UBC Smoke Forecast Data Curation and Distribution

Arleth Salinas and Valerio Pascucci
a 'WIRED Global Center' & 'National Science Data Fabric' collaboration

2024-07-28

Table of contents

Preface

Welcome to NSDF's UBC Firesmoke Data Curation website. Here we describe the data curation process of UBC's Smoke Forecast datasets. We inform readers of the challenges and solutions we found when repurposing these short term datasets into a long term dataset.

It is important to note that although we will present the data curation process in a linear fashion here, it was not a linear process. Rather, the process was cyclical, and at each iteration we introduced new improvements.

Navigation and Webpage Information

This webpage is produced using Quart.

You can find the source code for this page at **TODO**.

All files or directories in this website are hosted at our GitHub repository, unless otherwise specified.

All code blocks contain annotations, hover over the numbers on the right hand side to see the accompanying annotation.

```
Hover over me ----> (1)
```

(1) I'm an annotation.

1 IDX File Demo

WIRED Global Center +
National Science Data Fabric
Jupyter notebook created by
Arleth Z. Salinas, and Valerio
Pascucci

1.0.1 WIRED Global Center + National Science Data Fabric collaboration: Jupyter Notebook using 3 years of smoke forecast data over US and Canada stored in the cloud and dsitributed via regular internet connection.

Data source: BlueSky Canada Smoke Forecast

1.1 This notebook provide the instructions on how to read UBC firesmoke data from `firsmoke_metadata.nc` using xarray and the OpenVisus xarray backend.

Dashboard visible here: http://chpc3.nationalsciencedatafabric.org:9988/dashboards

1.2 Step 1: Importing the libraries

1.2.1 Please be sure to have libraries installed

```
# for numerical work
import numpy as np

# for accessing file system
import os

# for loading netcdf files, for metadata
```

```
import xarray as xr
# for connecting OpenVisus framework to xarray
# from https://github.com/sci-visus/openvisuspy,
from openvisuspy.xarray_backend import OpenVisusBackendEntrypoint

# Used for processing netCDF time data
import time
import datetime
import requests
# Used for indexing via metadata
import pandas as pd

# for plotting
import matplotlib.pyplot as plt
import cartopy.crs as ccrs


#Stores the OpenVisus cache in the local direcrtory
import os
os.environ["VISUS_CACHE"]="./visus_cache_can_be_erased"
os.environ['CURL_CA_BUNDLE'] = ''
```

1.3 Step 2: Reading the data & metadata from file

1.3.1 In this section, we load our data using `xr.open_dataset`.

```
# path to tiny NetCDF
url = 'https://github.com/sci-visus/NSDF-WIRED/raw/main/data/firesmoke_metadata.nc'

# Download the file using requests
response = requests.get(url)
local_netcdf = 'firesmoke_metadata.nc'
with open(local_netcdf, 'wb') as f:
    f.write(response.content)

# open tiny netcdf with xarray and OpenVisus backend
ds = xr.open_dataset(local_netcdf, engine=OpenVisusBackendEntrypoint)
```

```
ov.LoadDataset(http://atlantis.sci.utah.edu/mod_visus?dataset=UBC_fire_smoke_BSC&cached=1)
```

```
PM25
Adding field  PM25 shape  [27357, 381, 1081, 21] dtype  float32 labels  ['time', 'ROW', 'COL
```

```
ds
```

```
<xarray.Dataset>
Dimensions:            (time: 27357, ROW: 381, COL: 1081, resolution: 21,
                        VAR: 1, DATE-TIME: 2)
Dimensions without coordinates: time, ROW, COL, resolution, VAR, DATE-TIME
Data variables:
    PM25               (time, ROW, COL, resolution) float32 ...
    TFLAG              (time, VAR, DATE-TIME) int32 ...
    wrf_arw_init_time  (time, VAR, DATE-TIME) int32 ...
    resampled          (time) bool ...
    CDATE              (time) int32 ...
    CTIME              (time) int32 ...
    WDATE              (time) int32 ...
    WTIME              (time) int32 ...
    SDATE              (time) int32 ...
    STIME              (time) int32 ...
Attributes: (12/28)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                    ...
    EXEC_ID:        ????????????????                                         ...
    FTYPE:          1
    TSTEP:          10000
    NTHIK:          1
    NCOLS:          1081
    ...             ...
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                       ...
    HISTORY:
    idx_url:        http://atlantis.sci.utah.edu/mod_visus?dataset=UBC_fire_s...
```

1.3.1.1 Data Variables Description

Attribute	Description
PM25	The concentration of particulate matter (PM2.5) for each time step, layer, row, and column in the spatial grid.
TFLAG	The date and time of each data point.

Attribute	Description
wrf_arw_init_time	The time at which this prediction's weather forecast was initiated.
resampled	Whether this timestamp was resampled from a 381x1041 to 381x1081 grid or not.
CDATE	The creation date of the data point, in YYYYDDD format.
CTIME	The creation time of the data point, in HHMMSS format.
WDATE	The date for which the weather forecast is initiated, in YYYYDDD format.
WTIME	The time for which the weather forecast is initiated, in HHMMSS format.
SDATE	The date for which the smoke forecast is initiated, in YYYYDDD format.
STIME	The time for which the weather forecast is initiated, in HHMMSS format.

1.4 Step 2.5, Calculate derived metadata using original metadata above to create coordinates

1.4.1 This is required to allow for indexing of data via metadata

1.4.1.1 Calculate latitude and longitude grid

```
# Get metadata to compute lon and lat
xorig = ds.XORIG
yorig = ds.YORIG
xcell = ds.XCELL
ycell = ds.YCELL
ncols = ds.NCOLS
nrows = ds.NROWS

longitude = np.linspace(xorig, xorig + xcell * (ncols - 1), ncols)
latitude = np.linspace(yorig, yorig + ycell * (nrows - 1), nrows)

print("Size of longitude & latitude arrays:")
print(f'np.size(longitude) = {np.size(longitude)}')
print(f'np.size(latitude) = {np.size(latitude)}\n')
print("Min & Max of longitude and latitude arrays:")
print(f'longitude: min = {np.min(longitude)}, max = {np.max(longitude)}')
print(f'latitude: min = {np.min(latitude)}, max = {np.max(latitude)}')
```

```
Size of longitude & latitude arrays:
np.size(longitude) = 1081
np.size(latitude) = 381
```

```
Min & Max of longitude and latitude arrays:
longitude: min = -160.0, max = -51.99999839067459
latitude: min = 32.0, max = 70.00000056624413
```

1.4.1.2 Using calculated latitude and longitude, create coordinates allowing for indexing data using lat/lon

```
# Create coordinates for lat and lon (credit: Aashish Panta)
ds.coords['lat'] = ('ROW', latitude)
ds.coords['lon'] = ('COL', longitude)

# Replace col and row dimensions with newly calculated lon and lat arrays (credit: Aashish Pa
ds = ds.swap_dims({'COL': 'lon', 'ROW': 'lat'})
```

1.4.1.3 Create coordinates allowing for indexing data using timestamp

1.4.1.3.1 First, convert tflags to timestamps that are compatible with xarray

```
def parse_tflag(tflag):
    """
    Return the tflag as a datetime object
    :param list tflag: a list of two int32, the 1st representing date and 2nd representing ti
    """
    # obtain year and day of year from tflag[0] (date)
    date = int(tflag[0])
    year = date // 1000 # first 4 digits of tflag[0]
    day_of_year = date % 1000 # last 3 digits of tflag[0]

    # create datetime object representing date
    final_date = datetime.datetime(year, 1, 1) + datetime.timedelta(days=day_of_year - 1)

    # obtain hour, mins, and secs from tflag[1] (time)
    time = int(tflag[1])
    hours = time // 10000 # first 2 digits of tflag[1]
    minutes = (time % 10000) // 100 # 3rd and 4th digits of tflag[1]
    seconds = time % 100  # last 2 digits of tflag[1]

    # create final datetime object
```

```
    full_datetime = datetime.datetime(year, final_date.month, final_date.day, hours, minutes
    return full_datetime
```

1.4.1.3.2 Return an array of the tflags as pandas timestamps

```
# get all tflags
tflag_values = ds['TFLAG'].values

# to store pandas timestamps
timestamps = []

# convert all tflags to pandas timestamps, store in timestamps list
for tflag in tflag_values:
    timestamps.append(pd.Timestamp(parse_tflag(tflag[0])))

# check out the first 3 timestamps
timestamps[0:3]
```

```
[Timestamp('2021-03-04 00:00:00'),
 Timestamp('2021-03-04 01:00:00'),
 Timestamp('2021-03-04 02:00:00')]
```

```
# set coordinates to each timestep with these pandas timestamps
ds.coords['time'] = ('time', timestamps)
```

1.4.1.4 The timestamps may not be intuitive. The following utility function returns the desired pandas timestamp based on your date and time of interest.

1.4.1.4.1 When you index the data at a desired time, use this function to get the timestamp you need to index.

```
def get_timestamp(year, month, day, hour):
    """
    return a pandas timestamp using the given date-time arguments
    :param int year: year
    :param int month: month
    :param int day: day
    :param int hour: hour
    """
```

```
    # Convert year, month, day, and hour to a datetime object
    full_datetime = datetime.datetime(year, month, day, hour)

    # Extract components from the datetime object
    year = full_datetime.year
    day_of_year = full_datetime.timetuple().tm_yday
    hours = full_datetime.hour
    minutes = full_datetime.minute
    seconds = full_datetime.second

    # Compute tflag[0] and tflag[1]
    tflag0 = year * 1000 + day_of_year
    tflag1 = hours * 10000 + minutes * 100 + seconds

    # Return the Pandas Timestamp object
    return pd.Timestamp(full_datetime)
```

1.5 Step 3: Select a `data_slice`

1.5.1 This section shows you how to load the data you want.

1.5.1.1 You can index the data using indices, timestamps*, latitude & longitude, and by desired resolution**.

*Not setting any time means the first timestep available is selected. **Not setting quality means full data resolution is selected.

1.5.1.1.1 In this case, let's get all available firesmoke data for March 5, 2021 00:00:00 and the time and date for which it's weather and smoke forecast were initiated.

```
# select timestamp
my_timestamp = get_timestamp(2021, 3, 5, 0)

# select resolution, let's use full resolution since data isn't too big at one time slice
# data resolution can be -19 for lowest res and 0 for highest res
data_resolution = 0

# get PM25 values and provide 4 values, the colons mean select all lat and lon indices
data_array_at_time = ds['PM25'].loc[my_timestamp, :, :, data_resolution]
```

```
# the metadata specifying weather and smoke forecast initialization times
resampled = ds['resampled'].loc[my_timestamp]
sdate = ds['SDATE'].loc[my_timestamp]
stime = ds['STIME'].loc[my_timestamp]
wdate = ds['WDATE'].loc[my_timestamp]
wtime = ds['WTIME'].loc[my_timestamp]

# notice, to access the data, you must append ".values" to the data array we got above
print(f'timestamp: {my_timestamp}')
print(f'resampled: {resampled.values} (boolean)')
print(f'SDATE is {sdate.values} (YYYYDDD)')
print(f'STIME is {stime.values} (HHMMSS)')
print(f'WDATE is {wdate.values} (YYYYDDD)')
print(f'WTIME is {wtime.values} (HHMMSS)')
print(f'shape of data_array_at_time.values = {np.shape(data_array_at_time.values)}')
```

```
timestamp: 2021-03-05 00:00:00
resampled: True (boolean)
SDATE is 2021063 (YYYYDDD)
STIME is 210000 (HHMMSS)
WDATE is 2021063 (YYYYDDD)
WTIME is 204413 (HHMMSS)
Using Max Resolution:  20
Time: 24, max_resolution: 20, logic_box=(0, 1081, 0, 381), field: PM25
shape of data_array_at_time.values = (381, 1081)
```

1.5.1.1.2 Perhaps we want to slice a specific latitude longitude range from our `data_array_at_time`, for example, latitude range `[35, 50]` and longitude range `[-140, -80]`. Let's do that below.

```
# # define range for latitude and longitude to use
min_lat = 35
max_lat = 50
min_lon = -140
max_lon = -80

# get PM25 values and provide 4 values, but this time at our desired ranges
data_array_at_latlon = ds['PM25'].loc[my_timestamp, min_lat:max_lat, min_lon:max_lon, data_re

# notice, to access the data, you must append ".values" to the data array we got above
```

```
print(f'timestamp: {my_timestamp}')
print(f'shape of data_array_at_time.values = {np.shape(data_array_at_latlon.values)}')
```

```
timestamp: 2021-03-05 00:00:00
Using Max Resolution:  20
Time: 24, max_resolution: 20, logic_box=(200, 800, 30, 180), field: PM25
shape of data_array_at_time.values = (150, 600)
```

We show how to obtain this attribute information for a time step of one's choice, let's use

1.5.1.2 The following are the max and min timestamps, lon/lat values, and data resolutions you can index by

1.5.1.2.1 Be sure you index within the data range, otherwise you may get errors since no data exists outside these ranges!

```
# NOTE: there is one dummy date, ignore ds['time'].values[-1]
print(f"earliest valid timestamp is: {ds['time'].values[0]}")
print(f"latest valid timestamp is: {ds['time'].values[-2]}\n")

print(f"valid longitude range is: {ds['lon'].values[0]}, {ds['lon'].values[-1]}")
print(f"valid latitude range is: {ds['lat'].values[0]}, {ds['lat'].values[-1]}\n")

print(f"valid data resolutions range is: [-19, 0]")
```

```
earliest valid timestamp is: 2021-03-04T00:00:00.000000000
latest valid timestamp is: 2024-06-27T22:00:00.000000000

valid longitude range is: -160.0, -51.99999839067459
valid latitude range is: 32.0, 70.00000056624413

valid data resolutions range is: [-19, 0]
```

1.6 Step 4: Visualize `data_slice`

1.6.1 One can visualize the data either by:

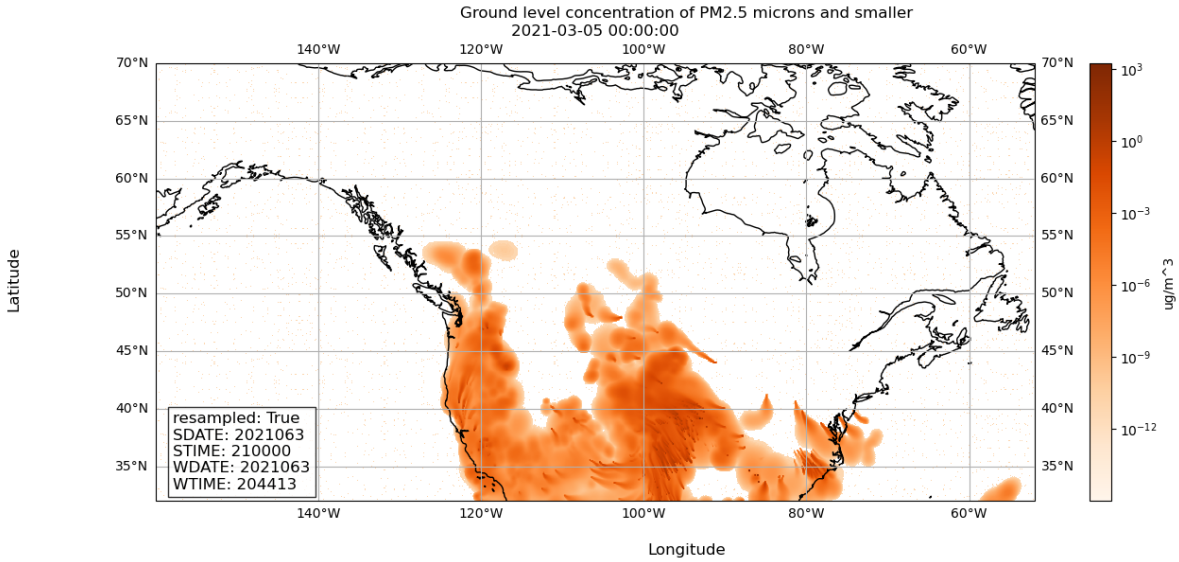
1.6.2 1. Get the values from your `data_array_at_time` and plot using your favorite python visualization library. We'll use matplotlib.

1.6.2.1 2. Use xarray's built in plotting function (not recommended, as it is not robust)

Here we plot `data_array_at_time` with matplotlib and its basemap extenstion to add geographic context.

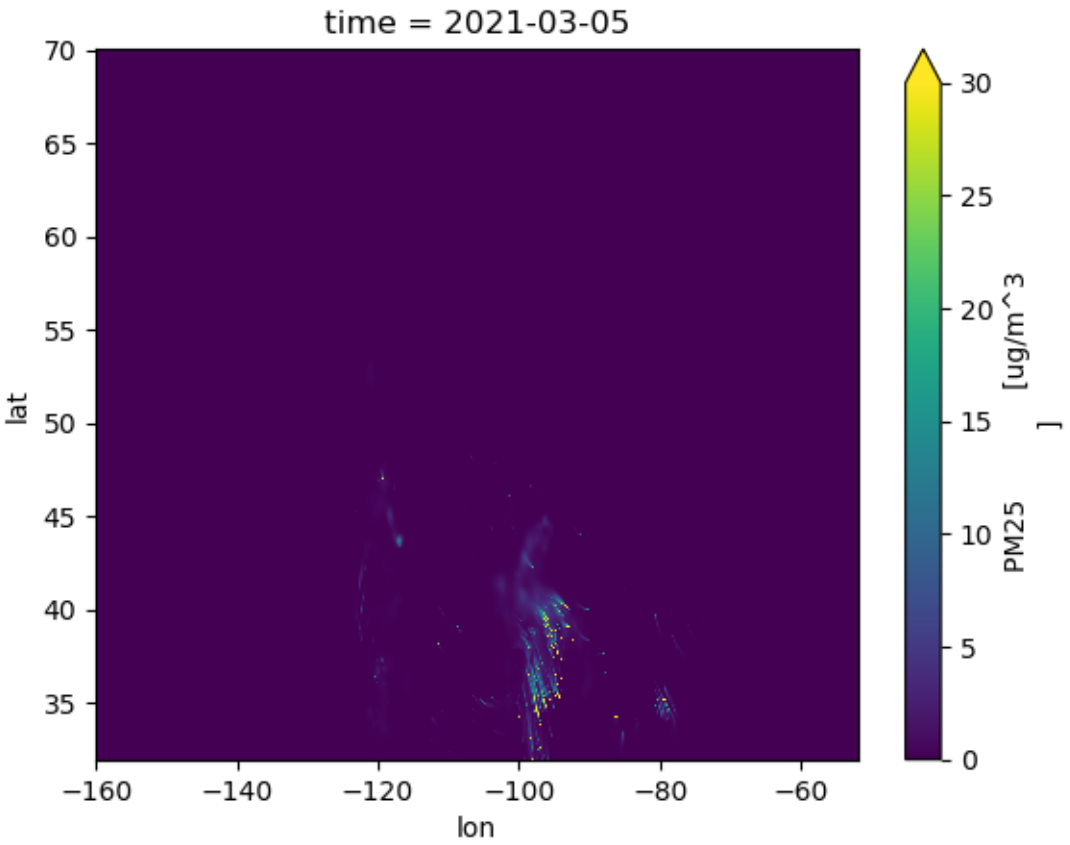
```
# Let's use matplotlib's imshow, since our data is on a grid
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html

# Initialize a figure and plot, so we can customize figure and plot of data
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html
# ref: https://scitools.org.uk/cartopy/docs/latest/getting_started/index.html
my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.PlateCarree())

# Let's set some parameters to get the visualization we want
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html

# color PM25 values on a log scale, since values are small
my_norm = "log"
# this will number our x and y axes based on the longitude latitude range
my_extent = [np.min(longitude), np.max(longitude), np.min(latitude), np.max(latitude)]
# ensure the aspect ratio of our plot fits all data, matplotlib can does this automatically
my_aspect = 'auto'
# tell matplotlib, our origin is the lower-left corner
my_origin = 'lower'
# select a colormap for our plot and the color bar on the right
my_cmap = 'Oranges'

# create our plot using imshow
plot = my_plt.imshow(data_array_at_time.values, norm=my_norm, extent=my_extent,
          aspect=my_aspect, origin=my_origin, cmap=my_cmap)

# draw coastlines
my_plt.coastlines()

# draw latitude longitude lines
```

```
# ref: https://scitools.org.uk/cartopy/docs/latest/gallery/gridlines_and_labels/gridliner.htm
my_plt.gridlines(draw_labels=True)

# add a colorbar to our figure, based on the plot we just made above
my_fig.colorbar(plot,location='right', label='ug/m^3')

# Add metadata as text annotations
metadata_text = (
    f'resampled: {resampled.values}\n'
    f'SDATE: {sdate.values}\n'
    f'STIME: {stime.values}\n'
    f'WDATE: {wdate.values}\n'
    f'WTIME: {wtime.values}'
)

# Place metadata text on the plot
my_plt.text(0.02, 0.02, metadata_text, transform=my_plt.transAxes,
            fontsize=12, verticalalignment='bottom', bbox=dict(facecolor='white', alpha=0.8))

# Set x and y axis labels on our ax
my_fig.supxlabel('Longitude')
my_fig.supylabel('Latitude')

# Set title of our figure
my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')

# Set title of our plot as the timestamp of our data
my_plt.set_title(f'{my_timestamp}')

# Show the resulting visualization
plt.show()
```



Here we plot with xarray's built-in matplotlib powered plotter.

```
data_array_at_time.plot(vmin=0, vmax=30)
```

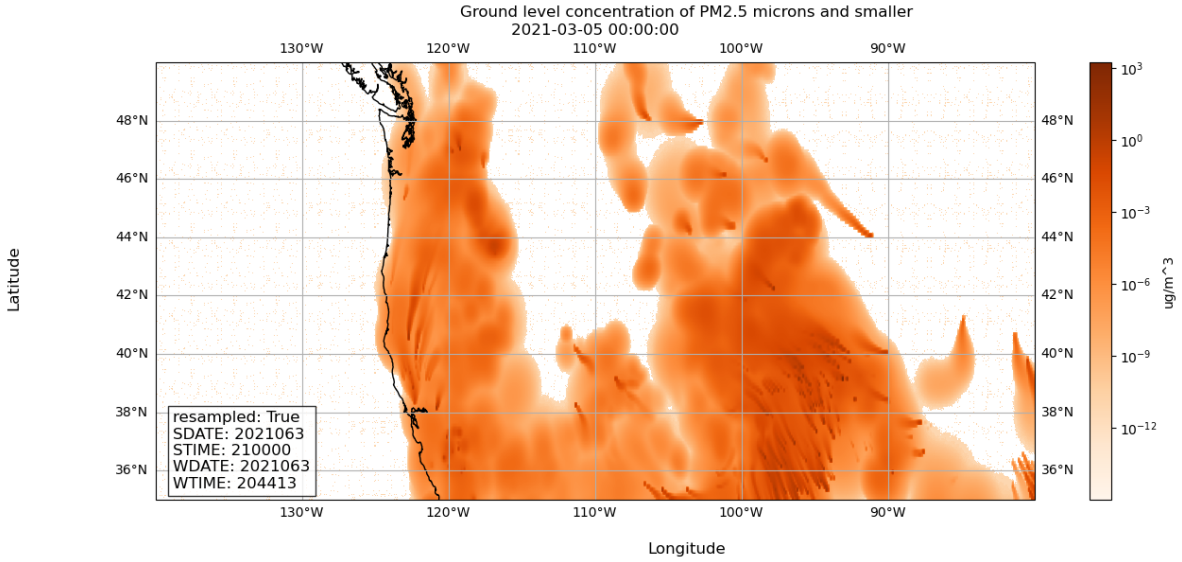


Here we plot data_array_at_latlon. We use the exact same code, but define `my_extent` accordingly.

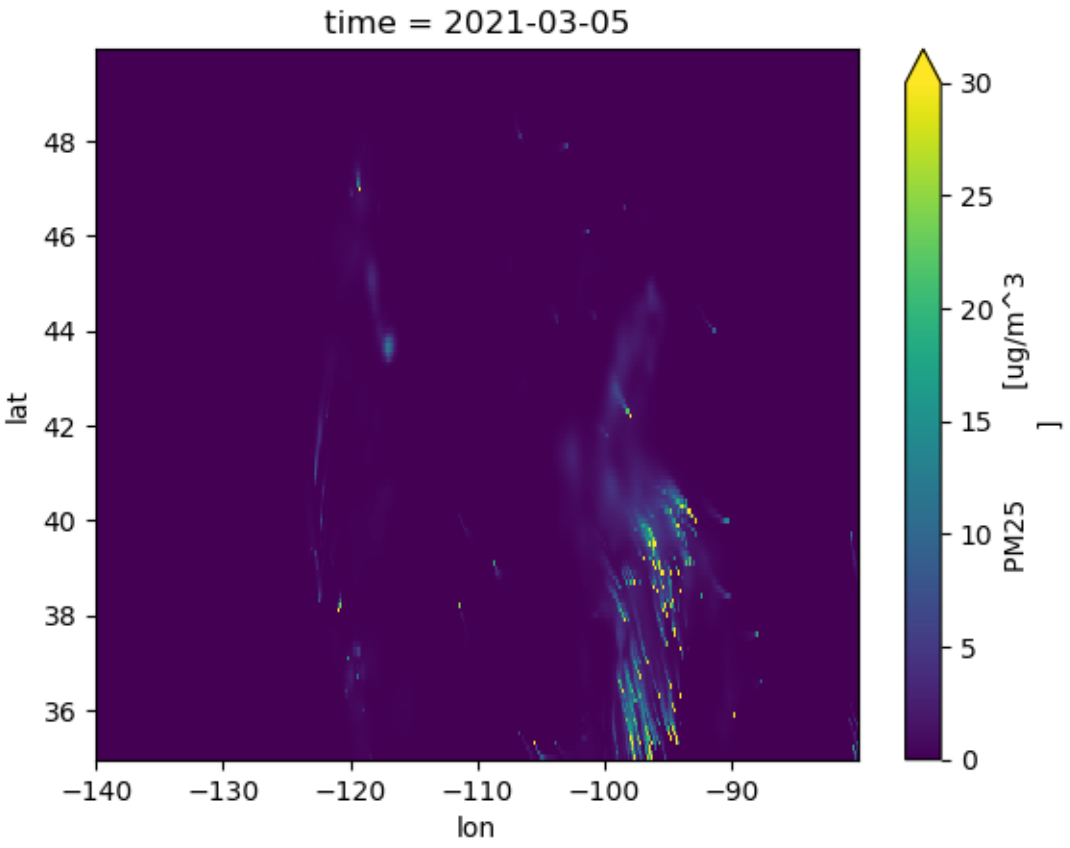
```
# Let's use matplotlib's imshow, since our data is on a grid
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html

# Initialize a figure and plot, so we can customize figure and plot of data
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html
my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.PlateCarree())

# Let's set some parameters to get the visualization we want
# ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html

# color PM25 values on a log scale, since values are small
my_norm = "log"
# ***this will number our x and y axes based on the longitude latitude range***
my_extent = [min_lon, max_lon, min_lat, max_lat]
# ensure the aspect ratio of our plot fits all data, matplotlib can does this automatically
```

```
my_aspect = 'auto'
# tell matplotlib, our origin is the lower-left corner
my_origin = 'lower'
# select a colormap for our plot and the color bar on the right
my_cmap = 'Oranges'

# create our plot using imshow
plot = plt.imshow(data_array_at_latlon.values, norm=my_norm, extent=my_extent,
          aspect=my_aspect, origin=my_origin, cmap=my_cmap)

# draw coastlines
my_plt.coastlines()

# draw latitude longitude lines
# ref: https://scitools.org.uk/cartopy/docs/latest/gallery/gridlines_and_labels/gridliner.htm
my_plt.gridlines(draw_labels=True)

# add a colorbar to our figure, based on the plot we just made above
my_fig.colorbar(plot,location='right', label='ug/m^3')

# Add metadata as text annotations
metadata_text = (
    f'resampled: {resampled.values}\n'
    f'SDATE: {sdate.values}\n'
    f'STIME: {stime.values}\n'
    f'WDATE: {wdate.values}\n'
    f'WTIME: {wtime.values}'
)

# Place metadata text on the plot
my_plt.text(0.02, 0.02, metadata_text, transform=my_plt.transAxes,
            fontsize=12, verticalalignment='bottom', bbox=dict(facecolor='white', alpha=0.8))

# Set x and y axis labels on our ax
my_fig.supxlabel('Longitude')
my_fig.supylabel('Latitude')

# Set title of our figure
my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')

# Set title of our plot as the timestamp of our data
my_plt.set_title(f'{my_timestamp}')
```

```
# Show the resulting visualization
plt.show()
```




```
data_array_at_latlon.plot(vmin=0, vmax=30)
```



1.6.3 *Please reach out to Arleth Salinas or Valerio Pascucci for any concerns about the notebook. Thank you!*

- Arleth Salinas (arleth.salinas@utah.edu)
- Valerio Pascucci (pascucci.valerio@gmail.com)

2 Introduction

Wildfires in North America have significantly impacted ecosystems and human society (McKENZIE et al. 2004). Climate change affects the frequency, duration, and severity of wildfires thus necessitating the use of wildfire prediction systems to effectively mitigate wildfire impact. However, data for understanding the impact of climate change on wildfires is limited, only available for a few regions and for only a few decades [2]. Furthermore, wildfire prediction systems in North America prioritize decision making and fire management on short timescales, from minutes to months. Therefore, long term wildfire prediction systems have limited access to aggregate short term data, due to resource constraints from fire management entities to share the data they collect and curate.

The Weather Forecast Research Team at the University of British Columbia (UBC) generates a short term dataset of PM2.5 smoke particulate presence in North America. Over the past 3 years, each day four times a day, UBC has created forecasts of PM2.5 smoke particulate on the ground for Canada and the continental United States. This is done using their The BlueSky Western Canada Wildfire Smoke Forecasting System (BlueSky Canada 2021). UBC provides access to this data to paying customers and for free on a daily basis via a web-based visualization and file download.

These smoke predictions are useful for those who must make decisions on how to deal with smoke as it comes. However, these years of forecasts are not available in a non-trivial fashion for long term forecasting. The data only exists among the hundreds of NetCDF files that UBC has generated.

Our task is to obtain a single long term dataset from the smoke forecast files that are available from UBC.

3 System and Environment

3.1 Machine Specification

The data we curate is over 300 gigabytes large. Therefore we used the SCI institute's in-house machine 'atlantis' for data staging and processing. See Table 3.1.

Table 3.1: 'atlantis' System Specifications

Property	Value
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
Address sizes	44 bits physical, 48 bits virtual
CPU(s)	48
On-line CPU(s) list	0-47
Thread(s) per core	2
Core(s) per socket	6
Socket(s)	4

3.2 Environment

We aim to work in an environment that can be most easily reproduced and documented. Therefore we used Python 3.9.19 via conda. We did all our work within the Project Jupyter environment.

To find our the `yml` file containing our exported conda environment please see the sidebar.

To work on the 'atlantis' machine, we used SSH to connect to the machine.

In the proceeding chapters we will specify which tools and libraries were used and why.

4 The Data Source

It is important to understand exactly what data is available and how to obtain that data.

We encourage you to explore the dataset on UBC's website to further understand the dataset's use and development by UBC (BlueSky Canada 2021).

Here we establish:

- What systems are available to obtain the smoke forecast data from UBC?
- What are in the files that UBC provides?
- What metadata is associated with the files we obtain, both from within the file and about the files as a whole?

This information **was not** determined the first time we explored this dataset.

As seen in the later chapters, we operated under misinformed assumptions and encountered issues only resolvable by operating on the data "blindly". The purpose of this chapter is to establish the final set of information we learned about our data source after extensive exploration and manipulation.

4.1 Overview

The Weather Forecast Research Team at the University of British Columbia (UBC) generates a short term dataset of PM2.5 smoke particulate presence in North America. This is done using their The BlueSky Western Canada Wildfire Smoke Forecasting System. Over the past 3 years, each day four times a day, UBC creates 2-day forecasts of PM2.5 smoke particulate on the ground for Canada and the continental United States. Each such forecast is downloadable as a NetCDF file or KMZ file. UBC provides access to these predictions for free on a daily basis at their website firesmoke.ca.

4.2 UBC Smoke Forecast Files Access

4.2.1 Available Forecasts

All forecast files are uniquely identifiable with a forecast ID based on when their meteorology forecast is initiated, a smoke forecast initialization time, and by date. The time ranges of available files by forecast ID is shown in Table 4.1. Please note, there are occassional **failed forecasts or otherwise unavailable files** within the date ranges specified Table 4.1, see Section 4.4 for further details.

Table 4.1: Dates for which all forecast ID datasets are publicly available. All times are in UTC and the grid size is 12 km.

Forecast ID	Meteorology Forecast Initialization (UTC)	Smoke Forecast Initialization (UTC)	Start Date	End Date
BSC00CA12-01	00Z	08Z	March 4, 2021	Present Day
BSC06CA12-01	06Z	14Z	March 4, 2021	Present Day
BSC12CA12-01	12Z	20Z	March 3, 2021	Present Day
BSC18CA12-01	18Z	02Z	March 4, 2021	Present Day

The smoke forecasts are updated daily, including the present day, so there is no fixed end date. Therefore, the latest data must be downloaded on a regular basis. We have not implemented this process yet, so the latest forecast files we use are up to June 27, 2024.

There is no official source stating the earliest available date for each forecast. So, knowing the project began in 2021, we inferenced that the earliest available date would be in 2021. Via trial and error we found the earliest available dates.

4.2.2 Download Instructions

To download the 2-day forecast for the forecast initialization date of one's choice, one follows the instructions below. The downloaded file can be a NetCDF or KMZ file.

Go to the URL: `https://firesmoke.ca/forecasts/{Forecast ID}/{YYYYMMDD}{InitTime}/{File Type}`

Where:

- `YYYYMMDD` is the date of choice.

- `ForecastID` and `InitTime` are the chosen values as described in Table 4.2.
- `File Type` is either `dispersion.nc` or `dispersion.kmz` for either the NetCDF file or KMZ file, respectively.

Table 4.2: UBC Smoke Forecast Data Download Parameters.

Forecast ID	Smoke Forecast Initialization (UTC)
BSC00CA12-01	08
BSC06CA12-01	14
BSC12CA12-01	20
BSC18CA12-01	02

4.2.2.1 Download Example

Let's try downloading the forecast for January 1, 2024 where the weather forecast is initiated at 00:00:00 UTC and the smoke forecast is initialized at 08:00:00 UTC by navigating to the corresponding URL.

```
forecast_id = "BSC00CA12-01"
yyyymmdd = "20210304"
init_time = "08"

url = (
    f"https://firesmoke.ca/forecasts/{forecast_id}/{yyyymmdd}{init_time}/dispersion.nc"
)

print(f"Navigate to this URL in your browser: {url}")
```

```
Navigate to this URL in your browser: https://firesmoke.ca/forecasts/BSC00CA12-01/2021030408/
```

4.3 The NetCDF File

Next, let's look at what is within the NetCDF file located at the URL in our previous example.

4.3.1 File Preview

We load `dispersion.nc` using `xarray`, which provides a preview of the file.

```
import xarray as xr

ds = xr.open_dataset("data_notebooks/data_source/dispersion.nc")
ds
```

```
<xarray.Dataset> Size: 81MB
Dimensions:  (TSTEP: 51, VAR: 1, DATE-TIME: 2, LAY: 1, ROW: 381, COL: 1041)
Dimensions without coordinates: TSTEP, VAR, DATE-TIME, LAY, ROW, COL
Data variables:
    TFLAG    (TSTEP, VAR, DATE-TIME) int32 408B ...
    PM25     (TSTEP, LAY, ROW, COL) float32 81MB ...
Attributes: (12/33)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                    ...
    EXEC_ID:        ????????????????                                         ...
    FTYPE:          1
    CDATE:          2021063
    CTIME:          101914
    WDATE:          2021063
    ...             ...
    VGLVLS:         [10.  0.]
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                       ...
    HISTORY:
```

4.3.2 File Attributes

`dispersion.nc` contains the following attributes. Note that for all files across forecast IDs, they have the same dimension and variable names:

4.3.2.1 Dimensions:

The dimensions described in Table 4.3 determine on which indicies we may index our variables.

Table 4.3: Description of Dimensions for Indexing Data in NetCDF Files

Dimension	Size	Description
TSTEP	51	This dimension represents the number of time steps in the file. Each file has 51 hours represented.
VAR	1	This dimension is a placeholder for the variables in the file.
DATE-TIME	2	This dimension stores the date and time information for each time step.
LAY	1	This dimension represents the number of layers in the file, which is 1 in this case.
ROW	381	This dimension represents the number of rows in the spatial grid.
COL	1041	This dimension represents the number of columns in the spatial grid.

4.3.2.2 Variables:

The variables described in Table 4.4 contain the data in question that we would like to extract.

Table 4.4: Description of Variables in NetCDF Files

Variable	Dimensions	Data Type	Description
TFLAG	TSTEP, VAR, DATE-TIME	int32	This variable stores the date and time of each time step.
PM25	TSTEP, LAY, ROW, COL	float32	This variable contains the concentration of particulate matter (PM2.5) for each time step, layer, row, and column in the spatial grid.

4.3.2.3 Attributes

Of the 33 available attributes we use the ones shown in Table 4.5:

Table 4.5: Description of Attributes in `dispersion.nc`

Attribute	Value	Description
CDATE	2021063	The creation date of the dataset, in YYYYDDD format.
CTIME	101914	The creation time of the dataset, in HHMMSS format.
WDATE	2021063	The date for which the weather forecast is initiated, in YYYYDDD format.

Attribute	Value	Description
WTIME	101914	The time for which the weather forecast is initiated, in HHMMSS format.
SDATE	2021063	The date for which the smoke forecast is initiated,in YYYYDDD format.
STIME	90000	The time for which the weather forecast is initiated, in HHMMSS format.
NCOLS	1041	The number of columns in the spatial grid.
NROWS	381	The number of rows in the spatial grid.
XORIG	-156.0	The origin (starting point) of the grid in the x-direction.
YORIG	32.0	The origin (starting point) of the grid in the y-direction.
XCELL	0.10000000149011612	The cell size in the x-direction.
YCELL	0.10000000149011612	The cell size in the y-direction.

Let's look closer at what exactly is within one NetCDF file in the following demo.

4.3.3 NetCDF Visualization Demo

In this demo we load a different `dispersion.nc` file and explore how to visualize the data within the file.

4.3.3.1 Accessing the File

We use the forecast for March 4, 2021 where the weather forecast is initiated at 00:00:00 UTC and the smoke forecast is initialized at 08:00:00 UTC. You can download this file by navigating to the URL below.

```
forecast_id = "BSC00CA12-01"
yyyymmdd = "20210304"
init_time = "08"

url = (
    f"https://firesmoke.ca/forecasts/{forecast_id}/{yyyymmdd}{init_time}/dispersion.nc"
)

print(f"Download this file from URL: {url}")

# import urllib.request
# urllib.request.urlretrieve(url, "dispersion.nc")
```

```
Download this file from URL: https://firesmoke.ca/forecasts/BSC00CA12-01/2021030408/dispersio
```

4.3.3.1.1 Opening the File

We use xarray to open the NetCDF file and preview it.

```
import xarray as xr

ds = xr.open_dataset("dispersion.nc")

ds
```

```
<xarray.Dataset>
Dimensions:  (TSTEP: 51, VAR: 1, DATE-TIME: 2, LAY: 1, ROW: 381, COL: 1041)
Dimensions without coordinates: TSTEP, VAR, DATE-TIME, LAY, ROW, COL
Data variables:
    TFLAG    (TSTEP, VAR, DATE-TIME) int32 ...
    PM25     (TSTEP, LAY, ROW, COL) float32 ...
Attributes: (12/33)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                    ...
    EXEC_ID:        ????????????????                                         ...
    FTYPE:          1
    CDATE:          2021063
    CTIME:          101914
    WDATE:          2021063
    ...             ...
    VGLVLS:         [10.  0.]
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                       ...
    HISTORY:
```

4.3.3.2 Using the Data

4.3.3.2.1 Accessing Arrays

The data we are interested in is the PM2.5 values. Let's use `xarray` to get the array in the `PM25` variable.

```
ds["PM25"]
```

```
<xarray.DataArray 'PM25' (TSTEP: 51, LAY: 1, ROW: 381, COL: 1041)>
[20227671 values with dtype=float32]
Dimensions without coordinates: TSTEP, LAY, ROW, COL
Attributes:
    long_name:  PM25
    units:      ug/m^3
    var_desc:   PM25                                                            ...
```

The dimensions of the `PM25` data array are composed of `TSTEP`, `LAY`, `ROW`, and `COL`. We do not need the `LAY` dimension, so let's use `numpy` to drop it.

```
import numpy as np

ds_pm25_vals = ds["PM25"].values                                                ①
print(f'The shape of the data contained in PM25 variable is: {np.shape(ds_pm25_vals)}')

ds_pm25_vals = np.squeeze(ds_pm25_vals)                                         ②
print(f'After squeezing, the shape is: {np.shape(ds_pm25_vals)}')
```

① Use `.values` to get the four dimensional array.
② Use `np.squeeze` to drop the LAY axis

```
The shape of the data contained in PM25 variable is: (51, 1, 381, 1041)
After squeezing, the shape is: (51, 381, 1041)
```

We now have `ds_pm25_vals`. Next, let's select a time step and visualize the data.

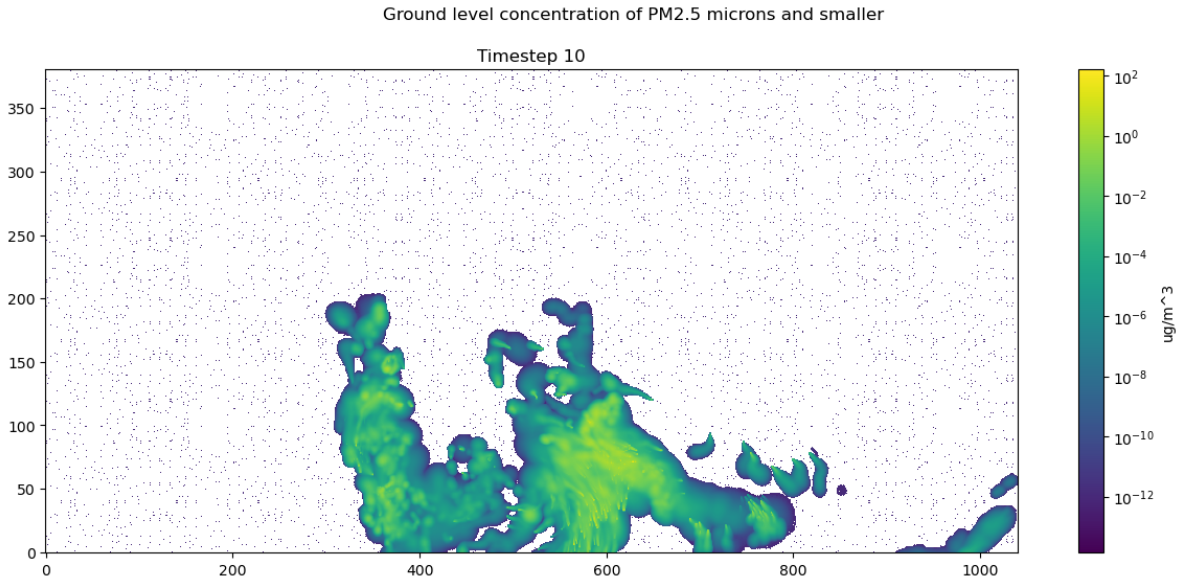
4.3.3.2.2 Visualize Array in `matplotlib`

We can index time step 10 and use `matplotlib` to visualize the timestep

```
import matplotlib.pyplot as plt

tstep = 10                                                                      ①
smoke_at_tstep = ds_pm25_vals[tstep, :, :]

my_fig, my_plt = plt.subplots(figsize=(15, 6))
```

```
my_norm = "log"                                                                    ②
my_aspect = 'auto'                                                                 ③
my_origin = 'lower'                                                                ④
my_cmap = 'viridis'                                                                ⑤

plot = my_plt.imshow(smoke_at_tstep, norm=my_norm, aspect=my_aspect, origin=my_origin, cmap=m

my_fig.colorbar(plot, location='right', label='ug/m^3')                            ⑦

my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')         ⑧

my_plt.set_title(f'Timestep {tstep}')                                              ⑨

plt.show()                                                                         ⑩
```

① Index `ds_pm25_vals` at `TSTEP = 10`, selecting all `ROW`s and `COL`s.
② Color PM25 values on a log scale, since values are small.
③ Ensure the aspect ratio of our plot fits all data, `matplotlib` can do this automatically.
④ Tell `matplotlib` our origin is the lower-left corner.
⑤ Select a colormap for our plot and draw the color bar on the right.
⑥ Create our plot using `imshow`.
⑦ Add a colorbar to our figure, based on the plot we just made above.
⑧ Set title of our figure.
⑨ Set title of our plot as the timestamp of our data.
⑩ Show the resulting visualization.



Notice there are no axis labels or metadata presented here. Next we will show how to use the metadata in `dispersion.nc` so the data is actually interpretable.

4.3.3.3 Incorporating Metadata to Visualization via Coordinates

4.3.3.3.1 Latitude and Longitude Coordinates

`dispersion.nc` includes attributes to generate the latitude and longitude values on the grid defined by `NCOLS` and `NROWS`. We use this grid to match each data point in the `PM25` variable to a lat/lon coordinate.

```
xorig = ds.XORIG
yorig = ds.YORIG
xcell = ds.XCELL
ycell = ds.YCELL
ncols = ds.NCOLS
nrows = ds.NROWS

longitude = np.linspace(xorig, xorig + xcell * (ncols - 1), ncols)
latitude = np.linspace(yorig, yorig + ycell * (nrows - 1), nrows)

print("Size of longitude & latitude arrays:")
print(f'np.size(longitude) = {np.size(longitude)}')
print(f'np.size(latitude) = {np.size(latitude)}\n')
print("Min & Max of longitude and latitude arrays:")
print(f'longitude: min = {np.min(longitude)}, max = {np.max(longitude)}')
print(f'latitude: min = {np.min(latitude)}, max = {np.max(latitude)}')
```

```
Size of longitude & latitude arrays:
np.size(longitude) = 1041
np.size(latitude) = 381

Min & Max of longitude and latitude arrays:
longitude: min = -156.0, max = -51.999998450279236
latitude: min = 32.0, max = 70.00000056624413
```

`xarray` allows us to create coordinates, which maps variable values to a value of our choice. In this case, we create coordinates mapping PM25 values to a latitude and longitude value.

```
ds.coords['lat'] = ('ROW', latitude)                                    ①
ds.coords['lon'] = ('COL', longitude)

ds = ds.swap_dims({'COL': 'lon', 'ROW': 'lat'})                         ②

ds
```

① Create coordinates for latitude and longitude.
② Replace `COL` and `ROW` dimensions with newly calculated longitude and latitude coordinates.

```
<xarray.Dataset>
Dimensions:  (TSTEP: 51, VAR: 1, DATE-TIME: 2, LAY: 1, lat: 381, lon: 1041)
Coordinates:
  * lat      (lat) float64 32.0 32.1 32.2 32.3 32.4 ... 69.6 69.7 69.8 69.9 70.0
  * lon      (lon) float64 -156.0 -155.9 -155.8 -155.7 ... -52.2 -52.1 -52.0
Dimensions without coordinates: TSTEP, VAR, DATE-TIME, LAY
Data variables:
    TFLAG    (TSTEP, VAR, DATE-TIME) int32 ...
    PM25     (TSTEP, LAY, lat, lon) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Attributes: (12/33)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                   ...
    EXEC_ID:        ????????????????                                        ...
    FTYPE:          1
    CDATE:          2021063
    CTIME:          101914
    WDATE:          2021063
    ...             ...
    VGLVLS:         [10.  0.]
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                      ...
    HISTORY:
```

Now let's move on to incorporating time stamp metadata.

4.3.3.3.2 Time Coordinates

Recall, there is a `TFLAG` variable in `dispersion.nc`.

```
ds['TFLAG']
```

```
<xarray.DataArray 'TFLAG' (TSTEP: 51, VAR: 1, DATE-TIME: 2)>
[102 values with dtype=int32]
Dimensions without coordinates: TSTEP, VAR, DATE-TIME
Attributes:
    units:      <YYYYDDD,HHMMSS>
    long_name:  TFLAG
    var_desc:   Timestep-valid flags:  (1) YYYYDDD or (2) HHMMSS             ...
```

The earliest and latest `TFLAG`s look like the following:

```
print(f"Earliest available TFLAG is {ds['TFLAG'].values[0][0]}")
print(f"Latest available TFLAG is {ds['TFLAG'].values[-1][0]}")
```

```
Earliest available TFLAG is [2021063   90000]
Latest available TFLAG is [2021065  110000]
```

This time flags require processing to be immediately legible. Let's write a function to process the time flag accordingly. We use the `datetime` library.

```
import datetime

def parse_tflag(tflag):
    """
    Return the tflag as a datetime object
    :param list tflag: a list of two int32, the 1st representing date and 2nd representing ti
    """
    date = int(tflag[0])                                                    ①
    year = date // 1000                                                     ②
    day_of_year = date % 1000                                               ③

    final_date = datetime.datetime(year, 1, 1) + datetime.timedelta(days=day_of_year - 1) ④

    time = int(tflag[1])                                                    ⑤
    hours = time // 10000                                                   ⑥
    minutes = (time % 10000) // 100                                         ⑦
    seconds = time % 100                                                    ⑧

    full_datetime = datetime.datetime(year, final_date.month, final_date.day, hours, minutes
    return full_datetime
```

① Obtain year and day of year from `tflag[0]` (date).
② Extract the year from the first 4 digits of `tflag[0]`.
③ Extract the day of the year from the last 3 digits of `tflag[0]`.
④ Create a `datetime` object representing the date.
⑤ Obtain hour, minutes, and seconds from `tflag[1]` (time).
⑥ Extract hours from the first 2 digits of `tflag[1]`.
⑦ Extract minutes from the 3rd and 4th digits of `tflag[1]`.
⑧ Extract seconds from the last 2 digits of `tflag[1]`.
⑨ Create the final `datetime` object with the extracted date and time components.

Now we have datetime objects to represent the timeflag in a more legible and usable format.

```
print(f"Earliest available TFLAG is {parse_tflag(ds['TFLAG'].values[0][0])}")
print(f"Latest available TFLAG is {parse_tflag(ds['TFLAG'].values[-1][0])}")
```

```
Earliest available TFLAG is 2021-03-04 09:00:00
Latest available TFLAG is 2021-03-06 11:00:00
```

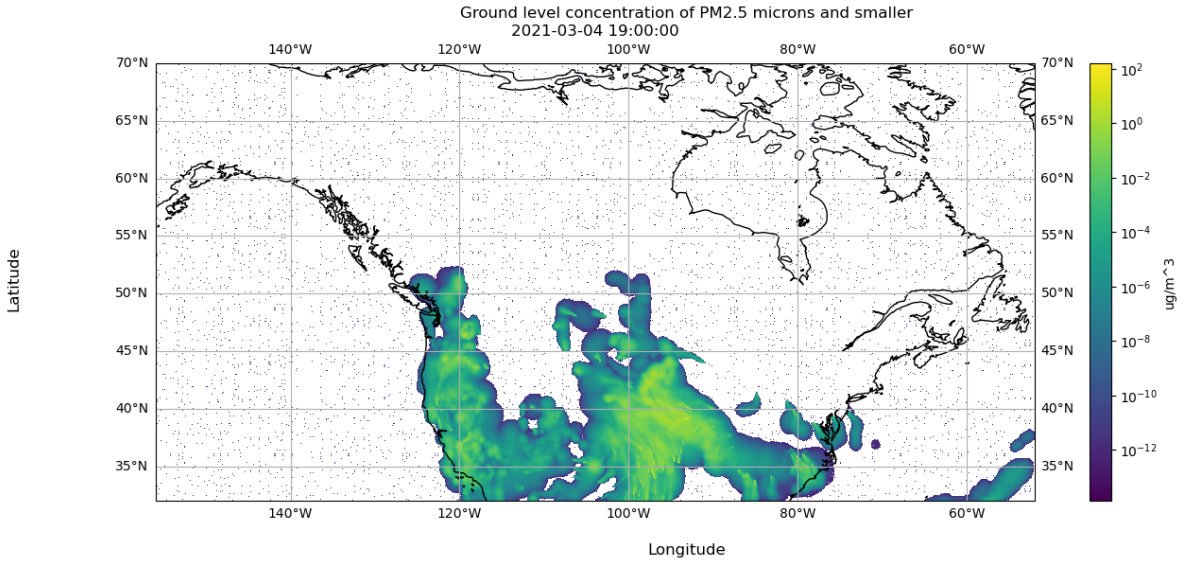
4.3.3.3.3 Visualize Array in `matplotlib`

Let's visualize timestep 10 again, but now we can label the data using latitudes and longitudes, and the corresponding time flag.

```
tstep = 10                                                                    ①
smoke_at_tstep = ds_pm25_vals[tstep, :, :]                                    ②
tstep_tflag = parse_tflag(ds['TFLAG'].values[tstep][0])                       ③

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

my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.PlateCarree())

my_norm = "log"                                                               ⑤
my_extent = [np.min(longitude), np.max(longitude), np.min(latitude), np.max(latitude)] ⑥
my_aspect = 'auto'                                                            ⑦
my_origin = 'lower'                                                           ⑧
my_cmap = 'viridis'                                                           ⑨

plot = my_plt.imshow(smoke_at_tstep, norm=my_norm, extent=my_extent,
          aspect=my_aspect, origin=my_origin, cmap=my_cmap)                   ⑩

my_plt.coastlines()                                                           ⑪
```

```
my_plt.gridlines(draw_labels=True)                                                    (12)

my_fig.colorbar(plot, location='right', label='ug/m^3')                               (13)

my_fig.supxlabel('Longitude')                                                         (14)
my_fig.supylabel('Latitude')                                                          (15)

my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')            (16)

my_plt.set_title(f'{tstep_tflag}')                                                    (17)

plt.show()                                                                            (18)
```

(1) Define the time step.
(2) Extract the PM2.5 data for the specified time step.
(3) Parse the time flag for the specified time step.
(4) Initialize a figure and plot with a specific projection.
(5) Set the normalization for PM2.5 values to a logarithmic scale.
(6) Define the extent of the plot based on the longitude and latitude range.
(7) Set the aspect ratio of the plot to fit all data automatically.
(8) Specify the origin of the plot as the lower-left corner.
(9) Choose a colormap for the plot.
(10) Create the plot using `imshow` with the specified parameters.
(11) Draw coastlines on the plot.
(12) Draw latitude and longitude lines with labels.
(13) Add a colorbar to the figure based on the plot.
(14) Set the x-axis label.
(15) Set the y-axis label.
(16) Set the title of the figure.
(17) Set the title of the plot as the timestamp of the data.
(18) Display the resulting visualization.




```
ds.close()
```

Now that we understand how to load the data and metadata from the file and process it for visualization, let's establish the data and metadata available to us across *all* NetCDF files.

4.4 Information Across all NetCDF Files

Knowing what is within one NetCDF file as well as the date range for which we can download them, let's establish the metadata associated with the NetCDF files as a *collection*.

4.4.1 Disk Size

For the time ranges we cover, Table 6.1 shows how large the set of files per forecast ID are.

Table 4.6: File Sizes and Counts for Each Forecast ID within the Specified Date Range

Forecast ID	Date Range	Size	File Count
BSC00CA12-01	March 4, 2021 - June 27, 2024	84G	1077
BSC06CA12-01	March 4, 2021 - June 27, 2024	78G	1022
BSC12CA12-01	March 3, 2021 - June 27, 2024	79G	1022
BSC18CA12-01	March 4, 2021 - June 27, 2024	79G	1023
Total		**320G**	**4144**

4.4.2 Temporal Data Availability

4.4.2.1 Loading Files

We have downloaded all NetCDF files available up to June 27, 2022.

Let's demonstrate the staggered nature of the forecasts by opening up the files for March 2, 2024 to March 4, 2022. We use the `parse_tflag` function.

```
1. Obtain year and day of year from tflag[0] (date)
2. Create datetime object representing date
3. Obtain hour, mins, and secs from tflag[1] (time)
4. Create final datetime object
```

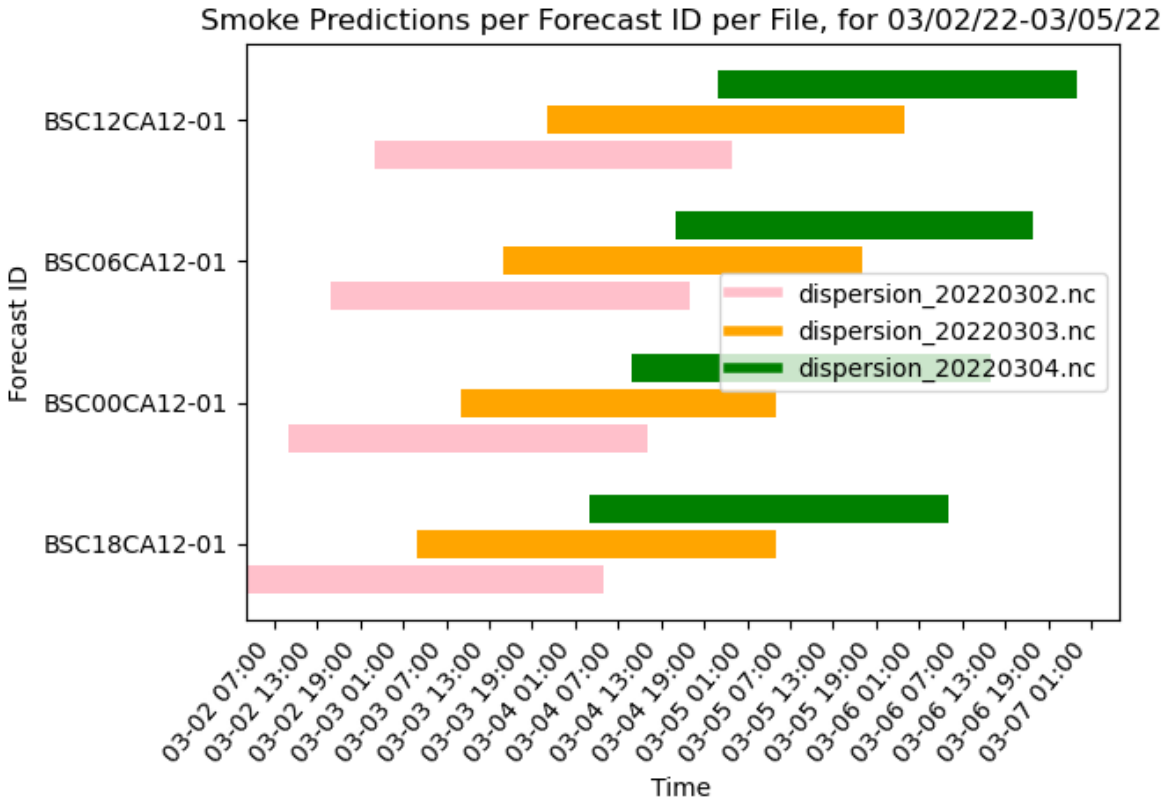
```
files_dir = 'dispersion_files'                                                    ①
ids = ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]            ②
file_names = ['dispersion_20220302.nc', 'dispersion_20220303.nc', 'dispersion_20220304.nc'] (
```

① Define the directory location of our files.
② List of the forecast IDs we use.
③ `file_names` is each file we will open per forecast ID next.

```
import xarray as xr
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.lines import Line2D

fig, ax = plt.subplots()

colors = ['pink', 'orange', 'green']
num_files = len(file_names)
bar_height = 1 / (num_files + 1)  # Dynamic height for the bars

for idx, id_ in enumerate(ids):                                                   ①
    for i, f in enumerate(file_names):
        curr_file = f'{files_dir}/{id_[3:5]}/{f}'                                 ②
        ds = xr.open_dataset(curr_file)                                           ③
        tflags = ds['TFLAG'].values

        earliest_time = parse_tflag(tflags[0][0])                                 ④
        latest_time = parse_tflag(tflags[-1][0])
```

```
        # Adjust y position to avoid overlap, using `idx` and `i` to space out bars
        y_position = idx + i * bar_height
        ax.barh(y_position, latest_time - earliest_time, left=earliest_time,
                height=bar_height * 0.8, color=colors[i])                    ⑤

# Adjust y-axis to display ID labels in a way that matches the bar positions
ax.set_yticks([i + (num_files - 1) * bar_height / 2 for i in range(len(ids))])
ax.set_yticklabels(ids)

ax.xaxis.set_major_formatter(mdates.DateFormatter("%m-%d %H:%M"))
ax.xaxis.set_major_locator(mdates.HourLocator(interval=6))
fig.autofmt_xdate(rotation=50)

ax.set_xlabel("Time")
ax.set_ylabel("Forecast ID")
ax.set_title("Smoke Predictions per Forecast ID per File, for 03/02/22-03/05/22")

legend_elements = [Line2D([0], [0], linewidth=4, color=colors[i], label=file_names[i]) for i
ax.legend(handles=legend_elements, loc='center right')

# plt.show()
plt.savefig('overlaps.png', bbox_inches="tight")
```

① Open each file for each forecast ID.
② Create the path string.
③ Open the file with `xarray` and get the TFLAG values.
④ Get the earliest and latest available time flags.
⑤ Plot the time range represented with the time flags as a horizontal bar.



The 6 loaded files cover the time ranges shown above.

Notice, for any given hour, there can be various predictions available to choose from. For example, for 2022-03-04 01:00, we can find it represented in `dispersion_20220303.nc` across all forecast IDs.

Given that these predictions are forecasts, we run on the assumption that the earliest available predictions per file are the most accurate.

Therefore, for example, if we want to choose the most accurate PM2.5 prediction for 2022-03-04 01:00 we would load the prediction with the BSC12CA12-01 forecast ID.

For further details on how we load the best prediction for every hour of the dataset, see Section 6.10.

Now that we know what exactly is within a NetCDF file and across all the NetCDF files, we will continue to describe our data curation process. Next we describe how we load all of the data available from UBC onto our machine.

5 Data Loading

Knowing what data is available and how to obtain the data, we can proceed with loading the data onto our staging system.

5.1 Downloading Data Locally

We decided to download all the available files from the data source onto our data staging machine to process from there.

We created 4 directories for each forecast ID that UBC provides at the following directory on our machine:

```
/usr/sci/cedmav/data/firesmoke
   BSC00CA12-01
   BSC06CA12-01
   BSC12CA12-01
   BSC18CA12-01
```

The following shows our approaches to doing this and discusses how our approach evolved. Note the scripts below refer to varying directories, but through simple copying operations we stored the final downloaded files to the directories listed above.

5.1.1 First Approach

We delineate our first approach by detailing our download script, which is available in it's entirety in the side bar.

```
import wget
import pandas as pd

ids = ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]      (1)
start_dates = ["20210304", "20210304", "20210304", "20210303"]
end_dates = ["20231016", "20240210", "20231016", "20231015"]
```

```
init_times = ["02", "08", "14", "20"]

for i in zip(start_dates, end_dates, ids, init_times):                    ②
    start_date = i[0]
    end_date = i[1]
    forecast_id = i[2]
    init_time = i[3]

    dates = pd.date_range(start=start_date, end=end_date)
    dates = dates.strftime("%Y%m%d").tolist()

    for date in dates:                                                    ③
      url = (
          "https://firesmoke.ca/forecasts/"
          + forecast_id
          + "/"
          + date
          + init_time
          + "/dispersion.nc"
      )
      directory = "/Users/arleth/Mount/firesmoke/" + forecast_id + "/dispersion_" + date + "
      wget.download(url, out=directory)
```

① First, create 4 lists containing forecast IDs, the start and end dates we wish to index on, and the smoke forecast initiation times. We will loop through the 4 sets of parameters.
② In a for loop, we use `pandas` to create a list of every date from the start date and end date of the current iteration. We will loop through these dates next.
③ For each `date` in the list, we create the `url` to download the file.
④ Finally, we use `wget` to download the contents at `url`to `directory`. We append `date` to the file name so each file downloaded is identifiable by date.

We assumed that for all URLs, there was an available NetCDF file for download

However, we realized that we downloaded either a NetCDF file *or an HTML webpage*. Using `wget` forcibly saved the contents at the URL into a NetCDF file.

This issue was not identified until *after* we visualized each hour of the data, and we noticed gaps and errors in our scripts to create visualizations. See Chapter 7 for further details on identifying these issues. For now, we show our modified approach to downloading the NetCDF files.

5.1.2 Second Approach

Our second approach is similar to the first, except we use `requests`, an HTTP client library that allows us to see the headers returned from the URL we query. The script is available in the side bar.

```
import requests
import pandas as pd
from datetime import datetime

ids = ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]      ①
start_dates = ["20210304", "20210304", "20210304", "20210303"]
today = datetime.now().strftime("%Y%m%d")
init_times = ["02", "08", "14", "20"]

for i in zip(start_dates, ids, init_times):                                  ②
    start_date = i[0]
    forecast_id = i[1]
    init_time = i[2]

    dates = pd.date_range(start=start_date, end=today)
    dates = dates.strftime("%Y%m%d").tolist()

    for date in dates:                                                       ③
        url = (
            "https://firesmoke.ca/forecasts/"
            + forecast_id
            + "/"
            + date
            + init_time
            + "/dispersion.nc"
        )
        directory = (                                                        ④
            "/usr/sci/scratch_nvme/arleth/basura_total/"
            + forecast_id
            + "/dispersion_"
            + date
            + ".nc"
        )

        response = requests.get(url, stream=True)                            ⑤
        header = response.headers
        if (
```

```
            "Content-Type" in header
            and header["Content-Type"] == "application/octet-stream"
        ):
            with open(directory, mode="wb") as file:                    ⑥
                file.write(response.content)
                print(f"Downloaded file {directory}")
        else:
            print(header["Content-Type"])
```

① First, create 3 lists containing forecast IDs, the start dates we wish to index on, and the smoke forecast initiation times . Notice we define a variable `today`, this allows us to run this script and query all URLs up to today's date. Note we ran up to June 27, 2024 for now. We will loop through these sets of parameters.
② In a for loop, we use `pandas` to create a list of every date from the start date and end date of the current iteration. We will loop through these dates next.
③ For each `date` in the list, we create the `url` to download the file.
④ Define `directory`, the directory and file name to save the file to.
⑤ We use `requests` to get the HTTP header at `url`. We inspect the `Content-Type` and if it is `application/octet-stream`, we download the file. We confirmed that a URL with a NetCDF file had this content type header.
⑥ We write the content to `directory`, else we print the content header out to check what it is.

This approach yielded the results we expected, we downloaded only NetCDF files. We had failed downloads which appeared during conversion as described in Chapter 6. We assumed those files were unavailable from UBC which we later confirmed as described in Chapter 7.

6 Data Conversion

So far, we have established what the data from UBC is and how to download it to our machine. Now we describe how to compile the data on our machine into an IDX file using the OpenViSUS libary and its access to the PIDX library.

6.1 On Data Validation

We decided to perform data validation *after* conversion to the IDX file format. However, we realized that performing data validation *both before and after* conversion would be best. This is explored further in Chapter 7.

For now, the reader should understand that data validation of the NetCDF files is different from data validation of the IDX file. In this chapter, we use the assumption that the NetCDF files *we can open* have complete and uncorrupted data for conversion to IDX.

6.2 Overview

For all our conversion script versions, the same general process is followed:

1. Check which NetCDF files were successfully downloaded from the data source by attempting to open each downloaded file with `xarray`.
2. Obtain a subset of data from the files to create a dataset of chronological, hour by hour, data.
3. Save this time series data to an `IDX` file using the OpenVisus framework.

We will describe the latest version of our conversion, **version 4**. Throughout, we will explain how previous attempts were deficient.

To see previous attemps in their entirety, refer to the side bar. Please note that the previous scripts were working scripts, therefore they may be incomplete.

6.3 Setting System Directories

First we set the directory paths we want to use during the conversion process, which is to our 4 directories of NetCDF files for each forecast ID.

```
firesmoke_dir = "/usr/sci/cedmav/data/firesmoke"                                    ①
idx_dir = "/usr/sci/scratch_nvme/arleth/idx/firesmoke"

ids = ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]              ②
start_dates = ["20210304", "20210304", "20210304", "20210303"]
end_dates = ["20240627", "20240627", "20240627", "20240627"]
```

① Establish the directory where all forecast ID NetCDFs are stored and where to save our IDX file on the 'atlantis' machine.
② Define the forecast IDs and dates we will loop over.

6.3.1 Rationale and Future Improvements

6.3.1.1 Data Usage

In versions 1 and 2 of our conversion attempt, we did not use all four sets of forecast ID files. We only used BSC12CA12-01 files to compile a single dataset. We learned that by not using all four sets of data, the dataset we created was less accurate. See Chapter 7 for further details.

Therefore we decided to use all four datasets, see Section 6.10 for details. We elect to use dates up to June 27, 2024 as this was the last time we ran our scripts. We have yet to address the issue of how to keep the IDX file constantly up to date with data available up to the present day.

6.4 Checking the NetCDF Files

Recall we downloaded all NetCDF files available from UBC onto our machine in their respective directories as follows:

```
/usr/sci/cedmav/data/firesmoke
   BSC00CA12-01
   BSC06CA12-01
   BSC12CA12-01
   BSC18CA12-01
```

Here, we identify which NetCDF files for each forecast ID successfully open with `xarray` and store them in a dictionary.

We also confirm the following conditions we established in Chapter 4 by using dictionaries to track the max values and unique values of these attributes across all files:

1. All files across all four forecast IDs have the same `NROWS`, `XORIG`, `YORIG`, `XCELL`, `YCELL` values.
2. Some files have either `NCOLS = 1041` or `NCOLS = 1081`, but always `NROWS = 381`.

```
import os
import xarray as xr
import numpy as np
import tqdm

successful_files = {id_: [] for id_ in ids}                                    ①

max_ncols = {id_: 0 for id_ in ids}                                            ②
max_nrows = {id_: 0 for id_ in ids}
ncols = {id_: set() for id_ in ids}                                            ③
nrows = {id_: set() for id_ in ids}

max_grid_x = {id_: {"xorig": 0.0, "xcell": 0.0} for id_ in ids}                ④
max_grid_y = {id_: {"yorig": 0.0, "ycell": 0.0} for id_ in ids}
xorigs = {id_: set() for id_ in ids}                                           ⑤
xcells = {id_: set() for id_ in ids}
yorigs = {id_: set() for id_ in ids}
ycells = {id_: set() for id_ in ids}

for id_ in ids:                                                                ⑥
    file_names = os.listdir(f"{firesmoke_dir}/{id_}/")                         ⑦

    for file in tqdm(file_names):                                              ⑧
        path = f"{firesmoke_dir}/{id_}/{file}"                                 ⑨

        try:                                                                   ⑩
            ds = xr.open_dataset(path)

            successful_files[id_].append(file)                                 ⑪

            max_ncols[id_] = max(max_ncols[id_], ds.NCOLS)                     ⑫
            max_nrows[id_] = max(max_nrows[id_], ds.NROWS)
            max_grid_x[id_]["xorig"] = max(max_grid_x[id_]["xorig"], ds.XORIG, key=abs)
```

```
            max_grid_y[id_]["yorig"] = max(max_grid_y[id_]["yorig"], ds.YORIG, key=abs)
            max_grid_x[id_]["xcell"] = max(max_grid_x[id_]["xcell"], ds.XCELL, key=abs)
            max_grid_y[id_]["ycell"] = max(
                max_grid_y[id_]["ycell"], ds.YCELL, key=abs
            )

            ncols[id_].add(ds.NCOLS)                                              (13)
            nrows[id_].add(ds.NROWS)
            xorigs[id_].add(ds.XORIG)
            yorigs[id_].add(ds.YORIG)
            xcells[id_].add(ds.XCELL)
            ycells[id_].add(ds.YCELL)

        except:                                                                   (14)
            continue

for id_ in successful_files:                                                      (15)
    successful_files[id_] = np.sort(successful_files[id_]).tolist()
```

(1) Initialize a dictionary to hold an empty list for each forecast ID. We update it with the file names that successfully open under the forecast ID directory.
(2) Initialize dictionaries to hold an integer for each forecast ID. We update it to hold the maximum `NCOLS`/`NROWS` value available within forecast ID's set of NetCDF files.
(3) Initialize dictionaries to hold a set for each forecast ID. We update the set to hold all the unique `NCOLS`/`NROWS` values available within the forecast ID's set of NetCDF files.
(4) Initialize dictionaries to hold a dictionary of `xorig`/`yorig` and `xcell`/`ycell` values for each forecast ID. We update it to hold the maximum `xorig`/`yorig` and `xcell`/`ycell` pairs available within the forecast ID's set of NetCDF files.
(5) Initialize dictionaries to track unique `xorig`/`yorig` and `xcell`/`ycell` values.
(6) For each forecast ID, we populate the dictionaries above.
(7) Obtain a list of file names under the directory for `id_`. We loop through each file next.
(8) Begin loop over each file. Note `tqdm` is just an accessory for generating a visible status bar in our Jupyter Notebook.
(9) Obtain absolute path name to current file.
(10) Here we use a `try` statement since opening the file with `xarray` may lead to an error. `except` allows us to catch the exception accordingly and continue trying to open each file.
(11) At this line, the file opened without issue in `xarray`, so append this file name to the `id_` list in the `successful_files` dictionary.
(12) Use `max` to save the largest values in our dictionaries accordingly.
(13) Update the dictionaries of sets with the file's attributes, to ensure we catch all unique values.

(14) If the file did not open during the `try` continue to the next file.
(15) Sort the lists of successfully opened files by name, so they are in chronological order.

The following shows the information gathered:

6.5 BSC18CA12-01

```
dataset: BSC18CA12-01
Number of successful files: 1010
Max cell sizes: max_ncols = 1081 and max_nrows = 381
Max xorig & xcell: {'xorig': -160.0, 'xcell': 0.10000000149011612}
Max yorig & ycell: {'yorig': 32.0, 'ycell': 0.10000000149011612}
ncols: {1081, 1041}
nrows: {381}
xorigs: {-160.0, -156.0}
yorigs: {32.0}
xcells: {0.10000000149011612}
ycells: {0.10000000149011612}
```

6.6 BSC00CA12-01

```
dataset: BSC00CA12-01
Number of successful files: 1067
Max cell sizes: max_ncols = 1081 and max_nrows = 381
Max xorig & xcell: {'xorig': -160.0, 'xcell': 0.10000000149011612}
Max yorig & ycell: {'yorig': 32.0, 'ycell': 0.10000000149011612}
ncols: {1081, 1041}
nrows: {381}
xorigs: {-160.0, -156.0}
yorigs: {32.0}
xcells: {0.10000000149011612}
ycells: {0.10000000149011612}
```

6.7 BSC06CA12-01

```
dataset: BSC06CA12-01
Number of successful files: 997
Max cell sizes: max_ncols = 1081 and max_nrows = 381
Max xorig & xcell: {'xorig': -160.0, 'xcell': 0.10000000149011612}
```

```
Max yorig & ycell: {'yorig': 32.0, 'ycell': 0.10000000149011612}
ncols: {1081, 1041}
nrows: {381}
xorigs: {-160.0, -156.0}
yorigs: {32.0}
xcells: {0.10000000149011612}
ycells: {0.10000000149011612}
```

6.8 BSC12CA12-01

```
dataset: BSC12CA12-01
Number of successful files: 1003
Max cell sizes: max_ncols = 1081 and max_nrows = 381
Max xorig & xcell: {'xorig': -160.0, 'xcell': 0.10000000149011612}
Max yorig & ycell: {'yorig': 32.0, 'ycell': 0.10000000149011612}
ncols: {1081, 1041}
nrows: {381}
xorigs: {-160.0, -156.0}
yorigs: {32.0}
xcells: {0.10000000149011612}
ycells: {0.10000000149011612}
```

6.8.1 Rationale and Future Improvements

6.8.1.1 Unloadable Files

On our first attempt to convert the data we discovered that various files failed to open. Therefore, we used a dictionary to keep track of which files successfully open.

6.8.1.2 Varying Grid Size

In this step we collect attribute information about the two different grids used in the dataset. We proceed to use these attributes to resample the grids accordingly, see Section 6.9.

6.8.1.3 Optimization and Scaling

One improvement to this step is to stop tracking maxes and unique values seperately. Instead, we could just track unique values then get maxes from there. Furthermore, modularizing this

process such that it handles the possibility of new grids being used in the dataset would make the conversion script more scalable.

6.9 Preparing Resampling Grids

Now that we have the sets of openable files, we begin to handle the need to resample data. To resample arrays of shape 381×1041 to 381×1081, we use the SciPy `griddata` function from the `interpolate` package. This function gives interpolated values on set of points `xi` from a set of `points` with corresponding `values`. We refer the reader to SciPy's documentation for details.

In this step, we obtain the grids we wish to use as our `points` and `xi`. See Section 6.9.2 for how we use these grids with the `griddata` function.

6.9.1 Generate Grids of Latitude and Longitude Points

Recall we can generate a set of latitude and longitude coordinates by using the attributes given in each NetCDF file, see Chapter 13 for an example. Here we generate two sets of latitude and longitude coordinates for each grid size.

```
max_xorig = max_grid_x[ids[0]]['xorig']                                    ①
max_xcell = max_grid_x[ids[0]]['xcell']
max_yorig = max_grid_y[ids[0]]['yorig']
max_ycell = max_grid_y[ids[0]]['ycell']

big_lon = np.linspace(max_xorig, max_xorig + max_xcell * (max_ncols[ids[0]] - 1), max_ncols[
big_lat = np.linspace(max_yorig, max_yorig + max_ycell * (max_nrows[ids[0]] - 1), max_nrows[

big_lon_pts, big_lat_pts = np.meshgrid(big_lon, big_lat)                   ③
big_tups = np.array([tup for tup in zip(big_lon_pts.flatten(), big_lat_pts.flatten())])

sml_ds = xr.open_dataset(firesmoke_dir + "/BSC00CA12-01/dispersion_20210304.nc") ④
sml_lon = np.linspace(sml_ds.XORIG, sml_ds.XORIG + sml_ds.XCELL * (sml_ds.NCOLS - 1), sml_ds
sml_lat = np.linspace(sml_ds.YORIG, sml_ds.YORIG + sml_ds.YCELL * (sml_ds.NROWS - 1), sml_ds

sml_lon_pts, sml_lat_pts = np.meshgrid(sml_lon, sml_lat)                   ⑤
sml_tups = np.array([tup for tup in zip(sml_lon_pts.flatten(), sml_lat_pts.flatten())])
```

① Get the x/y origin and cell size parameters for the big 381×1081 grid.
② Generate one two lists, defining a grid of latitudes and longitudes.

③ Using `big_lon` and `big_lat`, use meshgrid to generate our 381×1081 set of longitudes and latitudes.
④ Open a file that uses the small 381×1041 grid. Then, use the attributes in that file to generate two lists defining a grid of latitudes and longitudes.
⑤ Using `sml_lon` and `sml_lat`, use meshgrid to generate our 381×1041 set of longitudes and latitudes.

See below for an example of using these latitude and longitude grids to resample data on a 381×1041 grid to a 381×1081 grid.

6.9.2 Example with `griddata`

Now that we have the two sets of latitude and longitude points, we show an example of how these are used to resample an array of data from a 381×1041 grid to a 381×1081 grid.

In this example we use the latitude and longitude points generated from the attributes determine across all NetCDF files.

```
import numpy as np
import xarray as xr
```

```
max_xorig = -160.0                                                                    ①
max_xcell = 0.10000000149011612                                                       ②
max_yorig = 32.0                                                                      ③
max_ycell = 0.10000000149011612                                                       ④

big_lon = np.linspace(max_xorig, max_xorig + max_xcell * (1081 - 1), 1081)            ⑤
big_lat = np.linspace(max_yorig, max_yorig + max_ycell * (381 - 1), 381)              ⑥

big_lon_pts, big_lat_pts = np.meshgrid(big_lon, big_lat)                              ⑦
big_tups = np.array([tup for tup in zip(big_lon_pts.flatten(), big_lat_pts.flatten())]) ⑧

sml_ds = xr.open_dataset("dispersion_20210304.nc")                                    ⑨
sml_lon = np.linspace(sml_ds.XORIG, sml_ds.XORIG + sml_ds.XCELL * (sml_ds.NCOLS - 1), sml_ds
sml_lat = np.linspace(sml_ds.YORIG, sml_ds.YORIG + sml_ds.YCELL * (sml_ds.NROWS - 1), sml_ds

sml_lon_pts, sml_lat_pts = np.meshgrid(sml_lon, sml_lat)                              ⑫
sml_tups = np.array([tup for tup in zip(sml_lon_pts.flatten(), sml_lat_pts.flatten())]) ⑬
```

① Define the maximum x-origin coordinate.
② Define the maximum x-cell size.
③ Define the maximum y-origin coordinate.

④ Define the maximum y-cell size.
⑤ Create an array for the large longitude grid.
⑥ Create an array for the large latitude grid.
⑦ Create a meshgrid of points using the large longitude and latitude arrays.
⑧ Create a flattened array of tuples representing the large grid points.
⑨ Open the small dataset using xarray.
⑩ Create an array for the small longitude grid.
⑪ Create an array for the small latitude grid.
⑫ Create a meshgrid of points using the small longitude and latitude arrays.
⑬ Create a flattened array of tuples representing the small grid points.

```
print(f'Using the large grid, we have {np.shape(big_tups)[0]} lat/lon points to sample on.')
print(f'Using the small grid, we have {np.shape(sml_tups)[0]} lat/lon points to sample from.
```

```
Using the large grid, we have 411861 lat/lon points to sample on.
Using the small grid, we have 396621 lat/lon points to sample from.
```

Let's get the data at timestep 0 inside `dispersion_20210304.nc`, which uses a grid of size 381×1041.

```
timestep = 0

vals = np.squeeze(sml_ds['PM25'].values)

print(f'The shape of the PM25 array at timestep {timestep} is {np.shape(vals[timestep])}')
```

```
The shape of the PM25 array at timestep 0 is (381, 1041)
```

We use the following parameters with `griddata` to resample `vals`:

```
griddata(points, values, xi, method='cubic', fill_value)
```

- `points`: Data point coordinates.
- `values`: Data values.
- `xi`: Points at which to interpolate data.
- `method`: cubic, 2D: Return the value determined from a piecewise cubic, continuously differentiable (C1), and approximately curvature-minimizing polynomial surface.
- `fill_value`: Value used to fill in for requested points outside of the convex hull of the input points.

```
from scipy.interpolate import griddata

points = sml_tups                                                    ①
values = vals[timestep].flatten()                                    ②
xi = big_tups                                                        ③
method = 'cubic'                                                     ④
fill_value = 0                                                       ⑤

arr = griddata(points, values, xi, method, fill_value)               ⑥

print(f'We have interpolated 381×1081 = {np.shape(arr)[0]} points.') ⑦
```

① The points we want to sample from.
② The values for the grid we want to sample from, flattened into a 1D array.
③ The points we want to sample to.
④ The interpolation method used ('cubic' in this case).
⑤ The fill value to use (0 instead of NaN).
⑥ Perform the interpolation.
⑦ Print the number of points interpolated.

```
We have interpolated 381×1081 = 411861 points.
```

Notice that we have interpolated values that are negative.

```
print(f'The minimum PM25 value in our interpolated values is {np.min(arr)}')
```

```
The minimum PM25 value in our interpolated values is -0.0004518490243624799
```

We change values less than our specified threshold to 0. We then reshape the values to be 381×1081 and make all the values of type float32, as this number type is used in the original NetCDF file for PM25 values.

```
arr[arr < 1e-15] = 0                                                 ①

arr = arr.reshape((len(big_lat), len(big_lon)))                      ②

arr = arr.astype(np.float32)                                         ③

print(f'The shape of the resampled PM25 array at timestep {timestep} is {np.shape(arr)}') ④
```

① Any values that are less than a given threshold, make it 0.

② Reshape the result to match the new grid shape.
③ Cast number to float32.
④ Print the shape of the resampled PM25 array at the given timestep.

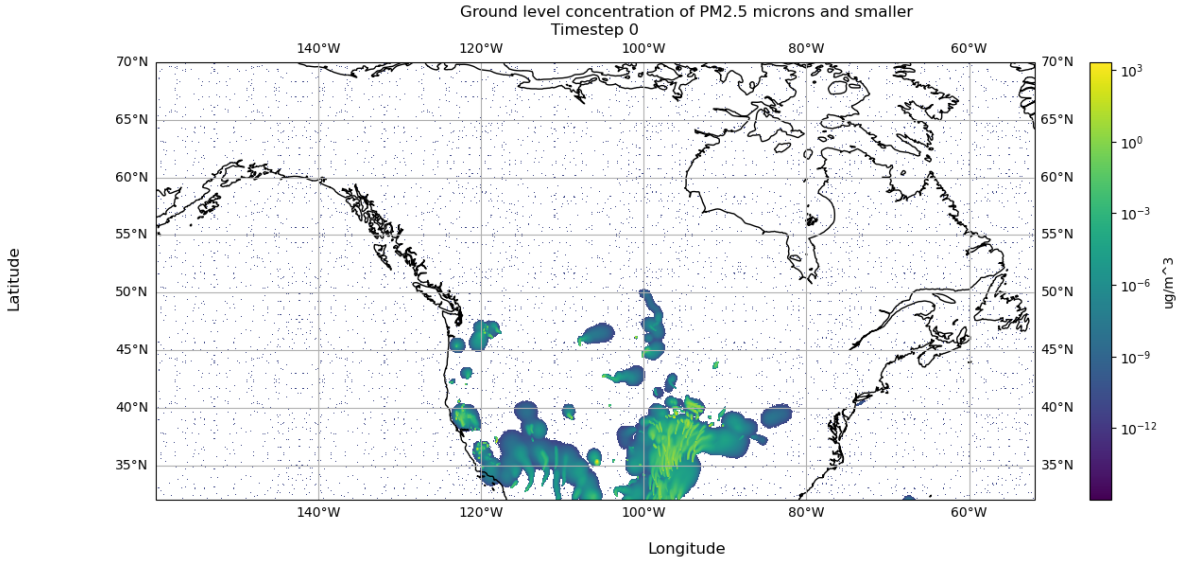
```
The shape of the resampled PM25 array at timestep 0 is (381, 1081)
```

Now we can visualize the resampled values:

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.PlateCarree())

my_norm = "log"                                                                    ②
my_extent = [np.min(big_lon), np.max(big_lon), np.min(big_lat), np.max(big_lat)] ③
my_aspect = 'auto'                                                                 ④
my_origin = 'lower'                                                                ⑤
my_cmap = 'viridis'                                                                ⑥

plot = my_plt.imshow(arr, norm=my_norm, extent=my_extent,
          aspect=my_aspect, origin=my_origin, cmap=my_cmap)                        ⑦

my_plt.coastlines()                                                                ⑧

my_plt.gridlines(draw_labels=True)                                                 ⑨

my_fig.colorbar(plot, location='right', label='ug/m^3')                            ⑩

my_fig.supxlabel('Longitude')                                                      ⑪
my_fig.supylabel('Latitude')                                                       ⑫

my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')         ⑬

my_plt.set_title(f'Timestep {timestep}')                                           ⑭

plt.show()                                                                         ⑮
```

① Initialize a figure and plot, so we can customize figure and plot of data.
② Color PM25 values on a log scale, since values are small.
③ This will number our x and y axes based on the longitude latitude range.
④ Ensure the aspect ratio of our plot fits all data, matplotlib can do this automatically.
⑤ Tell matplotlib our origin is the lower-left corner.

⑥ Select a colormap for our plot and the color bar on the right.
⑦ Create our plot using imshow.
⑧ Draw coastlines.
⑨ Draw latitude longitude lines.
⑩ Add a colorbar to our figure, based on the plot we just made above.
⑪ Set x axis label on our ax.
⑫ Set y axis label on our ax.
⑬ Set title of our figure.
⑭ Set title of our plot as the timestamp of our data.
⑮ Show the resulting visualization.



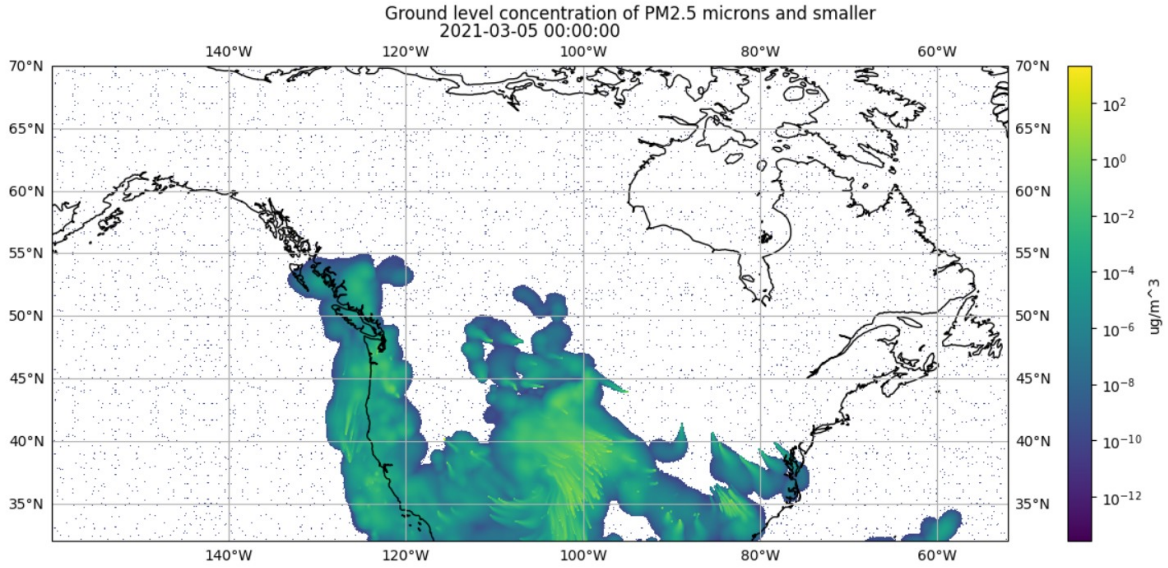
6.9.3 Rationale and Future Improvements

6.9.3.1 Discovering Varying Grids

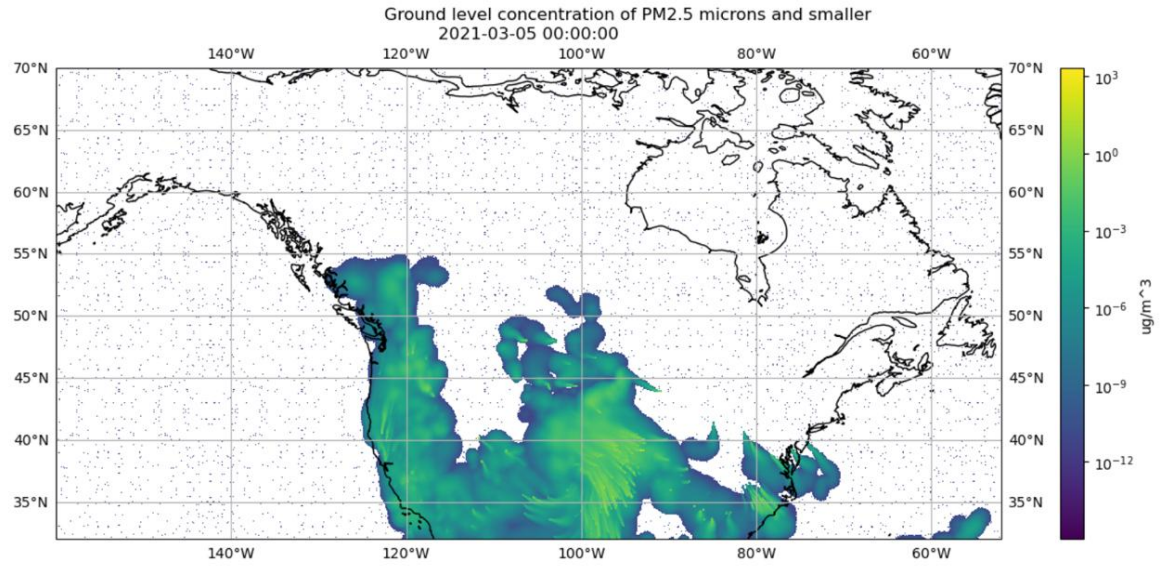
We discovered the two grid shapes in version 1 of our conversion script. We found this after noticing the smoke data visualizations were nonsensical.

For example, the visualizations showed smoke eminating from the ocean, as shown in Figure 6.1.

For further details on how we investigated this issue, see Chapter 7.



(a) Plotting Data from 381×1041 Grid on 381×1081 Grid



(b) Plotting Data Resampled from 381×1041 Grid onto 381×1081 Grid

Figure 6.1: Visualization of Timestamp March 5, 2021 00:00:00 using IDX file Created in Conversion Script Version 1

6.9.3.2 Handling Various Grids

We considered various approaches to handling the fact that the data was on a 381×1041 grid or 381×1081 grid.

Table 6.1: The Weaknesses of Approaches to Handling Varying Array Sizes

Approach	Weakness
Exclude arrays with 1041 columns.	Throwing away those data points would discard all the information they hold.
Force data with 1041 columns into an array with 1081 columns without resampling.	This results in unused columns within the 1081-column array, leading to discontinuities and potential artifacts in the data representation.
Crop arrays on 1081 columns to 1041 columns	Cropping the data would result in loss of information.

The approach we chose was to **resample the data with 1041 columns to arrays with 1081 columns**. This produced the most visually appealing result and preserved the most information possible.

6.9.3.3 Scaling

One future improvement is to generalize precomputing additional grids if in the future the smoke forecasts change their grid size again. As of now we manually compute points for the 2 grid sizes.

6.10 Sequencing of NetCDF Files

At this point we have the lists of openable files and our resampling grids. Now we will determine which files to use and in what order.

The unique challenge of UBC's short term dataset is that the forecasts overlap, creating staggered predictions as shown in Section 4.4.2. Additionally, there are often missing files or data points. In the following sections we show how we use the data given these factors.

6.10.1 Hourly Data per Forecast ID Dictionary

Recall that within each set of Forecast ID files, some files failed to download or open. Therefore, we check exactly what set of hours are available in each collection of forecast ID NetCDF files and store that information in a dictionary.

Below you can see the first few entries of this dictionary:

To generate this dictionary, we did the following:

```
id_sets = {id_: {} for id_ in ids}                                                    ①

for id_ in ids:
    for file in tqdm(successful_files[id_]):                                          ②
        path = f"{firesmoke_dir}/{id_}/{file}"                                        ③

        ds = xr.open_dataset(path)                                                    ④

        for h in range(ds.sizes["TSTEP"]):                                            ⑤
            id_sets[id_][(file, parse_tflag(ds["TFLAG"].values[h][0]))] = h
```

① Initialize a dictionary for each forecast ID.
② For all successfully opened files per forecast ID.
③ Build path string to the file.
④ Open the file with `xarray`.
⑤ For every time step, get the dictionary at key `id_`. Then, add a key, value pair. The *key* is a tuple with the current file name and the TFLAG at `h` parsed with `parse_tflag`. The *value* is the index `h`.

Now that we have an indexable set of all available hours for each forecast ID, we can generate the sequence to choose the best predictions for every time step.

6.10.2 Utility Functions

Recall that for each timestep, there are various predictions available as shown in Section 4.4.2.

The following utility functions encode the logic used for selecting the most accurate PM2.5 prediction per time step. Most accurate means, the prediction produced by the forecast run as close as possible to the time step of interest.

```
def prev_id(curr_id, verbose):
    """
    Return the string of the previous dataset ID to use based on the current ID.
    'Previous' means, last most recently updated forecast before curr_id.

    Details on forecast update time can be found here: https://firesmoke.ca/forecasts/

    Listed in order: ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]
```

```
    :param string curr_id: the ID used:
    :param boolean verbose: whether to enable print statements for debugging:
    """
    ret = ""

    if curr_id == "BSC18CA12-01":
        ret = "BSC12CA12-01"
    if curr_id == "BSC00CA12-01":
        ret = "BSC18CA12-01"
    if curr_id == "BSC06CA12-01":
        ret = "BSC00CA12-01"
    if curr_id == "BSC12CA12-01":
        ret = "BSC06CA12-01"

    if verbose:
        print(f"prev_id({curr_id}) = {ret}")

    return ret
```

```
def get_id_from_date(date, verbose):
    """
    Return the string of the dataset ID to use based on the date and hour given.

    We aim to use the dataset that provides the latest forecast update available for the hour

    Details on forecast update time can be found here: https://firesmoke.ca/forecasts/

    :param datetime date: pandas timestamp of the YYYYMMDD 00:00:00 date:
    :param boolean verbose: whether to enable print statements for debugging:
    """
    ret = ""

    if date <= date.replace(hour=2):                                    (1)
        ret = "BSC12CA12-01"
    if date >= date.replace(hour=3) and date <= date.replace(hour=8):
        ret = "BSC18CA12-01"
    if date >= date.replace(hour=9) and date <= date.replace(hour=14):
        ret = "BSC00CA12-01"
    if date >= date.replace(hour=15) and date <= date.replace(hour=20):
        ret = "BSC06CA12-01"
    if date >= date.replace(hour=21):
```

```
        ret = "BSC12CA12-01"

    if verbose:
        print(f"get_id_from_date({date}) = {ret}")
    return ret
```

① Based on the given `date`, use the optimal forecast ID.

```
def dispersion_date_str(date, id_, verbose):
    """
    For a given date object and forecats ID, generate the name for the dispersion file in wh
    :param pd.Timestamp date: pandas timestamp of the date to make file name string out of:
    :param string id_: string with the dataset id to use
    :param boolean verbose: whether to enable print statements for debugging:
    """
    ret = ""

    if id_ == "BSC12CA12-01":                                                    ①
        new_date = date + datetime.timedelta(days=-1)
        ret = f'dispersion_{new_date.strftime("%Y%m%d")}.nc'
    else:                                                                        ②
        ret = f'dispersion_{date.strftime("%Y%m%d")}.nc'

    if verbose:
        print(f"dispersion_date_str({date}, {id_}) = {ret}")

    return ret
```

① BSC00CA12-01 generates the first hours of the given `date` in 'yesterday's' file. For example, the hours 12am-6am for January 2, 2023 are generated in dispersion_01012023.nc in the BSC00CA12-01 dataset.

② For all other forecast IDs, the optimal hour for all hours is in "today's" file, where 'today' is the `date` given.

6.10.3 Populating the `idx_calls` Array

Here, we describe how we use our utility functions to parse all the NetCDF files we have to populate our `idx_calls` array. `idx_calls` is used in the final step by defining *which* predictions to load and in what *order*. See Section 6.11 for details on exact usage.

First, we initialize our variables.

```
idx_calls = []                                                                    ①

start_date = datetime.datetime.strptime("20210304", "%Y%m%d")                     ②
end_date = datetime.datetime.strptime("20240627", "%Y%m%d")

current_date = start_date                                                         ③
current_hour = datetime.datetime(current_date.year, current_date.month, current_date.day)

file_str = ''                                                                     ④

verbose = 1                                                                       ⑤
```

① Arrays to hold the final order we will index files
② Define the start and end dates we will step through.
③ Initialize these variables to the start date and time. We use these to increment through to the `end_date` next.
④ Initialize a variable for holding the current file name in use.
⑤ Tell utility functions to print for debugging.

Then, we use a `while` loop to populate `idx_calls` with the optimal sequence.

```
while current_date <= end_date:                                                   ①
    while current_hour < current_date + datetime.timedelta(days=1):               ②
        prev_day_count = 0
        found = 0

        while found == 0 and prev_day_count <= 4:                                 ③
            curr_date = current_hour + datetime.timedelta(days=-prev_day_count)   ④

            curr_id = get_id_from_date(curr_date, verbose)                        ⑤

            forecast_search_count = 0

            while found == 0 and forecast_search_count < 4:                       ⑥
                file_str = dispersion_date_str(curr_date, curr_id, verbose)       ⑦

                if (file_str, current_hour) in id_sets[curr_id]:                  ⑧
                    update_idx_calls(
                        idx_calls, curr_id, (file_str, current_hour), id_sets     ⑨
                    )
                    found = 1
                else:
```

```
                    forecast_search_count += 1

                curr_id = prev_id(curr_id, verbose)                    (10)

            prev_day_count += 1                                        (11)

        current_hour += datetime.timedelta(hours=1)                    (12)

    current_date += datetime.timedelta(days=1)                         (13)
```

① Iterate from `current_date` by one hour increments until the `end_date` is reached.
② Iterate through each hour until the next day starts.
③ If no optimal prediction is found and not more than 4 days have been searched.
④ Subtract `prev_day_count` from `current_hour` to test earlier dates.
⑤ Retrieve the forecast ID corresponding to the `curr_date` being tested.
⑥ Loop through potential forecasts until one is found or 4 different forecasts have been checked.
⑦ Construct the filename string for the forecast dataset corresponding to `curr_date` and `curr_id`.
⑧ Verify if the `file_str` and `current_hour` combination exists in `id_sets` for the given `curr_id`.
⑨ If available, record the forecast ID and time in `idx_calls`, and mark this combination as 'found.'
⑩ If the current ID doesn't have the forecast, move to a previous forecast ID.
⑪ Increment `prev_day_count` and repeat the search for an earlier day.
⑫ Increment `current_hour` to move to the next hour in the `current_date`.
⑬ After completing all hours of a day, move to the next day.

The following shows the first 2 and final 2 entires of the `idx_calls` array.

```
[['BSC12CA12-01',
  'dispersion_20210303.nc',
  datetime.datetime(2021, 3, 4, 0, 0),
  3],
 ['BSC12CA12-01',
  'dispersion_20210303.nc',
  datetime.datetime(2021, 3, 4, 1, 0),
  4],
 ...
 ['BSC12CA12-01',
  'dispersion_20240626.nc',
  datetime.datetime(2024, 6, 27, 22, 0),
```

```
  25],
 ['BSC12CA12-01',
  'dispersion_20240626.nc',
  datetime.datetime(2024, 6, 27, 23, 0),
  26]]
```

6.11 Creating the IDX File

At this point, we have precomputed the order in which we will load and write each array to our IDX file. We will now use `idx_calls` to create the final IDX file containing our single dataset.

```
f = Field("PM25", "float32")                                                    ①

db = CreateIdx(                                                                 ②
    url=idx_dir + "/firesmoke.idx",
    fields=[f],
    dims=[int(max_ncols[ids[0]]), int(max_nrows[ids[0]])],
    time=[0, len(idx_calls) - 1, "%00000000d/"],
)

tstep = 0                                                                       ③

thresh = 1e-15                                                                  ④

for call in tqdm(idx_calls):                                                    ⑤
    curr_id = call[0]
    curr_file = call[1]
    tstep_index = call[3]

    ds = xr.open_dataset(f"{firesmoke_dir}/{curr_id}/{curr_file}")              ⑥

    file_vals = np.squeeze(ds["PM25"].values)                                   ⑦

    resamp = ds.XORIG != max_xorig                                              ⑧

    if resamp:                                                                  ⑨
        file_vals_resamp = griddata(
            sml_tups,
            file_vals[tstep_index].flatten(),
```

```
            big_tups,
            method="cubic",
            fill_value=0,
        )

        file_vals_resamp[file_vals_resamp < thresh] = 0                    (10)

        file_vals_resamp = file_vals_resamp.reshape((len(big_lat), len(big_lon))) (11)

        db.write(data=file_vals_resamp.astype(np.float32), field=f, time=tstep) (12)
    else:                                                                  (13)
        db.write(data=file_vals[tstep_index], field=f, time=tstep)

    tstep = tstep + 1                                                      (14)
```

(1) Create an OpenVisus field to hold the PM25 variable data.
(2) Create the IDX file wherein `url` is the location to write the file, `fields` holds the data variables we will save, `dims` represents the shape of each array, and `time` defines how many time steps there are.
(3) We will usethis to keep track of which time step we are on as we step through our `idx_calls`.
(4) Threshold to use to change small-enough resampled values to 0.
(5) Get the information for current time step, in particular the [curr_id, file_str, parse_tflag(ds['TFLAG'].values[tstep_idx][0]), tstep_idx]
(6) Load the current file with `xarray`.
(7) Get the full array of PM25 values in the file.
(8) If `ds.XORIG` is not already for the 381×1081 grid, we need to resample it to the larger grid.
(9) Using `gridddata`, interpolate the values on a 381×1081 grid using the precomputed lat/lon points.
(10) Any values that are less than our threshold should have a value of 0. WHY
(11) Reshape the interpolated values to 381×1081.
(12) Write the resampled values for hour `h` to timestep `t` and field `f` of our IDX file.
(13) These values are already on a 381×1081 grid, so write the values at hour `h` to timestep `t` and field `f` of our IDX file.
(14) Increment to the next timestep for writing to IDX.

7 Data Validation

The success of this data curation project requires competent data validation. Data validation is the bridge between theoretical and real-world data curation. We unfortunately used many assumptions about the data and systems we used that did not hold true, leading to unexplainable issues throughout the data curation process.

Here, we will describe our deficiencies in performing data validation and the consequences we faced. We conclude with a discussion on how best data validation can be incorporated in this project and future such projects, so that one may avoid such issues next time.

7.1 Data Validation vs Data Exploration

Data exploration enlightens one on the contents of the data and metadata one presumes they have. We performed data exploration by loading and visualizing a few files. This allowed us to understand what data UBC aims to provide.

What data exploration *does not* do is explain the origin of issues such missing or seemingly corrupt data. Data exploration uses the assumption that the data is perfect.

Data validation forces one to consider, where do issues that appear in the data come from, and are these issues with the data or with the systems used to access and manipulate the data?

7.2 Data Validation for Data Loading

Here, we will describe how we discovered the consequences of failing to validate that the files we downloaded were true NetCDF files instead of HTML webpages, as described in Chapter 5.

7.2.1 The Problem

When doing rudimentary visualizations we saw missing timesteps from our final dataset in the IDX file. We assumed that any missing time steps were due to the files being unavailable from the data source. However, we decided to validate which time steps were missing and why, in case the cause for apparently missing time steps was our fault.

7.2.2 Visualizing all Timesteps

To determine exactly which time steps were unavailable, we decided to load and visualize every timestep from March 3, 2021 to June 27, 2024 as a video. We then identify which time steps fail to load or visualize and diagnose *why*. The scripts we proceed to describe can be found in the side bar or here.

In the following scripts, we generate PNG images for every time step in our IDX file *and* for every time step directly loaded from the downloaded NetCDF files. The time steps we load are the same ones specified in our `idx_calls` array from Chapter 6.

We load from both our IDX file and NetCDF files to crosscheck any issues we encounter in both file formats.

8 IDX File PNGs

9 Create .PNG images of all timesteps in IDX firesmoke dataset

9.1 Import necessary libraries

```
import numpy as np                                                          ①
import os                                                                   ②
import requests                                                             ③
import xarray as xr                                                         ④
from openvisuspy.xarray_backend import OpenVisusBackendEntrypoint           ⑤
import time                                                                 ⑥
import datetime
import pandas as pd                                                         ⑦
import matplotlib                                                           ⑧
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import pickle                                                               ⑨
os.environ["VISUS_CACHE"]="./visus_cache_can_be_erased"                     ⑩
from tqdm import tqdm                                                       ⑪
from OpenVisus import *                                                     ⑫
```

① For numerical work
② For accessing the file system
③ For downloading the latest firesmoke NetCDF
④ For loading NetCDF files and metadata
⑤ For connecting the OpenVisus framework to xarray (from openvisuspy)
⑥ Used for processing NetCDF time data
⑦ Used for indexing via metadata
⑧ For plotting
⑨ For exporting the dictionary of issue files at the end of the notebook
⑩ Stores the OpenVisus cache in the local directory
⑪ Accessory, used to generate a progress bar for running for loops
⑫ For importing OpenVisus functions

9.1.1 In this section, we load our data using `xr.open_dataset`.

```
url = 'https://github.com/sci-visus/NSDF-WIRED/raw/main/data/firesmoke_metadata_recent.nc' ①

response = requests.get(url)                                                              ②
local_netcdf = 'firesmoke_metadata.nc'                                                    ③
with open(local_netcdf, 'wb') as f:                                                       ④
    f.write(response.content)

ds = xr.open_dataset(local_netcdf, engine=OpenVisusBackendEntrypoint)                     ⑤
```

① Path to the tiny NetCDF file
② Download the file using `requests`
③ Local filename for the NetCDF file
④ Write the downloaded content to the local file system
⑤ Open the tiny NetCDF file with xarray and the OpenVisus backend

9.2 Calculate derived metadata using original metadata above to create coordinates

9.2.1 This is required to allow for indexing of data via metadata

9.2.1.1 Calculate latitude and longitude grid

```
xorig = ds.XORIG
yorig = ds.YORIG
xcell = ds.XCELL
ycell = ds.YCELL
ncols = ds.NCOLS
nrows = ds.NROWS

longitude = np.linspace(xorig, xorig + xcell * (ncols - 1), ncols)
latitude = np.linspace(yorig, yorig + ycell * (nrows - 1), nrows)
```

9.2.1.2 Using calculated latitude and longitude, create coordinates allowing for indexing data using lat/lon

```
ds.coords['lat'] = ('ROW', latitude)                                    ①
ds.coords['lon'] = ('COL', longitude)                                   ②

ds = ds.swap_dims({'COL': 'lon', 'ROW': 'lat'})                         ③
```

① Create coordinates for latitude based on the `ROW` dimension
② Create coordinates for longitude based on the `COL` dimension
③ Replace the `COL` and `ROW` dimensions with the newly calculated longitude and latitude arrays

9.3 Get timestamps to label video frames

Need to use `idx_calls` generated during conversion

```
with open('idx_calls_v4.pkl', 'rb') as f:                               ①
    idx_calls = pickle.load(f)
```

① Load idx_calls from file

9.3.0.0.1 Return an array of the tflags as pandas timestamps

```
timestamps = []                                                         ①

for call in idx_calls:                                                  ②
    timestamps.append(pd.Timestamp(call[2]))                            ③
```

① Initialize an empty list to store pandas timestamps
② Loop through the `idx_calls` to process each call
③ Convert the `tflags` to pandas `Timestamp` and store in the `timestamps` list

9.4 Create the video

```
data_resolution = 0                                                                    ①
folder = "/usr/sci/scratch_nvme/arleth/dump/idx_frames"                                ②

my_norm = "log"                                                                        ③
my_extent = [np.min(longitude), np.max(longitude), np.min(latitude), np.max(latitude)]
my_aspect = 'auto'
my_origin = 'lower'
my_cmap = 'hot'

issue_files = {}                                                                       ④

for i in tqdm(range(len(idx_calls))):                                                  ⑤
    data_array_at_time = ds['PM25'].loc[i, :, :, data_resolution]                      ⑥

    try:                                                                               ⑦
        my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.Plate
        plot = my_plt.imshow(data, norm=my_norm, extent=my_extent, aspect=my_aspect, origin=
        my_fig.colorbar(plot, location='right', label='ug/m^3')
        my_plt.coastlines()
        my_plt.gridlines(draw_labels=True)
        my_fig.suptitle(f'Ground level concentration of PM2.5 microns and smaller {timestamps
        my_plt.text(0.5, -0.1, 'IDX Data', ha='center', va='center', transform=my_plt.transA

        plt.savefig(folder + "/frames%05d.png" % i, dpi=280)                           ⑧
        plt.close(my_fig)
        matplotlib.pyplot.close()
    except:                                                                            ⑨
        issue_files[timestamps[i]] = data
        continue
```

① Set the resolution level of the `PM25` data to max
② Directory of environment to save frames
③ Set parameters for creating visualization of each timestep with matplotlib.
④ Dictionary to keep track of files with 'issues'.
⑤ For all timesteps create visualization of firesmoke at time.
⑥ Get PM2.5 values and provide 4 values, the colons mean select all lat and lon indices.
⑦ Try creating the visualization or catch exceptions accordingly.
⑧ Save images to file.
⑨ Print exception if one is found and save issue in issue dictionary using timestamp `t` as key.

```
with open('new_idx_issues.pkl', 'wb') as f:                                            ①
    pickle.dump(issue_files, f)
```

① Save ‘issue_files’ to review

10 NetCDF Files PNGs

11 Create .PNG images of all timesteps from `idx_calls` loading from netCDF files

11.1 Import necessary libraries

```
import numpy as np                                    ①
import os                                             ②
import xarray as xr                                   ③
import time                                           ④
import datetime
import pandas as pd                                   ⑤
import matplotlib                                     ⑥
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import pickle                                         ⑦
from tqdm import tqdm                                 ⑧
```

① For numerical work
② For accessing file system
③ For loading NetCDF files, for metadata
④ Used for processing netCDF time data
⑤ Used for indexing via metadata
⑥ For plotting
⑦ For exporting the dictionary of issue files at the end of notebook and importing `idx_calls.pkl`
⑧ Accessory, used to generate progress bar for running for loops

11.2 Get path to original firesmoke data

```
netcdf_dir = "/usr/sci/cedmav/data/firesmoke"
```

```
ids = ["BSC18CA12-01", "BSC00CA12-01", "BSC06CA12-01", "BSC12CA12-01"]      ①
start_dates = ["20210304", "20210304", "20210304", "20210303"]
end_dates = ["20240627", "20240627", "20240627", "20240627"]

id_dates = {ids[i]: {"start_date": start_dates[i], "end_date": end_dates[i]} for i in range(
```

① Date ranges for each smoke forecast dataset.
② Create dictionary of all file names using information above

11.2.1 In this section, we load metadata from 381x1041 and 381x1081 files using `xr.open_dataset`.

```
file_s = f'{netcdf_dir}/{ids[1]}/dispersion_20210304.nc'                    ①
file_b = f'{netcdf_dir}/{ids[1]}/dispersion_20240101.nc'

ds_s = xr.open_dataset(file_s)                                              ②
ds_b = xr.open_dataset(file_b)
```

① Path to small and big files
② Open metadata of each file

11.3 Calculate derived metadata using original metadata above to create coordinates

11.3.1 We'll use this for creating our visualizations

11.3.1.1 Calculate latitude and longitude grid for each set of files' metadata

```
longitude_s = np.linspace(ds_s.XORIG, ds_s.XORIG + ds_s.XCELL * (ds_s.NCOLS - 1), ds_s.NCOLS)
latitude_s = np.linspace(ds_s.YORIG, ds_s.YORIG + ds_s.YCELL * (ds_s.NROWS - 1), ds_s.NROWS)

longitude_b = np.linspace(ds_b.XORIG, ds_b.XORIG + ds_b.XCELL * (ds_b.NCOLS - 1), ds_b.NCOLS)
latitude_b = np.linspace(ds_b.YORIG, ds_b.YORIG + ds_b.YCELL * (ds_b.NROWS - 1), ds_b.NROWS)
```

11.3.1.2 The timestamps used in the files may not be intuitive. The following utility function returns the desired pandas timestamp based on your date and time of interest.

11.3.1.2.1 When you index the data at a desired time, use this function to get the timestamp you need to index.

```
def parse_tflag(tflag):
    """
    Return the tflag as a datetime object
    :param list tflag: a list of two int32, the 1st representing date and 2nd representing ti
    """
    date = int(tflag[0])                                                                    ①
    year = date // 1000
    day_of_year = date % 1000

    final_date = datetime.datetime(year, 1, 1) + datetime.timedelta(days=day_of_year - 1)  ②

    time = int(tflag[1])                                                                    ③
    hours = time // 10000
    minutes = (time % 10000) // 100
    seconds = time % 100

    full_datetime = datetime.datetime(year, final_date.month, final_date.day, hours, minutes
    return full_datetime
```

① Obtain year and day of year from tflag[0] (date)
② Create datetime object representing date
③ Obtain hour, mins, and secs from tflag[1] (time)
④ Create final datetime object

```
def get_timestamp(year, month, day, hour):
    """
    return a pandas timestamp using the given date-time arguments
    :param int year: year
    :param int month: month
    :param int day: day
    :param int hour: hour
    """
    full_datetime = datetime.datetime(year, month, day, hour)                               ①

    year = full_datetime.year                                                               ②
```

```
    day_of_year = full_datetime.timetuple().tm_yday
    hours = full_datetime.hour
    minutes = full_datetime.minute
    seconds = full_datetime.second

    tflag0 = year * 1000 + day_of_year                                    ③
    tflag1 = hours * 10000 + minutes * 100 + seconds

    return pd.Timestamp(full_datetime)                                    ④
```

① Convert year, month, day, and hour to a datetime object
② Extract components from the datetime object
③ Compute tflag[0] and tflag[1]
④ Return the Pandas Timestamp object

11.4 Import sequence of data slices to get at what time step

```
with open('idx_calls_v4.pkl', 'rb') as f:                                 ①
    idx_calls = pickle.load(f)
```

① Load idx_calls from file

11.5 Create the video

```
folder = "/usr/sci/scratch_nvme/arleth/dump/netcdf_frames"                ①

my_norm = "log"                                                           ②
my_extent_s = [np.min(longitude_s), np.max(longitude_s), np.min(latitude_s), np.max(latitude_
my_extent_b = [np.min(longitude_b), np.max(longitude_b), np.min(latitude_b), np.max(latitude_
my_aspect = 'auto'
my_origin = 'lower'
my_cmap = 'hot'

issue_files = {}                                                          ③

frame_num = 0                                                             ④
```

```
for call in tqdm(idx_calls):                                                    (5)
    curr_id = call[0]                                                           (6)
    curr_file = call[1]
    curr_date = call[2]
    tstep_index = call[3]

    ds = xr.open_dataset(f'{netcdf_dir}/{curr_id}/{curr_file}')                 (7)

    ds_vals = np.squeeze(ds['PM25'].values)                                     (8)

    data_at_time = ds_vals[tstep_index]                                         (9)

    t = pd.Timestamp(parse_tflag(ds['TFLAG'].values[tstep_index][0]))           (10)

    try:                                                                        (11)
        my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.Plate
        curr_extent = my_extent_s if ds['PM25'].shape[3] == 1041 else my_extent_b (12)
        plot = my_plt.imshow(data_at_time, norm=my_norm, extent=curr_extent, aspect=my_aspec
        my_fig.colorbar(plot,location='right', label='ug/m^3')
        my_plt.coastlines()
        my_plt.gridlines(draw_labels=True)
        my_fig.suptitle(f'Ground Level Concentration of PM2.5 Microns and Smaller\n{t}') (13)

        my_plt.text(0.5, -0.1, 'Original NetCDF Data', ha='center', va='center', transform=my

        plt.savefig(folder + "/frames%05d.png" % frame_num, dpi=280)            (15)
        plt.close(my_fig);                                                      (16)
        matplotlib.pyplot.close()
    except:                                                                     (17)
        print(f"issue! {t}")
        issue_files[t] = data_at_time
        continue
    frame_num = frame_num + 1                                                   (18)
```

① Directory of environment to save frames
② Set parameters for creating visualization of each timestep with matplotlib.
③ Dictionary to keep track of files with 'issues'.
④ To track what frame we're on in the following loop.
⑤ For all timesteps create visualization of firesmoke at time.
⑥ Get instructions from call.
⑦ Open the current file with xarray.
⑧ Get the PM25 values, squeeze out empty axis.

⑨ Get PM2.5 values at tstep_index and visualize them.
⑩ Get the timestamp for titling our plot, use hour 'h'.
⑪ Catch exceptions accordingly.
⑫ Extent is either with the 381x1041 lons/lats or 381x1081 lons/lats.
⑬ Add a title with the time information.
⑭ Add an additional caption for context.
⑮ Save the visualization as a frame.
⑯ Close the figure after saving.
⑰ Print exception if one is found and save issue in issue dictionary using timestamp `t` as key.
⑱ Whether exception or not, next frame count to align with idx script.

```
with open('new_netcdf_issues.pkl', 'wb') as f:                    ①
    pickle.dump(issue_files, f)
```

① Save 'issue_files' to review

Now with all images for each time step generated, we create videos to save to file.

12 Create videos using .PNG images generated

```
# ref: https://stackoverflow.com/questions/43048725/python-creating-video-from-images-using-o
import cv2                                                                    ①
import numpy as np                                                            ②
import os                                                                     ③
import time                                                                   ④
```

① For creating the video
② For numerical work
③ For accessing file system
④ Used for processing NetCDF time data

```
def make_video(img_dir, video_dir, video_name):
    '''
    Create a video made of frames at img_dir and save to video_dir with the name video_name
    :param
    '''
    # ref: https://stackoverflow.com/questions/27593227/listing-png-files-in-folder
    images = [img for img in os.listdir(img_dir) if img.endswith(".png") and img.startswith('
    images = np.sort(images)

    frame = cv2.imread(os.path.join(img_dir, images[0]))                      ②
    height, width, layers = frame.shape

    video = cv2.VideoWriter(video_dir + video_name,cv2.VideoWriter_fourcc(*'mp4v'), 10, (widt

    start_time = time.time()                                                  ③

    for img in images:                                                        ④
        frame = cv2.imread(f'{img_dir}{img}')
        video.write(frame)

    end_time = time.time()                                                    ⑤
```

```
    execution_time = end_time - start_time

    print(f"Total execution time: {execution_time:.2f} seconds")          ⑥

    cv2.destroyAllWindows()                                               ⑦
    video.release()
```

① Generate list of images sorted by name, this is chronological order of timesteps.
② Initialize to first frame.
③ Record start time.
④ Write each image to video.
⑤ Record end time.
⑥ Print execution time.
⑦ Close applications.

```
idx_image_folder = '/usr/sci/scratch_nvme/arleth/dump/idx_frames/'        ①
idx_video_folder = '/usr/sci/scratch_nvme/arleth/dump/videos/'
idx_vid_name = 'idx_video_7_10_24.mp4'

netcdf_image_folder = '/usr/sci/scratch_nvme/arleth/dump/netcdf_frames/'  ②
netcdf_video_folder = '/usr/sci/scratch_nvme/arleth/dump/videos/'
netcdf_vid_name = 'netcdf_video_7_10_24.mp4'
```

① Directories to IDX .PNGs and video name.
② Directories to netCDF .PNGs and video name.

Visually inspecting these videos allowed us to explore where significant bouts of missing time-series data were missing.

12.1 Data Conversion

13 NetCDF Visualization Demo

In this demo we load a different `dispersion.nc` file and explore how to visualize the data within the file.

13.0.0.1 Accessing the File

We use the forecast for March 4, 2021 where the weather forecast is initiated at 00:00:00 UTC and the smoke forecast is initialized at 08:00:00 UTC. You can download this file by navigating to the URL below.

```
forecast_id = "BSC00CA12-01"
yyyymmdd = "20210304"
init_time = "08"

url = (
    f"https://firesmoke.ca/forecasts/{forecast_id}/{yyyymmdd}{init_time}/dispersion.nc"
)

print(f"Download this file from URL: {url}")

# import urllib.request
# urllib.request.urlretrieve(url, "dispersion.nc")
```

```
Download this file from URL: https://firesmoke.ca/forecasts/BSC00CA12-01/2021030408/dispersi
```

13.0.0.1.1 Opening the File

We use xarray to open the NetCDF file and preview it.

```
import xarray as xr

ds = xr.open_dataset("dispersion.nc")

ds
```

```
<xarray.Dataset>
Dimensions:  (TSTEP: 51, VAR: 1, DATE-TIME: 2, LAY: 1, ROW: 381, COL: 1041)
Dimensions without coordinates: TSTEP, VAR, DATE-TIME, LAY, ROW, COL
Data variables:
    TFLAG    (TSTEP, VAR, DATE-TIME) int32 ...
    PM25     (TSTEP, LAY, ROW, COL) float32 ...
Attributes: (12/33)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                    ...
    EXEC_ID:        ????????????????                                         ...
    FTYPE:          1
    CDATE:          2021063
    CTIME:          101914
    WDATE:          2021063
    ...             ...
    VGLVLS:         [10.  0.]
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                       ...
    HISTORY:
```

13.0.0.2 Using the Data

13.0.0.2.1 Accessing Arrays

The data we are interested in is the PM2.5 values. Let's use `xarray` to get the array in the `PM25` variable.

```
ds["PM25"]
```

```
<xarray.DataArray 'PM25' (TSTEP: 51, LAY: 1, ROW: 381, COL: 1041)>
[20227671 values with dtype=float32]
Dimensions without coordinates: TSTEP, LAY, ROW, COL
Attributes:
    long_name:  PM25
    units:      ug/m^3
    var_desc:   PM25                                                         ...
```

The dimensions of the `PM25` data array are composed of `TSTEP`, `LAY`, `ROW`, and `COL`. We do not need the `LAY` dimension, so let's use `numpy` to drop it.

```
import numpy as np

ds_pm25_vals = ds["PM25"].values                                                    ①
print(f'The shape of the data contained in PM25 variable is: {np.shape(ds_pm25_vals)}')

ds_pm25_vals = np.squeeze(ds_pm25_vals)                                             ②
print(f'After squeezing, the shape is: {np.shape(ds_pm25_vals)}')
```

① Use `.values` to get the four dimensional array.
② Use `np.squeeze` to drop the LAY axis

```
The shape of the data contained in PM25 variable is: (51, 1, 381, 1041)
After squeezing, the shape is: (51, 381, 1041)
```

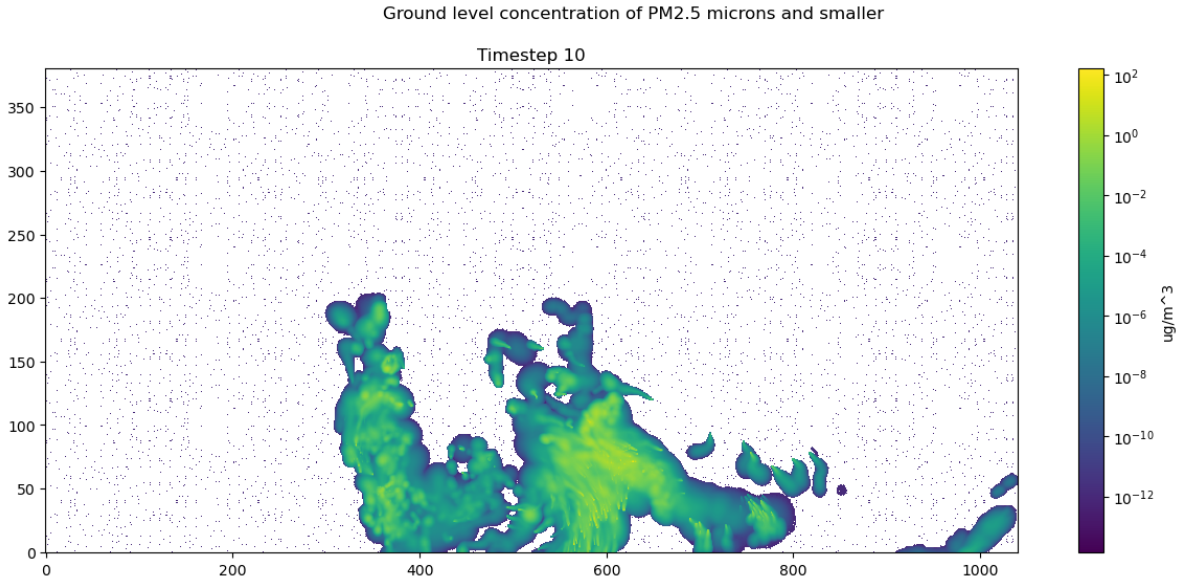
We now have `ds_pm25_vals`. Next, let's select a time step and visualize the data.

13.0.0.2.2 Visualize Array in `matplotlib`

We can index time step 10 and use `matplotlib` to visualize the timestep

```
import matplotlib.pyplot as plt

tstep = 10                                                                          ①
smoke_at_tstep = ds_pm25_vals[tstep, :, :]

my_fig, my_plt = plt.subplots(figsize=(15, 6))

my_norm = "log"                                                                     ②
my_aspect = 'auto'                                                                  ③
my_origin = 'lower'                                                                 ④
my_cmap = 'viridis'                                                                 ⑤

plot = my_plt.imshow(smoke_at_tstep, norm=my_norm, aspect=my_aspect, origin=my_origin, cmap=m

my_fig.colorbar(plot, location='right', label='ug/m^3')                             ⑦

my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')          ⑧

my_plt.set_title(f'Timestep {tstep}')                                               ⑨

plt.show()                                                                          ⑩
```

① Index `ds_pm25_vals` at `TSTEP = 10`, selecting all `ROW`s and `COL`s.
② Color PM25 values on a log scale, since values are small.
③ Ensure the aspect ratio of our plot fits all data, `matplotlib` can do this automatically.
④ Tell `matplotlib` our origin is the lower-left corner.
⑤ Select a colormap for our plot and draw the color bar on the right.
⑥ Create our plot using `imshow`.
⑦ Add a colorbar to our figure, based on the plot we just made above.
⑧ Set title of our figure.
⑨ Set title of our plot as the timestamp of our data.
⑩ Show the resulting visualization.



Notice there are no axis labels or metadata presented here. Next we will show how to use the metadata in `dispersion.nc` so the data is actually interpretable.

13.0.0.3 Incorporating Metadata to Visualization via Coordinates

13.0.0.3.1 Latitude and Longitude Coordinates

`dispersion.nc` includes attributes to generate the latitude and longitude values on the grid defined by `NCOLS` and `NROWS`. We use this grid to match each data point in the `PM25` variable to a lat/lon coordinate.

```
xorig = ds.XORIG
yorig = ds.YORIG
xcell = ds.XCELL
```

```
ycell = ds.YCELL
ncols = ds.NCOLS
nrows = ds.NROWS

longitude = np.linspace(xorig, xorig + xcell * (ncols - 1), ncols)
latitude = np.linspace(yorig, yorig + ycell * (nrows - 1), nrows)

print("Size of longitude & latitude arrays:")
print(f'np.size(longitude) = {np.size(longitude)}')
print(f'np.size(latitude) = {np.size(latitude)}\n')
print("Min & Max of longitude and latitude arrays:")
print(f'longitude: min = {np.min(longitude)}, max = {np.max(longitude)}')
print(f'latitude: min = {np.min(latitude)}, max = {np.max(latitude)}')
```

```
Size of longitude & latitude arrays:
np.size(longitude) = 1041
np.size(latitude) = 381

Min & Max of longitude and latitude arrays:
longitude: min = -156.0, max = -51.999998450279236
latitude: min = 32.0, max = 70.00000056624413
```

`xarray` allows us to create coordinates, which maps variable values to a value of our choice. In this case, we create coordinates mapping PM25 values to a latitude and longitude value.

```
ds.coords['lat'] = ('ROW', latitude)                                    ①
ds.coords['lon'] = ('COL', longitude)

ds = ds.swap_dims({'COL': 'lon', 'ROW': 'lat'})                         ②

ds
```

① Create coordinates for latitude and longitude.
② Replace `COL` and `ROW` dimensions with newly calculated longitude and latitude coordinates.

```
<xarray.Dataset>
Dimensions:  (TSTEP: 51, VAR: 1, DATE-TIME: 2, LAY: 1, lat: 381, lon: 1041)
Coordinates:
  * lat      (lat) float64 32.0 32.1 32.2 32.3 32.4 ... 69.6 69.7 69.8 69.9 70.0
  * lon      (lon) float64 -156.0 -155.9 -155.8 -155.7 ... -52.2 -52.1 -52.0
Dimensions without coordinates: TSTEP, VAR, DATE-TIME, LAY
```

```
Data variables:
    TFLAG    (TSTEP, VAR, DATE-TIME) int32 ...
    PM25     (TSTEP, LAY, lat, lon) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Attributes: (12/33)
    IOAPI_VERSION:  $Id: @(#) ioapi library version 3.0 $                    ...
    EXEC_ID:        ????????????????                                         ...
    FTYPE:          1
    CDATE:          2021063
    CTIME:          101914
    WDATE:          2021063
    ...             ...
    VGLVLS:         [10.  0.]
    GDNAM:          HYSPLIT CONC
    UPNAM:          hysplit2netCDF
    VAR-LIST:       PM25
    FILEDESC:       Hysplit Concentration Model Output                       ...
    HISTORY:
```

Now let's move on to incorporating time stamp metadata.

13.0.0.3.2 Time Coordinates

Recall, there is a `TFLAG` variable in `dispersion.nc`.

```
ds['TFLAG']
```

```
<xarray.DataArray 'TFLAG' (TSTEP: 51, VAR: 1, DATE-TIME: 2)>
[102 values with dtype=int32]
Dimensions without coordinates: TSTEP, VAR, DATE-TIME
Attributes:
    units:      <YYYYDDD,HHMMSS>
    long_name:  TFLAG
    var_desc:   Timestep-valid flags:  (1) YYYYDDD or (2) HHMMSS             ...
```

The earliest and latest `TFLAG`s look like the following:

```
print(f"Earliest available TFLAG is {ds['TFLAG'].values[0][0]}")
print(f"Latest available TFLAG is {ds['TFLAG'].values[-1][0]}")
```

```
Earliest available TFLAG is [2021063   90000]
Latest available TFLAG is [2021065  110000]
```

This time flags require processing to be immediately legible. Let's write a function to process the time flag accordingly. We use the `datetime` library.

```
import datetime

def parse_tflag(tflag):
    """
    Return the tflag as a datetime object
    :param list tflag: a list of two int32, the 1st representing date and 2nd representing ti
    """
    date = int(tflag[0])                                                    ①
    year = date // 1000                                                     ②
    day_of_year = date % 1000                                               ③

    final_date = datetime.datetime(year, 1, 1) + datetime.timedelta(days=day_of_year - 1) ④

    time = int(tflag[1])                                                    ⑤
    hours = time // 10000                                                   ⑥
    minutes = (time % 10000) // 100                                         ⑦
    seconds = time % 100                                                    ⑧

    full_datetime = datetime.datetime(year, final_date.month, final_date.day, hours, minutes
    return full_datetime
```

① Obtain year and day of year from `tflag[0]` (date).
② Extract the year from the first 4 digits of `tflag[0]`.
③ Extract the day of the year from the last 3 digits of `tflag[0]`.
④ Create a `datetime` object representing the date.
⑤ Obtain hour, minutes, and seconds from `tflag[1]` (time).
⑥ Extract hours from the first 2 digits of `tflag[1]`.
⑦ Extract minutes from the 3rd and 4th digits of `tflag[1]`.
⑧ Extract seconds from the last 2 digits of `tflag[1]`.
⑨ Create the final `datetime` object with the extracted date and time components.

Now we have datetime objects to represent the timeflag in a more legible and usable format.

```
print(f"Earliest available TFLAG is {parse_tflag(ds['TFLAG'].values[0][0])}")
print(f"Latest available TFLAG is {parse_tflag(ds['TFLAG'].values[-1][0])}")
```

```
Earliest available TFLAG is 2021-03-04 09:00:00
Latest available TFLAG is 2021-03-06 11:00:00
```

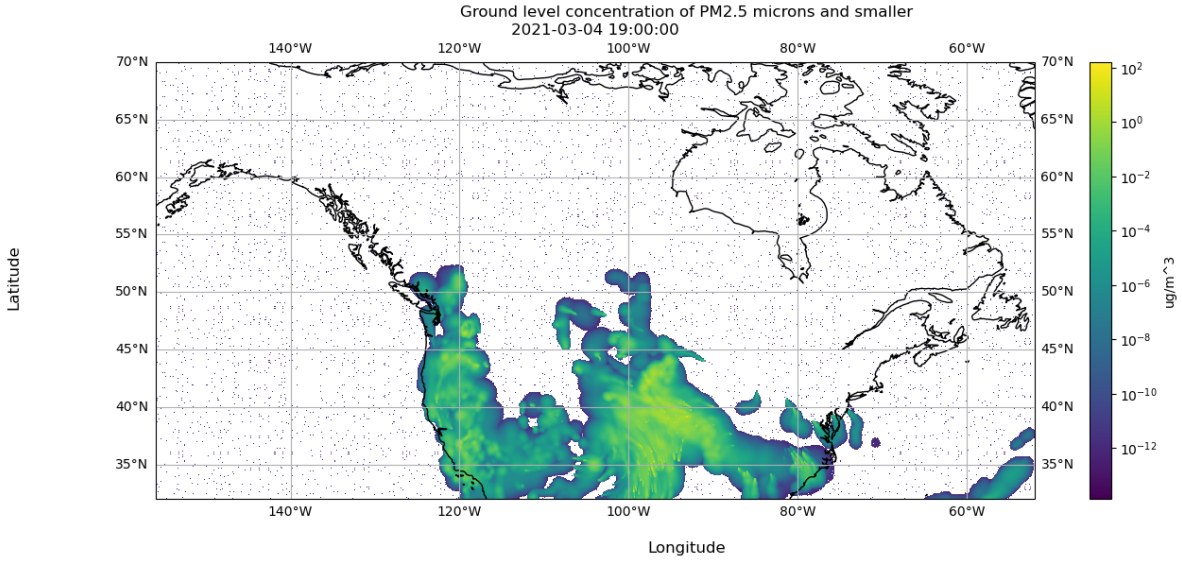
13.0.0.3.3 Visualize Array in `matplotlib`

Let's visualize timestep 10 again, but now we can label the data using latitudes and longitudes, and the corresponding time flag.

```
tstep = 10                                                                    ①
smoke_at_tstep = ds_pm25_vals[tstep, :, :]                                    ②
tstep_tflag = parse_tflag(ds['TFLAG'].values[tstep][0])                       ③

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

my_fig, my_plt = plt.subplots(figsize=(15, 6), subplot_kw=dict(projection=ccrs.PlateCarree())

my_norm = "log"                                                               ⑤
my_extent = [np.min(longitude), np.max(longitude), np.min(latitude), np.max(latitude)] ⑥
my_aspect = 'auto'                                                            ⑦
my_origin = 'lower'                                                           ⑧
my_cmap = 'viridis'                                                           ⑨

plot = my_plt.imshow(smoke_at_tstep, norm=my_norm, extent=my_extent,
          aspect=my_aspect, origin=my_origin, cmap=my_cmap)                   ⑩

my_plt.coastlines()                                                           ⑪

my_plt.gridlines(draw_labels=True)                                            ⑫

my_fig.colorbar(plot, location='right', label='ug/m^3')                       ⑬

my_fig.supxlabel('Longitude')                                                 ⑭
my_fig.supylabel('Latitude')                                                  ⑮

my_fig.suptitle('Ground level concentration of PM2.5 microns and smaller')    ⑯

my_plt.set_title(f'{tstep_tflag}')                                            ⑰

plt.show()                                                                    ⑱
```

① Define the time step.
② Extract the PM2.5 data for the specified time step.
③ Parse the time flag for the specified time step.
④ Initialize a figure and plot with a specific projection.
⑤ Set the normalization for PM2.5 values to a logarithmic scale.

⑥ Define the extent of the plot based on the longitude and latitude range.
⑦ Set the aspect ratio of the plot to fit all data automatically.
⑧ Specify the origin of the plot as the lower-left corner.
⑨ Choose a colormap for the plot.
⑩ Create the plot using `imshow` with the specified parameters.
⑪ Draw coastlines on the plot.
⑫ Draw latitude and longitude lines with labels.
⑬ Add a colorbar to the figure based on the plot.
⑭ Set the x-axis label.
⑮ Set the y-axis label.
⑯ Set the title of the figure.
⑰ Set the title of the plot as the timestamp of the data.
⑱ Display the resulting visualization.




```
ds.close()
```

Now that we understand how to load the data and metadata from the file and process it for visualization, let's establish the data and metadata available to us across *all* NetCDF files.

References

BlueSky Canada. 2021. "FireSmoke Canada." https://firesmoke.ca/.

McKENZIE, Donald, Ze'ev Gedalof, David L. Peterson, and Philip Mote. 2004. "Climatic Change, Wildfire, and Conservation." *Conservation Biology* 18 (4): 890–902. https://doi.org/10.1111/j.1523-1739.2004.00492.x.