lat, min_lon, max_lon : float
    Latitude and longitude ranges for the map projection region in
    degrees.
width, height : float
    Width and height of map domain in meters.
    Only this set of parameters or the previous set of parameters
    (min_lat, max_lat, min_lon, max_lon) should be specified.
    If neither set is specified then the map domain will be determined
    from the extend of the radar gate locations.
shapefile : str
    Filename for a shapefile to add to map.
shapefile_kwargs : dict
    Key word arguments used to format shapefile. Projection defaults
    to lat lon (cartopy.crs.PlateCarree())
resolution : '10m', '50m', '110m'.
    Resolution of NaturalEarthFeatures to use. See Cartopy
    documentation for details.
gatefilter : GateFilter
    GateFilter instance. None will result in no gatefilter mask being
    applied to data.
filter_transitions : bool
    True to remove rays where the antenna was in transition between
    sweeps from the plot. False will include these rays in the plot.
    No rays are filtered when the antenna_transition attribute of the
    underlying radar is not present.
edges : bool
    True will interpolate and extrapolate the gate edges from the
    range, azimuth and elevations in the radar, treating these
    as specifying the center of each gate. False treats these
    coordinates themselved as the gate edges, resulting in a plot
    in which the last gate in each ray and the entire last ray are not
    not plotted.
embellish: bool
    True by default. Set to False to supress drawing of coastlines
    etc.. Use for speedup when specifying shapefiles.
    Note that lat lon labels only work with certain projections.
raster : bool
```

```
    high resolution data over large areas. Be sure to set the dpi
    of the plot for your application if you save it as a vector format
    (i.e., pdf, eps, svg).
alpha : float or None
    Set the alpha tranparency of the radar plot. Useful for
    overplotting radar over other datasets.
edgecolor : str
    Set the behavior of the edges of the pixels, by default
    it will color them the same as the pixels (faces).
**kwargs : additional keyword arguments to pass to pcolormesh.
File:       ~/git_repos/pyart/pyart/graph/radarmapdisplay.py
Type:       method
```

For example, let's change the colormap to something different

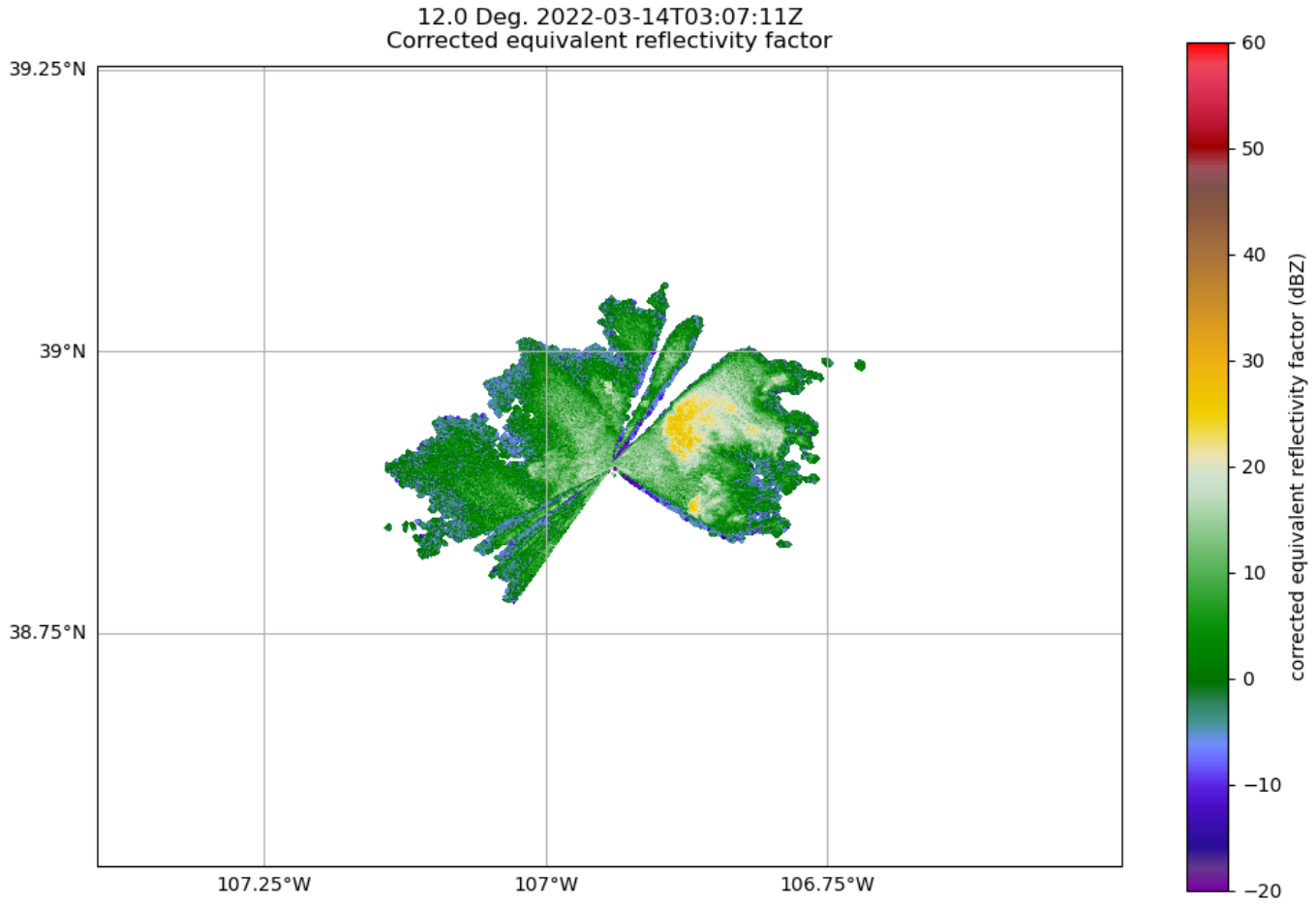
```
fig = plt.figure(figsize=[12, 8])
display = pyart.graph.RadarMapDisplay(radar)
display.plot_ppi_map('corrected_reflectivity',
                     sweep=3,
                     vmin=-20,
                     vmax=60,
                     projection=ccrs.PlateCarree(),
                     lat_lines = np.arange(38, 39.5, .25),
                     lon_lines = np.arange(-107.5, -106.5, .25),
                     cmap='pyart_Carbone42')
plt.show()
```



Skip to main content

Or, let's view a different elevation scan! To do this, change the sweep parameter in the plot_ppi_map function.

```
fig = plt.figure(figsize=[12, 8])
display = pyart.graph.RadarMapDisplay(radar)
display.plot_ppi_map('corrected_reflectivity',
                     sweep=6,
                     vmin=-20,
                     vmax=60,
                     lat_lines = np.arange(38, 39.5, .25),
                     lon_lines = np.arange(-107.5, -106.5, .25),
                     projection=ccrs.PlateCarree(),
                     cmap='pyart_Carbone42')
plt.show()
```



Plotting an RHI

Another common plot that is requested by the radar community is a **Range Height Indicator (RHI)** Plot.

Fortunately, Py-ART has a utility to help us create one of these from our radar!

Skip to main content

Read in an RHI file

During this same time period during SAIL, the ARM program collected RHI scans, which provide a vertical cross section through the preciptiation! Let's read in one of those files. The IO line is the same!

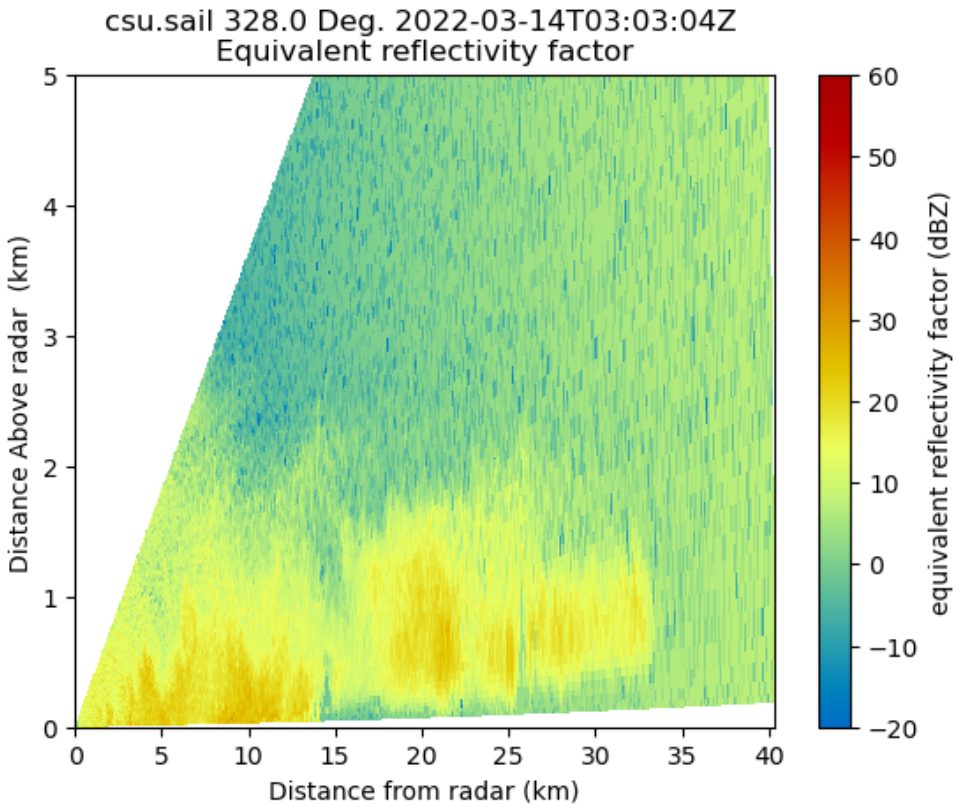
```
rhi_file = '../data/sample_sail_rhi.nc'
rhi_radar = pyart.io.read(rhi_file)
```

Plot our RHI

We want to use the `RadarDisplay` here to visualize, using the reflectivity field (`DBZ`)

Note - this is uncorrected data, so be sure take caution working with this

```
radar = pyart.graph.RadarDisplay(rhi_radar)
radar.plot("DBZ", vmin=-20, vmax=60,)
plt.ylim(0, 5)
plt.savefig("sample-rhi.png", dpi=300)
```



Skip to main content

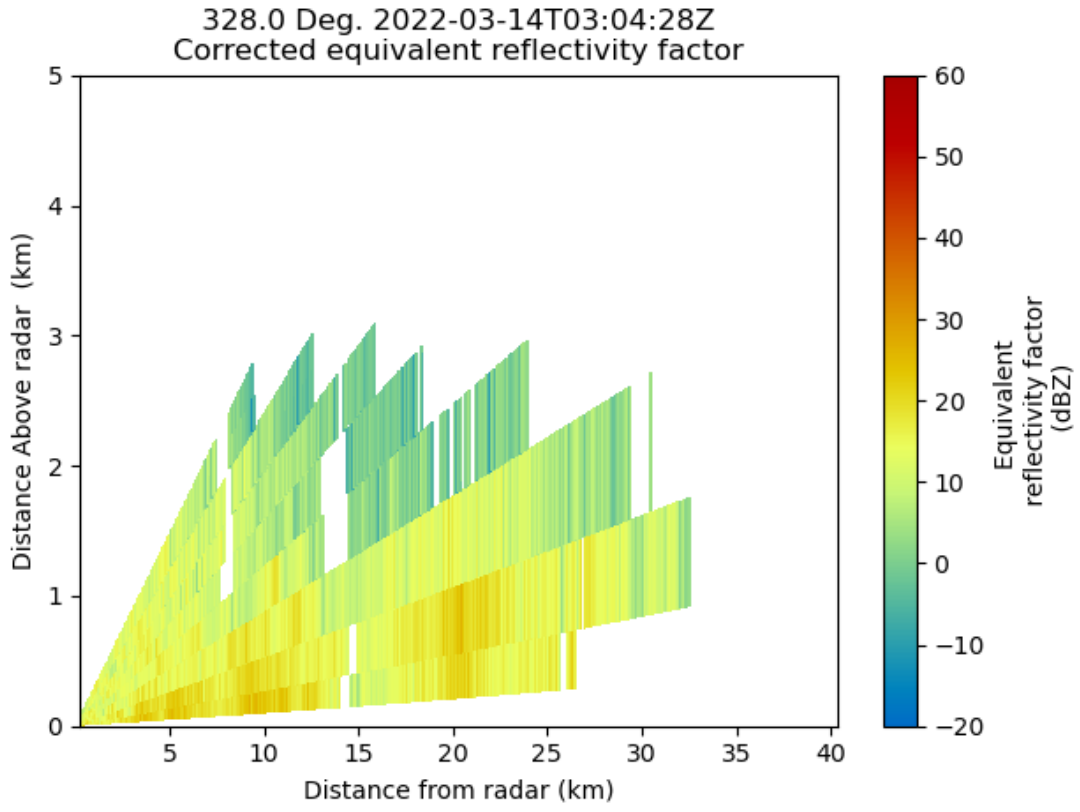
Add a “Pseudo-RHI” from our PPI data

But let’s say we wanted to compare the vertical resolution we get from an RHI, compared to PPI… we can do this with Py-ART!

```
# Load our PPI data back in
file = '../data/sample_sail_ppi.nc'
radar = pyart.io.read(file)
radar

# Create a cross section at our 334 degree azimuth
xsect = pyart.util.cross_section_ppi(radar, [328])
```

Now, notice how coarse the resolution of the precipitation region!

```
colorbar_label = 'Equivalent \n reflectivity factor \n (dBZ)'
display = pyart.graph.RadarDisplay(xsect)
display.plot('corrected_reflectivity', 0, vmin=-20, vmax=60, colorbar_label=colorbar_label
plt.ylim(0, 5)
plt.tight_layout()
```



Skip to main content

Summary

Within this notebook, we covered the basics of working with radar data using `pyart`, including:

- Reading in a file using `pyart.io`
- Investigating the `Radar` object
- Visualizing radar data using the `RadarMapDisplay`

What's Next

In the next few notebooks, we walk through gridding radar data, applying data cleaning methods, and advanced visualization methods!

Resources and References

Py-ART essentials links:

- Landing page
- Examples
- Source Code
- Mailing list
- Issue Tracker