A TECHNICAL NOTE ON THE ANALYSIS OF MODERN METEOROLOGICAL GRIDDED DATASETS USING PYTHON

By

Lekshmi S

Dr. Rajib Chattopadhyay

Dr. Pulak Guhathakurta

Dr. D. S. Pai

India Meteorological Department, Pune, India

OFFICE OF THE HEAD, CLIMATE RESEARCH & SERVICES

IMD, PUNE- 411005

Corresponding Author:
Dr. Rajib Chattopadhyay
Email: caui.imdpune@gmail.com

Table of Contents

ABSTRACT

Big data analytics using python as a programming language has gained popularity in several fields. The modern meteorological data is structured as well as has a complex multidimensional structure involving several geometrical, spatial and time coordinates. There is a need to seamlessly integrate as well read and write climate data with other applications, e.g. machine learning models which are popularly written using a pythonic interface. The technical report provides a conceptual framework in using python language for climate data analytics using India Meteorological Department's (IMD's) gridded data and other modern reanalysis data. The report provides several codes written using python which can be used for meteorological data analysis. Also, as a case study, an analysis is done to explore the intraseasonal variability in the daily maximum and minimum temperature data from IMD during different season over the monsoon zone. All the codes are available in a GitHub repository *(https://github.com/CRS-IMDPune/Python-Demo-Scripts).*

1. INTRODUCTION AND PURPOSE

Operational climate services use climate data from different sources which now a days falls under the domain of big data analytics. Data from various sources e.g. surface observation, remote sensed observation, model generated data, reanalysis data are commonly used by operational forecasters and researchers all over the world which are generated every day. The data also comes in several formats. Some data like gridded binary (or GRIB), network common dataset (or NetCDF) is popular and is a standard dataset now a days. These formats are routinely used by several operational centers to store their data which can be downloaded through internet. In the present scenario of changing climate and increased frequency of extreme events, climate and weather data analytics have gained great importance. Historical data analysis and forecasts using different dynamical and statistical models have increased among the scientific community. Huge amounts of data are being collected and created all over the world which are left for analysis and for the development of theoretical and applicational frameworks. The information about the atmospheric conditions spatially, temporally and their variation at different vertical levels are to be managed together. Thus, the efficient processing of these multidimensional datasets is one of the major challenges faced by the climate research community.

With the availability of vast amounts of data, the demand for techniques and tools for analyzing them also increased in the recent times. The requirement of high-performance computing, image analysis software, programming languages for analysis of data are few among the lists. Multiple formats of data and a variety of tools for their analysis are available now-a-days which are getting updated day-by-day. Although several “traditional” programming language are used frequently for big-data analytics e.g. FORTRAN, MATLAB or C++, big data analytics using Python programming language has gained popularity in the recent decades. The open-source availability of software and advanced scientific programming concept and a large community support provides additional advantage in using it. In the meteorological data analysis domain several pythonic tools are available now a days. In this report we document a systematic usage of Python programming language as a tool for analyzing the meteorological gridded datasets. This report describes the major datasets available for the weather and climate research & the modules and packages in Python that are commonly used for analyzing them. It also includes few programing methods that can be used for analyzing data. While a number of ways are possible to do the same job, the applications discussed here are adopted to suit the operational forecasters in

the India Meteorological Department. We have also included a scientific case study conducted by making use of the tools and packages in Python to provide basic understanding of the packages. The case study would focus on capturing the intraseasonal variability in the gridded IMD temperature data. All the codes written here are available in a GitHub Repository: *https://github.com/CRS-IMDPune/Python-Demo-Scripts*

2. METEOROLOGICAL GRIDDED DATASETS

Datasets which are commonly used for climate research include: (a) netCDF (b) GRIB and (c) BINARY. NetCDF and GRIB data formats are self-describing. That is, even though the users are unaware of the structural form of these datasets, they can be read and examined with the use of suitable software. There is some additional information about the data available with the datasets which are called their *'metadata'*. Metadata is defined as the data that provides information about one or more aspects of the data which includes all the attributes and coordinate information of all variables. Some of the typical metadata includes information about the source of the data, information about each variable such as their description, name, units, missing values etc. All the forms of data have evolved over time so as to support the needs of the respective user communities.

2.1. NETCDF

Network Common Data Form[9] is a portable, self-describing format in which the header describes the layout of the file which includes the structure of the data and the metadata associated with it. Usually, the files are written following a certain standard NetCDF conventions. The most commonly used NetCDF conventions are COARDS and CF conventions. NetCDF is the format most commonly used by the climate model generated data. A typical NetCDF file can be 'dumped' to obtain the information such as the dimension names, dimension sizes, variables along with their attributes, spatial and temporal extent, coordinate variables (one dimensional variable same as the dimensions) and global attributes. The dimension size 'Unlimited' is used when a variable can grow to any length along that dimension. Some operations require an unlimited dimension such as combining multiple files along a dimension (Ex: Time).

There are two versions of NetCDF data (3 and 4). NetCDF3 was used for a long time and as the grids became complicated and the users demanded for more flexibility, NetCDF4 was developed.

2.2. GRIB

GRIB[8,13] format is used by all the world's operational weather centers and is designed and maintained by the World Meteorological Organization (WMO). It is a file format used for the storage and transport of gridded meteorological data. It is designed in such a way that it is compact and is an efficient vehicle for transmitting large volumes of data over high-speed telecommunication lines. GRIB format may or may not be considered as 'self-describing'. All the information which is necessary to unpack the data are contained within each record. But the variable which is unpacked is denoted by an 'indicator parameter' and the variable's name and units are to be obtained separately from a grib table.

A GRIB file contains one or more data records which are arranged in a sequential bit stream. Each record has a header which is followed by a packed binary data. The header contains information about the qualitative nature of the data (field, level, date of production, forecast valid time, etc.), information about the header itself, methods and parameters to be used to decode the data and the layout and geographical characteristics of the grid of the data.

WMO has issued three editions of GRIB, Edition 0, 1 and 2. GRIB 0 is now obsolete, unsupported and is rarely used. GRIB 1 is still used but its future enhancements are frozen. Each GRIB1 record contains information for two horizontal dimensions, such as latitude and longitude, for one time and one level. But GRIB 2 allows each record to store multiple horizontal grids and levels for each time. A collection of GRIB records is called a GRIB file. Generally operational weather centers create reanalysis and forecast products in this format.

2.3 BINARY

Binary[7,12] files used in the climate data analysis are generally created from compiled languages such as Fortran, C/C++. They are supported by the compiled languages and are easy to create. For reading a binary data file, the user should know the following details beforehand:

(a) The structure of the data.

(b) The data type of all the variables in the file (such as integer, float, double etc.)

(c) The 'endian' type of the file being read ('little' endian or 'big' endian)

Without knowing this information, it is difficult to read the contents in binary files and therefore there are no generic file tools to deal with binary files. This is why binary files are not very commonly used for archiving climate data.

In GrADS (Grid Analysis and Display System), the binary data and the metadata are stored in separate files. The metadata file is called the *data descriptor file*, with the complete description as well as instructions to read the data. The extension of the descriptor file is '*.ctl*' and so it is also referred to as *'control file'*. The other file contains entirely of the data with no space or time identifiers.

3. TOOLS FOR HANDLING METEOROLOGICAL DATA

A computer program is a set of instructions, which performs a specific task when executed by a computer. Computer, being an electronic machine, can understand any instruction written in binary form (0s and 1s) which is called '*Machine Language*'. A computer can easily understand this language, but it is very difficult for humans to write an instruction in it. Thus, another convenient language was developed which is the assembly language. In this machine operations were represented using mnemonic codes (Ex: ADD, MUL) and symbolic names for memory addresses. To translate an assembly language to machine language a program called *assembler* was needed. Both these languages form the low-level languages. High level languages are much easier to write as these are similar to instructions written in the English language. Generally high-level languages are platform independent which makes it easy to run in different machines. The program written in high level language is called *source code*. An *interpreter* (line by line translation) or *compiler* (whole source code is translated) translates the source code into *object code* which is understandable by the machine. A *linker* will combine the object code and necessary libraries and modules into machine language and the code is then executed. There are a lot of high-level languages available now and choosing one language from them is based on the purpose it is expected to fulfill.

While considering the meteorological gridded datasets, reading and analyzing them can be quite problematic because of their multidimensionality. Also dealing with the frequent missing values in the grid points and their enormous size makes them often difficult to handle with. So, the need for efficient software, tools and techniques are of high priority for weather and climate related studies. Some common programming languages that are used include C, C++ and FORTRAN

which are used in developing climate models. For the analysis of the observational and model generated meteorological gridded datasets the languages which are used commonly now-a-days includes Grid Analysis and Display System (GrADS), Climate Data Operator (CDO), NCAR Command Language (NCL), FERRET, MATLAB, R and Python.

4. PYTHON- AS A TOOL FOR HANDLING METEOROLOGICAL DATA

Python was developed by Guido van Rossum[11] at the National Research Institute for Mathematics and Computer Science in Netherlands in 1990. Python became a popular programming language, widely used in both industry and academia because of its simple, concise and extensive support of libraries. Python code is available under General Public License (GPL) and maintained by a core development team at the same institute. There are some very well-known advantages of Python that make it a popular programming language. Python is an interpreted language which can be treated in procedural, object-oriented or functional ways. It is capable of cross-platform applications, easily readable with syntax similar to English language, supported by a vast collection of libraries, can be easily integrated with other programming languages and written with minimum lines of code. In addition to these, Python is a free and open-source software which makes it easy to use, modify and redistribute. Because of these, Python has a very wide application in different fields such as business, scientific and numerical, creation of web applications, software development, image processing and many others.

Exploratory data analysis is an approach to analyze data, to summarize the main characteristics of data, and better understand the data set. It also allows us to quickly interpret the data and adjust different variables to see their effect. Accessing, reading and analyzing meteorological datasets forms an instance of the exploratory data analysis. Python has been used more frequently and is more popular among the weather and climate research community for a few years. As the usage of Python in climate data analysis has increased, lots of libraries, packages and modules have been introduced which support the access and analysis of meteorological datasets efficiently. Since it is a free and open source software, frequent updates according to the rising demands in the user community are available for all sorts of Python applications.

5. PYTHON MODULES AND PACKAGES FOR DATA ANALYSIS

A lot of modules and packages are available in Python which gives extensive support in handling the data and executing the operations. A *module* is a Python file that is to be imported into the Python scripts or other modules. It often defines members such as classes, functions and variables that can be used in the files where it is imported. At the same time, a *package* is a collection of related modules that work together to provide certain functionality. All the related modules are placed within a folder (sometimes nested within subfolders) along with an *__init__* file inside the folder. This *__init__* file informs Python that it is a package and it can be imported as any other module. A *library* is an umbrella term that loosely means "*a bundle of code*" which may contain tens or even hundreds of individual modules. They can provide a wide range of functionality and *Matplotlib* is one such example of a plotting library. The *Python Standard Library* contains a wide range of built-in modules (written in C) which can be used to deal with everyday programming. These are included in the standard version of Python, so there is no need for any additional installations. For handling the meteorological datasets, few packages in Python are most necessary which includes *numpy, pyngl, pynio, pandas xarray, cfgrib, datetime, cartopy, basemap, pandas, geopandas, regionmask* etc. Each of these packages are important in multiple ways such as for doing mathematical operations, reading different formats of data, plotting, for masking specific regions and so on. A brief note on these packages are as follows:

5.1 NUMPY

NumPy[4,14] is the fundamental package used for scientific computing in Python. This Python library provides a multidimensional array object and various derived objects (such as masked arrays and matrices). It also provides an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. *NumPy* stands for numerical python. In Python there are *lists* which serve the purpose of arrays, but they are slow to process. *NumPy* aims to provide an array object that is much faster than Python lists. The array object in *NumPy* is called *ndarray* that encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. *NumPy* is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

5.2 XARRAY and CFGRIB

The unlabeled, n-dimensional arrays of numbers (*NumPy ndarray*) are the most widely used data structure for scientific computing. But the metadata associated with the data cannot be meaningfully represented. It depends on the functionality of individual users and domain-specific packages. *Xarray*[5,21] expands the capabilities of the *NumPy ndarrays* by providing a lot of streamline data manipulation. *Xarray* includes functions for advanced analytics and visualization which is hugely inspired from the Python library *'Pandas'* and uses it internally. *Pandas* can work well with tabular data but *xarray* is more suitable for data with higher dimensions. *Xarray* makes the handling of n-dimensional arrays easier in many ways such as:

- While *Numpy* uses axis labels, *xarray* uses named dimensions which makes it easy to select data and apply operations over dimensions.
- *Xarray* can hold heterogeneous data in a ndarray while *NumPy ndarray* can handle only one data type.
- *Xarray* makes Nan handling easier in Python
- *Xarray* keeps track of the arbitrary metadata on the object.

Xarray has two data structures:

- *DataArray* — for a single data variable
- *Dataset* — a container for multiple DataArrays (data variables)

Cfgrib is a Python interference to map GRIB files to the NetCDF Common Data Model following the CF Convention using ecCodes. It is designed to support a grib engine to *xarray* and enables the *engine='cfgrib'* option to read GRIB files with *xarray*. It reads most GRIB 1 and 2 files including heterogeneous ones.

5.3 PANDAS

Pandas[6,20] is a Python package providing fast, flexible and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It is one of the most powerful and flexible open source data analysis/manipulation tools available in any language. *Pandas* is well suited for different types of datasets such as tabular data with heterogeneously-typed columns, ordered and unordered time series data, arbitrary matrix data and any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a *pandas* data structure. *Pandas* is built on top of *NumPy* and is designed in such a way

as to integrate well within a scientific computing environment with many other third-party libraries. The two primary data structures of *pandas* are *Series* (one dimensional) and *DataFrame* (two dimensional).

5.4 PyNGL & PyNIO

PyNGL[15] is a Python module built on top of NCAR Command Language's (NCL) graphics library, which produces publication-quality, two-dimensional visualizations especially for climate and weather data. There are interfaces for the display of contour plots, XY plots, map projections, vector and streamline plots.

PyNIO is a Python module that allows read and/or write access to a variety of scientific data formats popular in climate and weather such as NetCDF3 / NetCDF4, GRIB1 / GRIB2, HDF4, HDF-EOS2, HDF-EOS5 and Shapefile. *PyNIO* has a data model interface that is modeled on the NetCDF data model.

5.5 DATETIME

In Python, for dealing with date as well as time a module named '*datetime*' has to be imported because they are not a data type of their own in Python. It is an inbuilt module that comes with Python. This module supplies classes to manipulate date and time. They provide a variety of functions that can be performed on dates, times and time intervals.

5.6 MATPLOTLIB & BASEMAP

Matplotlib[2,18] is a comprehensive library for creating static, animated and interactive visualizations in Python. It can create publication quality plots, make interactive figures that can zoom, pan and update, customize visual style and layout, export to multiple file formats and use a rich array of third-party packages built on *Matplotlib*.

The *matplotlib basemap* toolkit is a library for plotting 2D data on maps in Python. *Basemap* does not do any plotting on its own. It is an extension of *matplotlib* which provides the facilities to transform coordinates to one of the 25 different map projections. *Matplotlib* is then used to plot contours, images, vectors, lines or points in the transformed coordinates. *Basemap* can add features such as shoreline, river, political boundary etc. to the desired map projection region. It was mainly developed to address the needs of earth scientists, particularly oceanographers and meteorologists, but it has a wide application in other disciplines also. It is to be noted that this

package is being deprecated in favor of the new package '*cartopy*'. It is highly recommended for the users to use *cartopy* rather than using *basemap*.

5.7 CARTOPY

Cartopy[16] is a Python package designed for geospatial data processing in order to produce maps and other geospatial data analyses. The major features of *cartopy* are its object-oriented projection definitions, its ability to transform points, lines, vectors, polygons and images between those projections. This package makes use of *PROJ, NumPy* and *Shapely* libraries and has a programmatic interface built on top of the *Matplotlib* library for the creation of maps having publication quality. Basically, *cartopy* is a cartographic Python library with *Matplotlib* support.

5.8 GEOPANDAS

GeoPandas[3,19] project is an extension of the *pandas* library to add support for geospatial data to *pandas* objects. The core data structures in *GeoPandas* are the *geopandas.GeoDataFrame* and *geopandas.GeoSeries* which are the respective subclasses of *Pandas DataFrame and Series*. The *DataFrame* can store geometry columns and perform spatial operations whereas *Series* handles the geometry. That is, the *geopandas.DataFrame* is a combination of *pandas.Series* (numerical, boolean, text etc.) and *geopandas.GeoSeries* (points, polygons etc.). *GeoPandas* objects can act on *shapely* geometry objects and perform geometric operations. *GeoPandas* further depends on *fiona* for file access and *matplotlib* for plotting.

5.9 REGIONMASK

Regionmask[17] is a Python module which contains a number of predefined regions such as countries, a landmask and regions used in scientific literatures. It can plot the figures of these regions by using *matplotlib* and *cartopy*. This module can be used for creating user-defined masks of regions for arbitrary longitude and latitude grids (2D integer mask and 3D boolean mask). This module can use *geopandas* to read shapefiles and can create user-defined masks over regions required.

6. GETTING HOLD OF PYTHON PACKAGES

The modules that are described in the previous sections form only a small fraction of the available packages and libraries supported by Python. Python has a very rich standard library which can be used immediately on installing Python. Different repositories are there from which

one can access multiple packages for Python. Among them the *Python Package Index* (PyPI, *https://pypi.org/)* is the largest Python repository which contains many packages developed and maintained by the Python community.

There are multiple ways to install Python. Downloading the official Python distributions from *Python.org* and by using package manager such as *Anaconda (https://www.anaconda.com/)* are two of the most common methods. *Anaconda* is a package manager, an environment manager, a Python and R data science distribution and a collection of over 7500 open source packages. It is free to install and offers free community support as well. The environment management system in *Anaconda* named '*Conda*' makes it easy to install/update packages and create /load environments. *Conda* is a cross platform package and environment manager which installs and manages conda packages from Anaconda repository as well as from Anaconda Cloud. *Conda* packages are binaries and there is no need to have compilers available to install them. Over 250 packages are automatically downloaded with *Anaconda* and over 7500 additional open-source packages can be installed individually according to the need. One can install additional packages by using the command '*conda install package_name*' which will download and install the package and its dependencies by default.

Once Python is installed the packages can be installed either using *conda* or using *pip* which are nearly identical. *Pip* is a package installer for Python and it can install packages from PyPI as well as other indexes. *Pip* is the Python Packaging Authority's recommended tool for installing packages from PyPI. *Pip* installs Python packages whereas conda installs packages which may contain software written in any language. Before invoking *pip*, one must make sure that they have installed a Python interpreter. At the same time, conda can install Python packages and the Python interpreter directly. Using *conda* it is also possible to create isolated environments with different Python versions and packages rather than installing all packages to the same environment. In *pip* there is no built-in support for environments, but have to depend on other tools for creating environments. While installing packages using *pip* the dependencies of packages are not installed simultaneously which may lead to the creation of broken environments. In contrast *conda* verifies that all requirements for installing the package are met in the environment. This may cause some extra time, but prevents the creation of broken environments. Despite the large collection of packages available in Anaconda repository and Anaconda Cloud, it is still small compared to over 150,000 packages available in PyPI. Occasionally the package which is not

available in *conda* and available in PyPI can be installed using *pip*. So, it makes sense that one can use *conda* and *pip* together when needed.

7. ANALYSIS OF METEOROLOGICAL GRIDDED DATASETS USING PYTHON

In the previous sections we have discussed the usage of Python tools for data analysis and the packages available for that. Here we have included some sample Python codes which use the above packages to read, write and analyze meteorological gridded datasets. The detailed codes and sample dataset are available in the GitHub repository: *https://github.com/CRS-IMDPune/Python-Demo-Scripts*. In every data analysis, calling the data and reading the content in it forms the most important part. Below given are the parts of Python scripts which will help us read the datasets of the formats discussed above.

NetCDF4

```
##########Import necessary modules#################
import netCDF4 as nc
############ File to be read ####################
file_name=
"/mnt/e/Python_Scripts/Sample_Data/RFone_imd_rf_1x1_2019.nc"
################## open file ######################
f = nc.Dataset(file_name)
print(f)                        # gives us information about the variables
                                #contained in the file and their dimensions
for dim in f.dimensions.values():
    print(dim)                  # Metadata for all dimensions
for var in f.variables.values():
    print(var)                  # Metadata for all variables
print(f['rf'])                  # Metadata of single variable
################## read variables  ################
rf   = f.variables['rf'][:]
lats = f.variables['lat'][:]
lons = f.variables['lon'][:]
time = f.variables['time'][:]
print(rf.min()," ,",rf.max())
```

For reading multiple netCDF files, use the function netCDF4.MFDataset(). By using the function, the files will be concatenated in the unlimited dimension in the file (usually 'time' dimension). One can reshape the files according to the need of the analysis.

```
f = nc.MFDataset(file_name)     # Read all data files together
```

For example, if you want to reshape an hourly data for a year of dimension 8760 x 129 x 129 (time, latitude and longitude) into 365 x 24 x 129 x 129, then the following method can be used.

```
RF=np.reshape(rf,(ndays,ntime,len(lats),len(lons)))
print(RF.shape)       # print shape of the variable
```

GRIB

Below given are the methods for reading data from grib files.

```
import xarray as xr
ds=xr.open_dataset('/mnt/e/Python_Scripts/Sample_Data/ERA5_Temperature_
2020.grib',engine='cfgrib',backend_kwargs={'indexpath':''})
                                                    ##For single file

ds=xr.open_mfdataset('/mnt/e/Python_Scripts/Sample_Data/ERA5_Temperatur
e_*.grib',engine='cfgrib',backend_kwargs={'indexpath':''})
                                                    ## For multiple files
print(ds)                          ###Print summary of the file
print(ds.t)                        ###Print summary of the variable
print(ds.latitude)
print(ds.longitude)
print(ds.isobaricInhPa)
temp=ds.t                          ## Read temperature in a variable
```

BINARY

The binary data files are not read in the same way as the GRIB or NetCDF4 data. Because as we have already mentioned data structure of binary files should be read from a control file. For other datasets even though we don't know the structure, it is possible to read them and get information from them. So, data in binary files are read as a single dimensional variable and then reshaped into the appropriate dimensions as obtained from the control file.

```
import numpy as np
filename='/mnt/e/Python_Scripts/Sample_Data/Maxtemp_MaxT_2018.GRD'
                                                    ##File path
nlat=31                                        # Obtained from the control file
nlon=31
ntime=365
lons=np.arange(67.5,98.5,1)              # Define latitude and longitude as
                                                    obtained from control file
print(lons)
lats=np.arange(7.5,38.5,1)
print(lats)
f=open(filename,'rb')
data=np.fromfile(f,dtype="float32",count=-1)      #Opening and reading
                                                    the file into a one-dimensional array
#################Reshaping data#######################
temp=np.reshape(data,(365,31,31),order='C')
print(temp.shape)
exit()
```

Sometimes it is not necessary to read and analyse the dataset for the whole spatial and temporal regions. Extracting data for a particular location or region, for a particular vertical level and for a particular period of time are much frequently used in the weather and climate studies. Given below are the methods for subsetting the data for required region, period and level.

```
latbounds = [ -15 , 15 ]      #degrees north
lonbounds = [ 70.5 , 100.5 ]  # degrees east
levbounds = [500.0 , 900.0 ]
```

```
latselect=np.logical_and(lats>=latbounds[0],lats<=latbounds[1])
lonselect=np.logical_and(lons>=lonbounds[0],lons<=lonbounds[1])
levselect=np.logical_and(levs>=levbounds[0],levs<=levbounds[1])
temp_space1=temp[:,levselect,latselect,lonselect]
print(temp_space1)

dt1=datetime.datetime(2020,1,10,15,00,00) ##Give the required start time
dt2=datetime.datetime(2020,2,10,15,00,00) ##Give the required end time
def minutes_of_year(dt):                  ####Defining a function to obtain
      return seconds_of_year(dt)//60      hours_of_year from the
                                           datetime provided

def hours_of_year(dt):
      return minutes_of_year(dt)//60
def seconds_of_year(dt):
      dt0=datetime.datetime(dt.year,1,1,tzinfo=dt.tzinfo)
      delta=dt-dt0
      return int(delta.total_seconds())
ind1=hours_of_year(dt1)                   ###Using the function the hour is
ind1=hours_of_year(dt2)                   obtained  which is the index
temp_time1=temp[ind1:ind2,:,:,:]          #Selecting data for a period
```

The same method can be used to obtain data for single latitude, longitude, level and time step by making minor modifications in the above method. The above example shows how to extract hourly data from a file having hourly data for a single year. According to the time steps in the dataset used, modifications have to be done accordingly. It is also necessary to write the extracted datasets or some calculated parameters in a file which can be used at any later point of time without repeatedly running the codes. One of the most common ways of saving a dataset is to write it in NetCDF format. Given below is a method showing how to write a NetCDF file.

```
import netCDF4 as nc

fn='/mnt/e/Python_Scripts/DEMO/olr_subsets.nc'  #Set path and name of output
                                                 file
```

```
fout=nc.Dataset(fn,'w',format='NETCDF4')          #Open output file
time=fout.createDimension('time',None)            # Set all required  dimensions
                                                  Time-Unlimited
Lats=fout.createDimension('lat',13)
Lons=fout.createDimension('lon',12)
fout.title="Subset of NOAA OLR Data"              #Setting some attributes
fout.subtitle="Lat, Lon and Time subset"

lat=fout.createVariable('lat',np.float32,('lat',))    #Create variables and
                                                      attributes
lat.units='degrees_north'
lat.long_name='latitude'
lon=fout.createVariable('lon',np.float32,('lon',))
lon.units='degrees_east'
lon.long_name='Longitude'
time=fout.createVariable('time',np.float64,('time',))
time.units='hours since 1800-01-01 00:00:0.0'
time.long_name='time'

olr=fout.createVariable('olr',np.float64,('time','lat','lon'))
olr.units='W/m^2'
olr.standard_name='Outgoing Longwave Radiation'
olr.dataset='NOAA Interpolated OLR'
olr.long_name='Daily OLR'

lat[:]=lats[latselect]              # Giving values to the dimensions and variables
lon[:]=lons[lonselect]
time[:]=Time[istart:iend+1]
olr[:,:,:]=olrsub1
```

The sample code written above deals with how to read the datasets, extract the variables according to spatial and temporal requirements and writing data in a file. Apart from these basic necessities, it is very important to visualize and analyze the data. For that, different plotting

techniques are used, in which line graphs, scatter plots, contours, vector plots are of high demand in weather and climate studies. Given below are some of the fundamental, yet most convenient plotting methods to visualize 1-d and 2-d variables.

The sample code here shows how to plot a time series using hourly data and give the time stamps as the tick marks in x-axis. The output obtained using the script is shown in Figure 1.

```
import datetime as dt
import numpy as np
import matplotlib.pyplot as plt
#########Creating date strings for X axis##############
date_string=[]
for n in range(0,u10sub.shape[0],96):
      dates=dt.datetime(2019,6,1,0,0,0)+n*dt.timedelta(hours=1)
      i=int(n/24)
      date_string.append(dates.strftime('%Y%m%d'))
      del dates
x=np.arange(0,u10sub.shape[0],96)
##################Plot time series #################
fig=plt.figure(figsize=(16,7))
plt.plot(u10sub[:,0,0],linewidth=1.0,linestyle='-',color='b')
plt.title(' Time Series of U wind at 10m from 01/6 to 01/8 2019')
plt.ylabel('U wind (m/s)')    ;plt.xlabel('Time')
plt.xticks(x,date_string,rotation=60)
plt.grid(True)

plt.savefig('time_series.png')
```

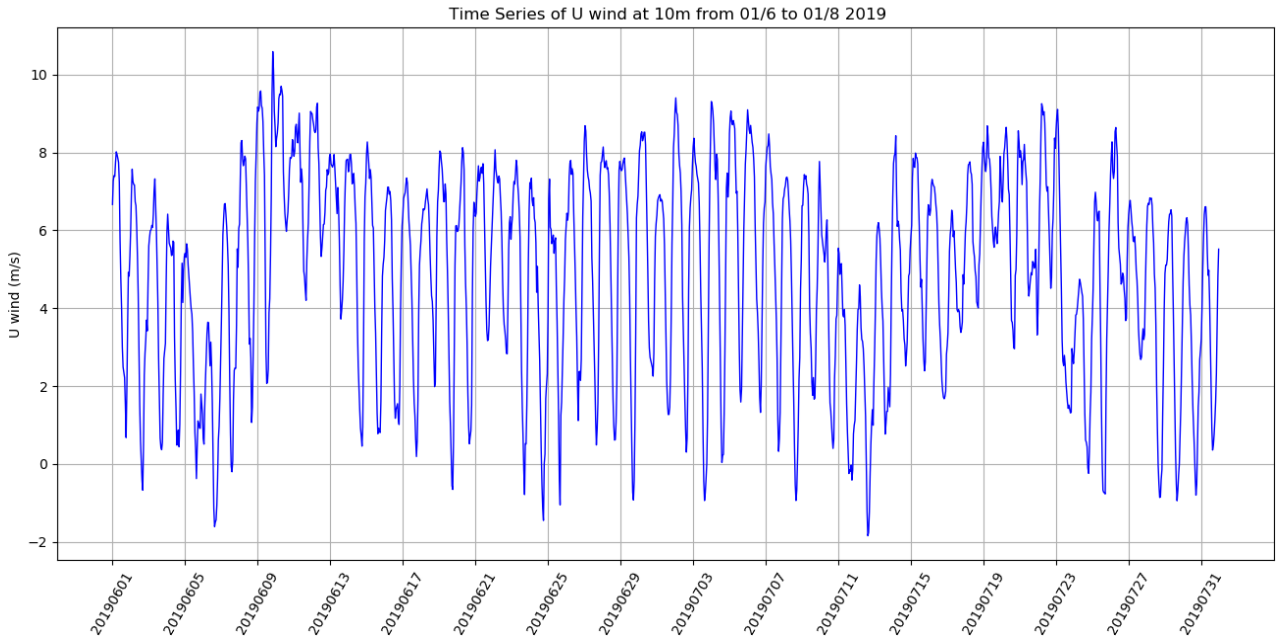

Figure 1: *Time Series of zonal wind at 10m for June and July 2019 from the ERA5 hourly data*

The method shown below shows how to plot a filled contour diagram over the map which helps to visualize the spatial distribution of any variable(2-d) over a region. The output is shown in Figure 2.

```
import cartopy
import cartopy.crs as ccrs
import cartopy.feature as cfeature
from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER
import matplotlib.pyplot as plt

m = plt.axes(projection=ccrs.PlateCarree())
m.set_extent([lonbounds[0], lonbounds[1], latbounds[0], latbounds[1]],
crs=ccrs.PlateCarree())
m.coastlines(resolution='110m')
m.add_feature(cartopy.feature.COASTLINE, edgecolor='black')
```

```
gl=m.gridlines(crs=ccrs.PlateCarree(),draw_labels=True,color='gray',lin
ewidth=1.0)
gl.xlabels_top = False
gl.ylabels_right = False
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER

c=m.contourf(Lons,Lats,U10SUB,transform=ccrs.PlateCarree(),cmap='jet')
plt.title('Wind (m/s)', loc='right')
plt.colorbar(c,shrink=0.75)
plt.savefig('contour_map_cartopy.png')
```

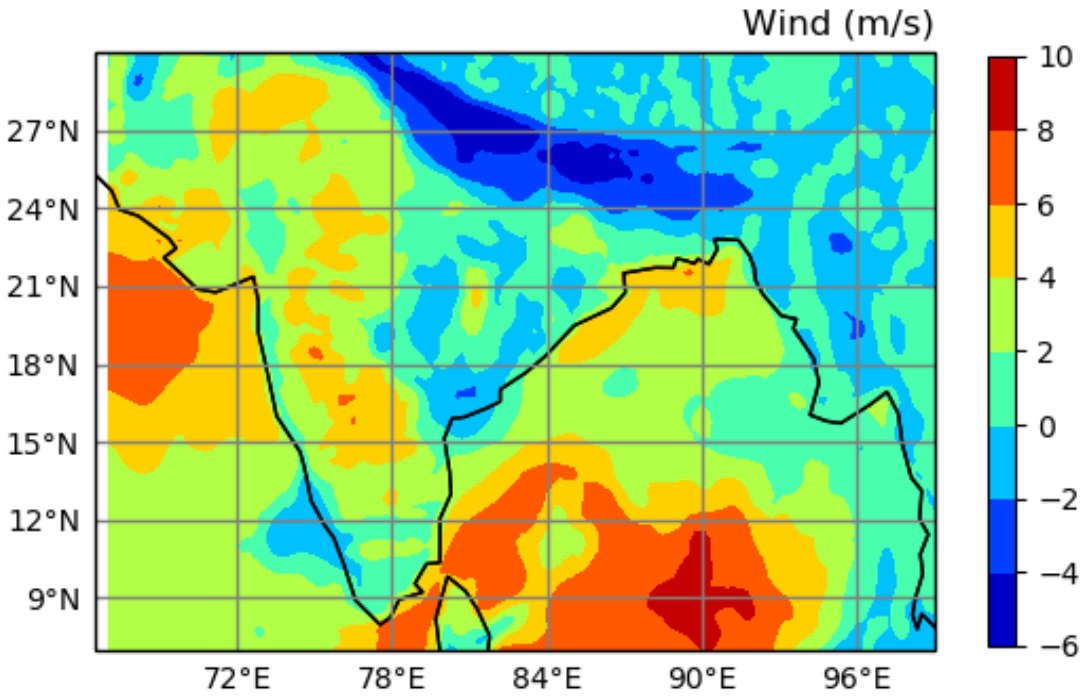

Figure 2: *Filled contour plot of zonal wind at 10m on 1st June 2019 from the ERA5 hourly data*

The method above uses cartopy for including maps. It is also possible to use the *basemap* package for the same purpose. But, as mentioned earlier, *basemap* is being deprecated and it is highly recommended to use *cartopy*. For plotting a line contour the *'contourf'* function can be replaced with *'contour'* in the above example. The sample code below shows a method for

overlaying the wind vector over a filled contour for a particular region and for a particular time step (See Figure 3).

```
plt.figure()
m = plt.axes(projection=ccrs.PlateCarree())
m.add_feature(cartopy.feature.BORDERS, edgecolor='black')
m.set_extent([lonbounds[0], lonbounds[1], latbounds[0], latbounds[1]],
crs=ccrs.PlateCarree())
m.coastlines(resolution='110m')

gl=m.gridlines(crs=ccrs.PlateCarree(),draw_labels=True,color='gray',lin
ewidth=1.0)
gl.xlabels_top = False
gl.ylabels_right = False
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER

plt.suptitle('ERA5 Wind Vector over U wind at 10 m (01/06/2019)')
plt.title('Wind (m/s)', loc='left')
plt.xlabel('Lon')
plt.ylabel('Lat')

c= m.contourf(Lons, Lats, U10SUB, transform=ccrs.PlateCarree())

##################Now overlay Plot vector#################################
q=m.quiver(Lons, Lats, U10SUB, V10SUB, width=0.003, scale_units='xy',
scale=5, transform=ccrs.PlateCarree(), regrid_shape=20)
qk=plt.quiverkey (q,0.95, 1.02, 20, '20m/s', labelpos='N')
plt.colorbar(c)
plt.savefig('vector_overlay_cartopy.png')
```

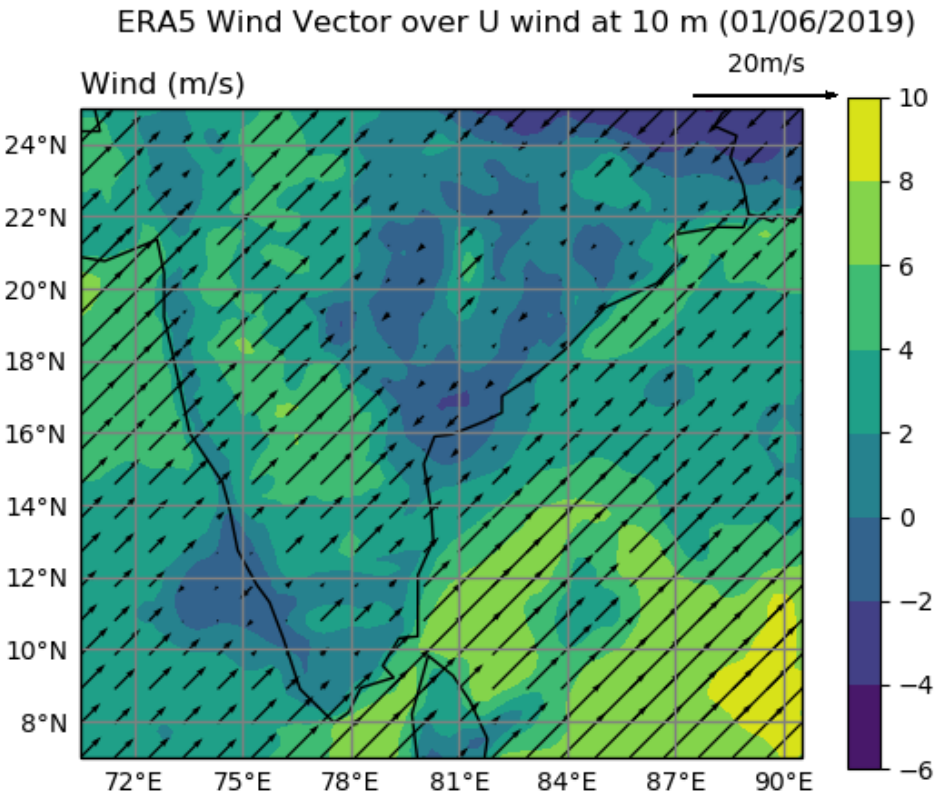

Figure 3: *Wind Vector at 10 m overlaid on the filled contour showing the zonal wind at 10m on 1st June 2019*

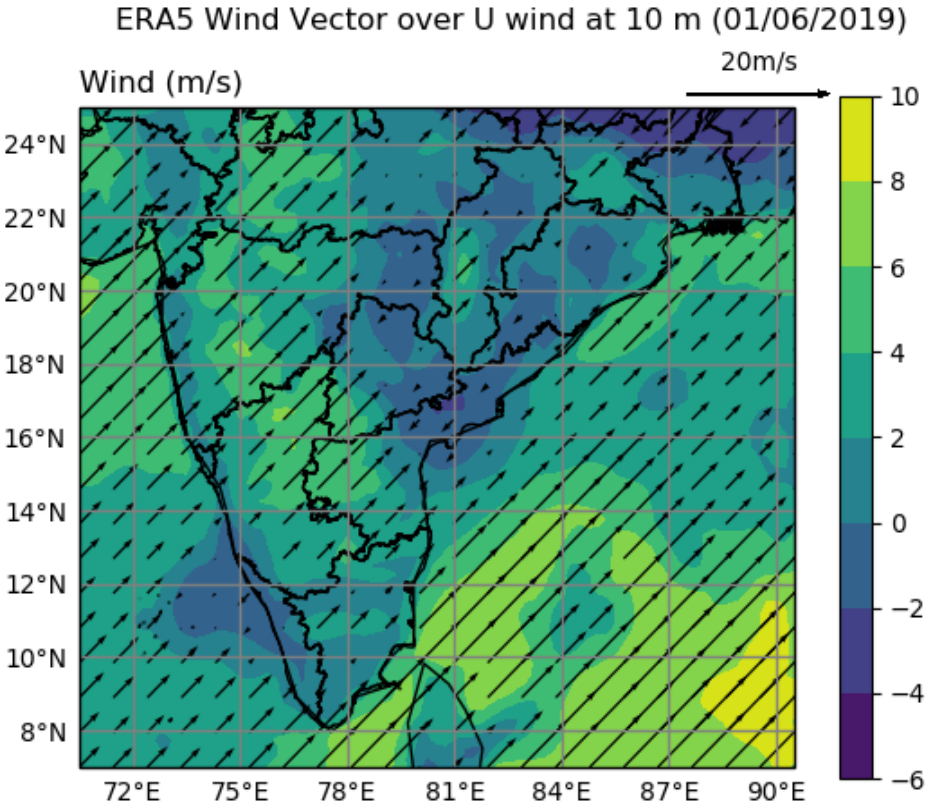

Figure 4: *Wind Vector at 10 m overlaid on the filled contour showing the zonal wind at 10m on 1st June 2019 along with the state boundaries added to it.*

Apart from these common plotting techniques, the use of shapefiles for providing administrative boundaries to the plots are prevalent in operational weather and climate forecast scenarios. The use of shapefile varies from simply overlaying the boundaries in the plots to calculation of zonal statistics for the regions specified using the shapefiles. It is utilized for providing forecasts for administrative regions such as states and districts and also river basins etc. Given below are the sample codes to add a shapefile into the contour plot (Figure 4) and to calculate state average of any variable using shapefile and to plot it (Figure 5).

```
import cartopy
import cartopy.crs as ccrs
import cartopy.feature as cfeature
from cartopy.io.shapereader import Reader
import matplotlib.pyplot as plt
from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER

plt.figure()
m = plt.axes(projection=ccrs.PlateCarree())
m.add_feature(cartopy.feature.BORDERS, edgecolor='black')
m.set_extent([lonbounds[0], lonbounds[1], latbounds[0], latbounds[1]],
crs=ccrs.PlateCarree())
m.coastlines(resolution='110m')

gl=m.gridlines(crs=ccrs.PlateCarree(), draw_labels=True, color='gray',
linewidth=1.0)
gl.xlabels_top = False
gl.ylabels_right = False
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER

plt.suptitle('ERA5 Wind Vector over U wind at 10 m (01/06/2019)')
plt.title('Wind (m/s)', loc='left')
plt.xlabel('Lon')
plt.ylabel('Lat')
```

```
##################Add Shapefile#########################
fname='/mnt/e/Python_DEMO_Scripts/plotting/shpfile/Admin2.shp'
m.add_geometries(Reader(fname).geometries(),ccrs.PlateCarree(),
edgecolor='k',facecolor='none')
c= m.contourf(Lons, Lats, U10SUB, transform=ccrs.PlateCarree())

#################Now overlay Plot vector###################
q=m.quiver(Lons, Lats, U10SUB, V10SUB, width=0.003, scale_units='xy',
scale=5, transform=ccrs.PlateCarree(), regrid_shape=20)
qk=plt.quiverkey (q,0.95, 1.02, 20, '20m/s', labelpos='N')
plt.colorbar(c)
plt.savefig('add_shapefile_cartopy.png')
```

The above example shows how to plot a filled contour plot over a particular region with wind vectors overlaid on it and to add a shapefile into it. The script below shows the entire steps from reading the data, creating a mask using shapefile, calculating state averages using the mask and plotting the averages for each state.

```
import geopandas as gpd
import regionmask
import numpy as np
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import matplotlib.patheffects as pe
import xarray as xr
import netCDF4 as nc

############### Read the data file#############################
file_name ='/mnt/d/DATA/ERA5/Wind/ERA5_Wind_2019.nc'
f = xr.open_dataset(file_name)
u10=f.u10.values
lat=f.latitude.values
```

```
lon=f.longitude.values
lat_size=len(lat)
lon_size=len(lon)

################Read the shape file#########################
fname='/mnt/e/Python_DEMO_Scripts/plotting/shpfile/Admin2.shp'
shp=gpd.read_file(fname)
#print(shp.head())

state_name=list(shp['ST_NM'])
state_1=list(shp['ST_NM'])
indexes=[state_name.index(x) for x in state_1]  # Obtain state indexes

state1=regionmask.Regions(outlines=list(shp.geometry.values[i] for i in
range(0,shp.shape[0])),names=shp.ST_NM[indexes],abbrevs=shp.ST_NM[index
es], name='state',)                              ##Obtaining region boundaries
state1_mask=state1.mask(lon,lat)                 ##### State mask variable

###################Calculating first state average#################
time_ind=10                                      ####Any time step as per need
u10_all=np.full([lat_size,lon_size],np.nan,order='C')        ##Create a
                                  variable to store state average in whole lat-lon range
result=np.where(state1_mask==0)                  ###Obtain indices for first state
lat_ind=result[0]                                # latitude index for state1
lon_ind=result[1]                                # longitude index for state1
u10_state=np.mean((u10[time_ind,:,:][lat_ind,:][:,lon_ind]),axis=(0,1))

#######Calculate state mean (single value)###############
u10_all[result]=u10_state              ##Store state mean in the whole lat-lon range
del u10_state                          # delete variables for future use
del result
del lat_ind
del lon_ind
```

```
#####################Plotting Settings##############################
###############Plotting state 1 average#############################
lev_min=-5                          #Setting contour min,max levels and divisions
lev_max=5
lev_n=30
plt.figure(figsize=(18,8))                          # Plot settings
ax=plt.axes()
x,y=np.meshgrid(lon,lat)
c=ax.contourf(x,y,u10_all,cmap='tab20b',levels=np.linspace(lev_min,lev_
max,lev_n))                                         #Contour plot
shp.plot(ax=ax,alpha=0.8,facecolor='None',lw=1) # Shape file plot

###########Calculating and plotting remaining state average in a loop##############
for i in range(1,shp.shape[0]):              ## Remaining states in a loop
      result=np.where(state1_mask==i)
      lat_ind=result[0]
      lon_ind=result[1]
u10_state=np.mean((u10[time_ind,:,:][lat_ind,:][:,lon_ind]),axis=(0,1))
      u10_all[result]=u10_state
      del u10_state

ax.contourf(x,y,u10_all,cmap='tab20b',alpha=0.8,levels=np.linspace(lev_
min,lev_max,lev_n))
shp.plot(ax=ax, alpha=0.8, facecolor='None, lw=1)

cbar=plt.colorbar(c)                                ## Give colorbar
cbar.set_label('U at 10m', rotation=270)            ##Colorbar label
plt.suptitle('State Average of U Wind at 10m/s at 01/01/2019 09 UTC')
plt.title('Wind (m/s)', loc='left')
plt.xlabel('Lon')
plt.ylabel('Lat')
plt.savefig('zonal_stat_final.png')
```

```
exit()
```

It is to be noted that there are multiple ways to do the same application in Python. These examples are meant to introduce a few methods which can be helpful for performing the most common steps in the weather and climate research. According to the demand of the user these codes can be modified and can be extended to further applications according to his/her expertise.

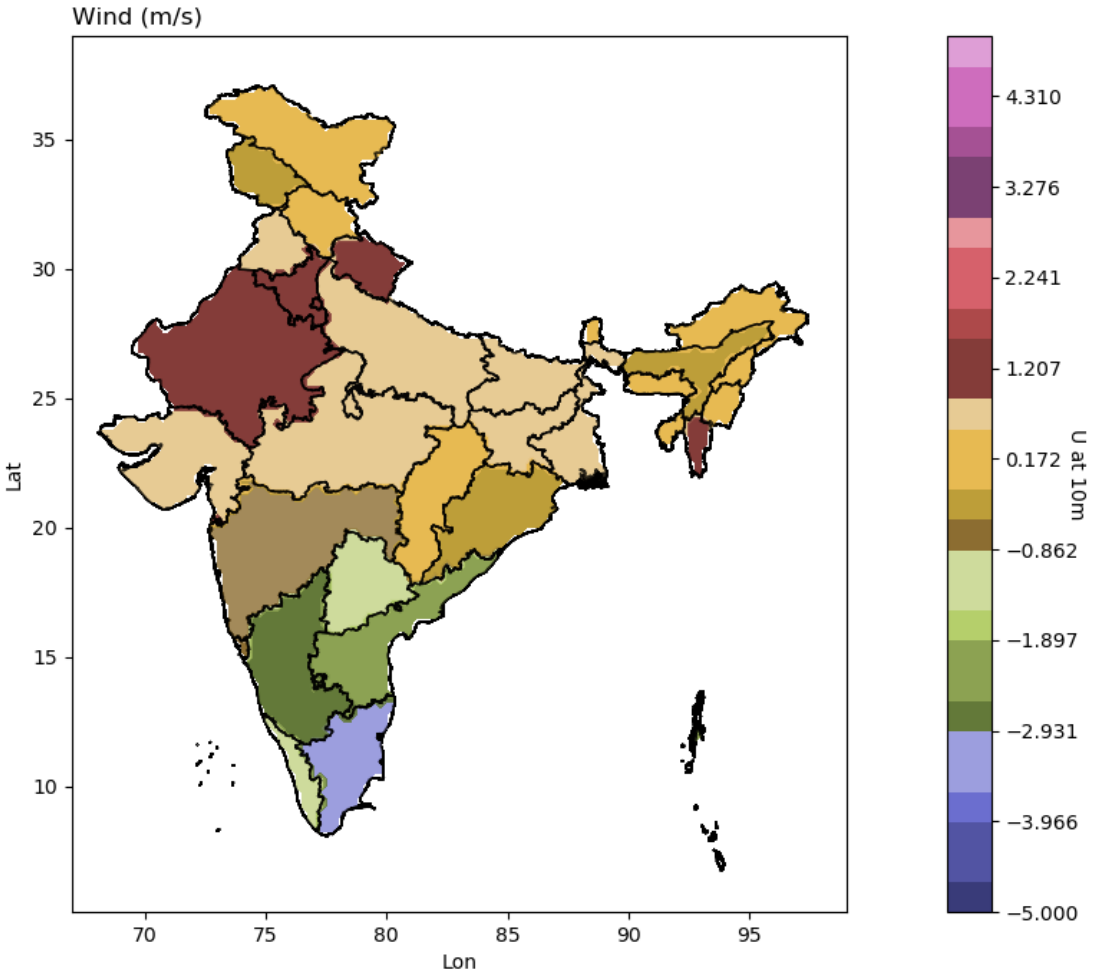

Figure 5: *State average of zonal wind at 10m on June 1, 2019 09 UTC calculated using the shapefile of states*

8. A CASE STUDY

This section demonstrates the usage of Python as a tool for conducting a scientific study in the weather and climate sciences. We have depicted some of the most commonly used plotting and analytical methods in climate science which has been done with the help of Python.

We would like to explore the evidence of *Intraseasonal Variability of Minimum and Maximum Temperature over the Monsoon Zone of India during pre-monsoon, monsoon and winter seasons using the IMD daily gridded temperature data.* Several studies on heatwave and cold wave implicitly assume the intraseasonal variability in the temperature data[1]. In the monsoon season intraseasonal variability is well known in the rainfall data. How do the intraseasonal variability are reflected in a temperature field? For the study we have considered the gridded minimum and maximum temperature data from India Meteorological Department[10] which is in binary format. This dataset is freely available at: *https://imdpune.gov.in/Clim_Pred_LRF_New/Grided_Data_Download.html*. The data is selected for a period of 21 years from 2000 to 2020 which has been used for the study. The region considered for the study is the Monsoon Zone of India (MZI) which extends from 18-29 ^{0}N and 65-89 ^{0}E. The analysis of time series, the presence of intraseasonal oscillations in the minimum and maximum temperature and their relationship with the moisture availability in the atmosphere has been done for the study.

Figure 6 shows the time series of maximum temperature and minimum temperature area averaged over MZI from 2000 to 2020 along with their corresponding mean value shown as a dotted line. The continuous red line shows the maximum temperature pattern during 21 years with a primary peak during the pre-monsoon season of March-April-May (MAM). On the onset of monsoon (June-July) the maximum temperature falls and then a secondary peak occurs during the peak monsoon months of Aug-Sept. Again, it falls during the winter months of Nov-Dec-Jan-Feb (NDJF). While considering the same for minimum temperature it follows the same pattern as that of maximum temperature during the pre-monsoon and winter months. But after the onset of monsoon the fall in minimum temperature is gradual during monsoon and falls steeply during winter months. Apart from the falling trend in minimum temperature there is no secondary peak

in it corresponding to that in the maximum temperature during peak monsoon. That is, during the monsoon a rise in maximum temperature is associated with a gradual fall in minimum temperature.

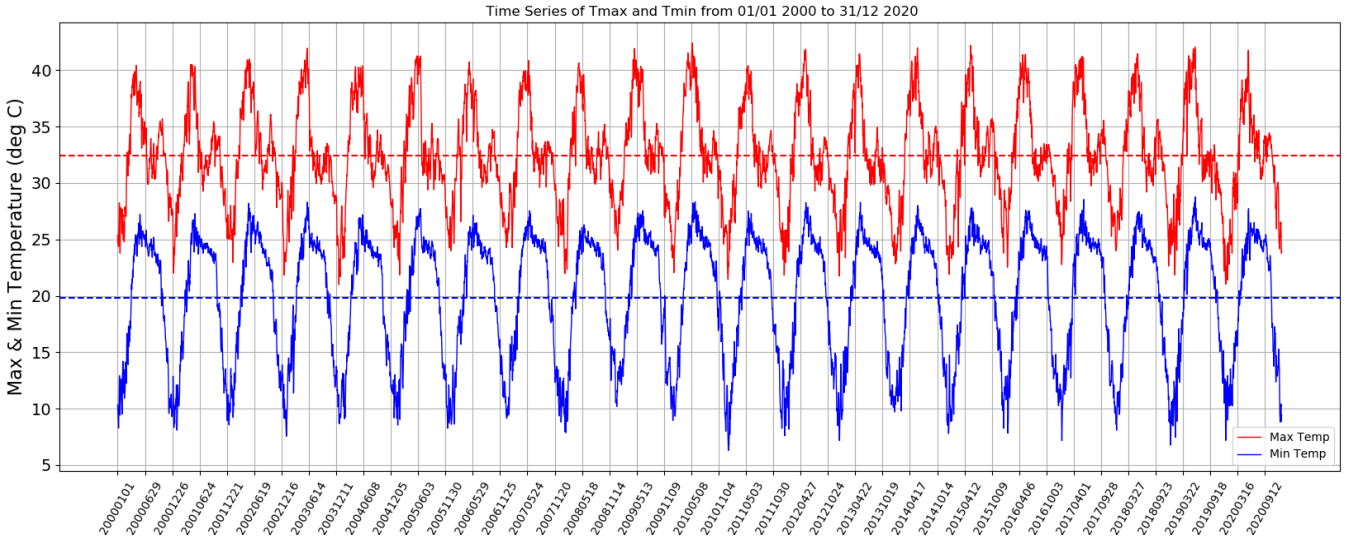

***Figure 6**: Time Series of maximum temperature (red continuous line) and minimum temperature (blue continuous line) in degree Celsius from January 2000 to December 2020. The dotted lines correspond to the mean maximum (red) and minimum (blue) temperature over all these years.*

The power spectrum of a time series describes the distribution of power into frequency components composing that signal. Here we have analyzed the mean power spectrum of minimum and maximum temperature over all the years for each season considered. Figure 7 shows the average power spectrum of minimum and maximum temperature during MAM, JJAS and NDJF over all the years from 2000-2020. The X-axis here represents the logarithmic frequency and the Y-axis here represents the power multiplied with the corresponding frequency. From the figure we can see that there is a strong intraseasonal variation in the evolution of maximum temperature during all the three seasons. The same while considering minimum temperature shows oscillations during MAM and NDJF and not very significant during monsoon.

During the pre-monsoon months (MAM), strong intraseasonal fluctuations with time periods of 20-30, 40-50 and 9-10 days are noticed in maximum temperature. Similar oscillations are present in the spectrum of minimum temperature also but with a lesser magnitude. While considering the monsoon months (JJAS) it is seen that multiple intraseasonal oscillations are seen with periods of 85, 60-70, 25 and 10 days. It is to be noted that even though strong oscillations are there in the maximum temperature, no such signals are obtained in the spectrum of minimum

temperature. In winter (NDJF) also intraseasonal oscillations with periods of 80-90, 60, 30-40 and 10 days are noticeable both in the minimum and maximum temperature with almost the same magnitude. So it is understood that there is strong intraseasonal fluctuations present in the maximum and minimum temperatures during all the seasons (except for minimum temperature in JJAS). It is also concluded that during monsoon the minimum temperature does not follow the oscillations in maximum temperature which is consistent with Figure 6.

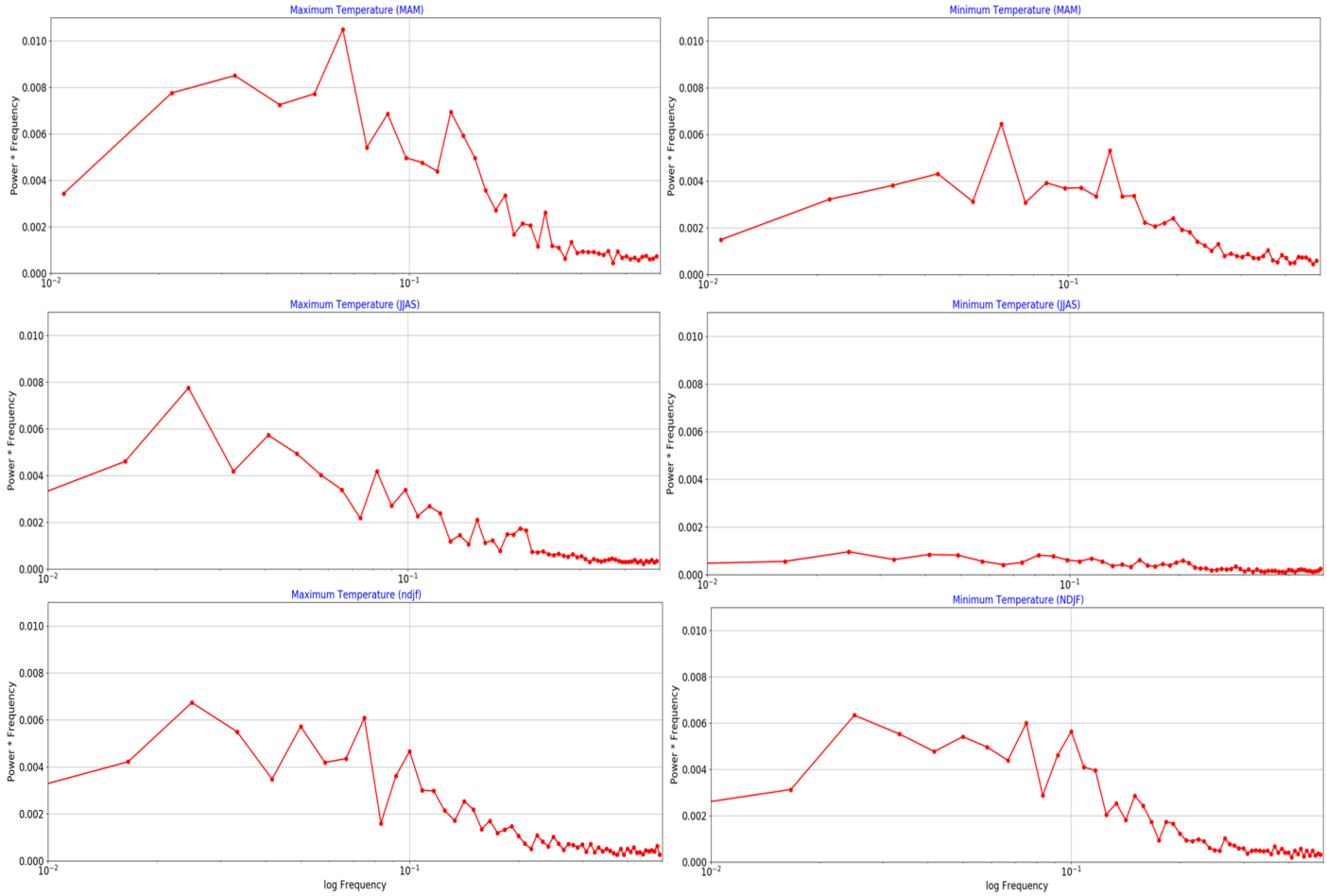

Figure 7: *Average power spectrum of minimum and maximum temperature during MAM, JJAS and NDJF over all the years from 2000-2020 with x-axis showing log frequency and y-axis showing power multiplied with frequency.*

The diurnal temperature range is the difference between the maximum and minimum temperature within a day. This diurnal range in temperature represents the moisture availability in the atmosphere. Figure 8 compares the relation of diurnal temperature range with the maximum and minimum temperature of MZI for the three seasons. From the top panel we can see that there is a strong linear relationship between the diurnal range in temperature to the maximum

temperature during JJAS. That is the moisture availability and the maximum temperature attained during a day have a linear relation during the monsoon season.

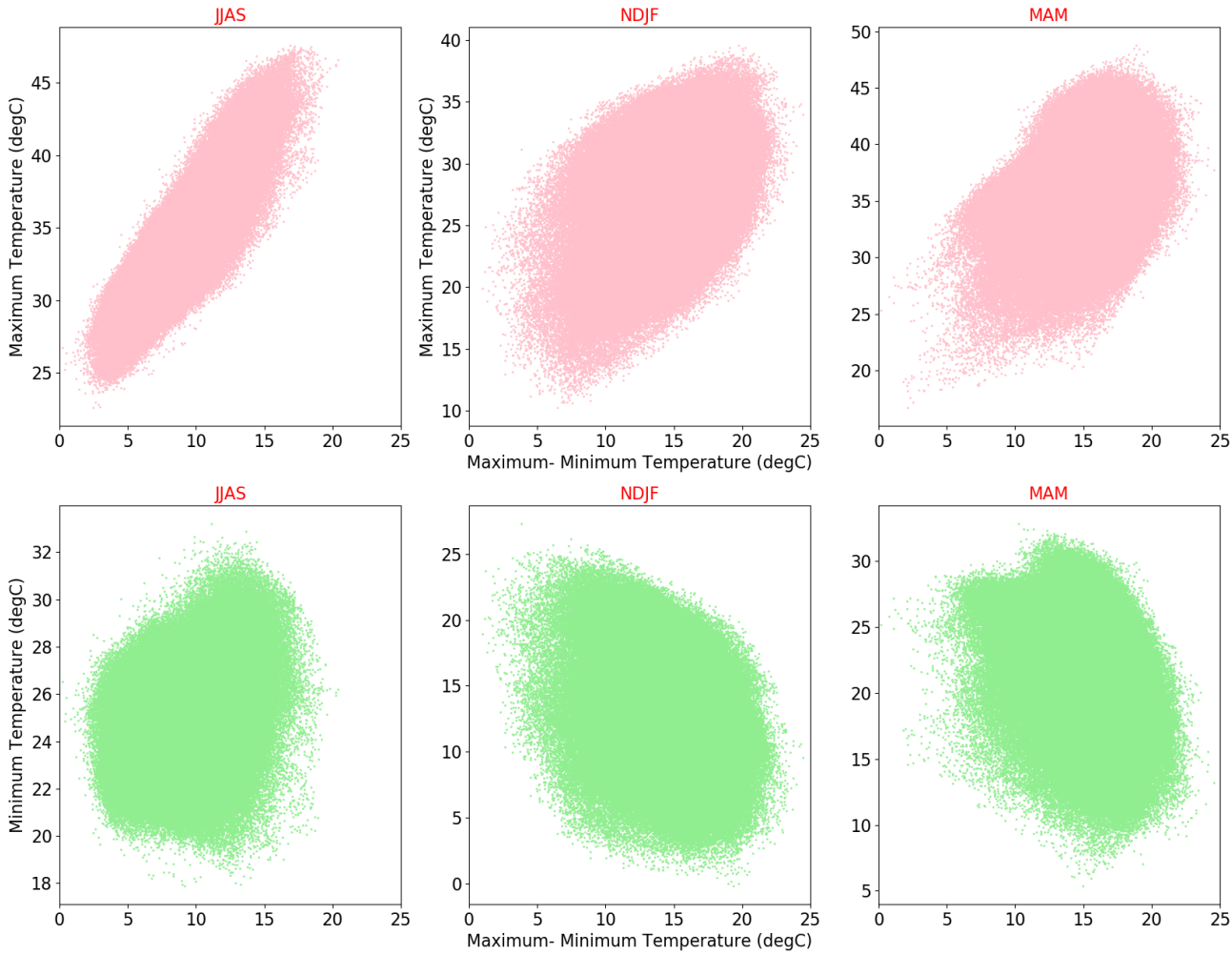

***Figure 8**: Scatter plot of diurnal range of temperature (Maximum-Minimum Temperature) with maximum and minimum temperature during MAM, JJAS and NDJF of 2000-2020 over MZI.*

In the NDJF and MAM seasons the linear relationship between the diurnal range of temperature is not so prominent because of the spread in the distribution. While considering the same in minimum temperature (bottom panel) there is no strong linear relationship between the moisture availability and minimum temperature in all the three seasons. It is to be noted that while there is a positive slope in the distribution of maximum temperature with respect to diurnal range in temperature during all the seasons, there exists a negative slope for the minimum temperature during NDJF and MAM. Thus, from the figure we can say that the relation of moisture availability

in the atmosphere is different with minimum and maximum temperature, considering the fact that both are temperature fields.

From this study we can say that there is a strong intraseasonal oscillation in maximum temperature during all seasons and in minimum temperature during pre-monsoon and winter. It is also found that the moisture present in the atmosphere is having a strong linear relation with maximum temperature during monsoon. During all other seasons they are not linearly related and for minimum temperature no such relation is present during any of the seasons.

9. CONCLUSION

The accelerating changes in data format, presentation and data analytics have created a high demand for the tools and techniques for extracting information from the available data. One of the major areas of research that makes use of huge datasets is weather and climate science. Understanding the state of the atmosphere and its evolution spatially and temporally forms an integral part of predicting the future atmospheric conditions. The availability of efficient analytical tools thus forms an essential part in climate research. A variety of tools and techniques are available now-a-days for this purpose, in which Python has a significant position. This report deals with the usage of Python for accessing and analyzing the major meteorological gridded dataset formats. The characteristics of these gridded datasets and the packages and modules in Python for supporting the basic analysis of these datasets are described here (See Sect. 2-5). Some example Python scripts for reading, sub setting, writing and plotting data are included in Section 7. The same scripts are available in the GitHub repository.

To have some analytical insight using python, a case study is conducted. We analyze the *Intraseasonal Variability of Minimum and Maximum Temperature over the Monsoon Zone of India during pre-monsoon, monsoon and winter seasons* using Python as the tool for analysis. The analysis brings out the intraseasonal fluctuation in maximum and minimum temperature during different seasons. It is shown that, similar to rainfall, the maximum and minimum surface temperature shows intraseasonal variability. At times they are similar but at times they differ from each other. The physical causality of the intraseasonal oscillation in temperature is not explored, although the plots indicate low frequency variability of the intraseasonal oscillations beyond weather scale. It would be worthwhile to explore what is the origin of such low frequency oscillation in the temperature field outside the monsoon zone (such as extratropical Rossby waves).

This report discusses the potential application of Python for the operational research in the climate and weather sciences. There is immense scope for this fast-developing programming language as a tool that can support the rising demands of meteorologists.

REFERENCES

1. D. S. Pai, Smitha Anil Nair and A. N. Ramanathan, 2014, Long term climatology and trends of heat waves over India during the recent 50 years (1961-2010), *Mausam*, p585.
2. J. D. Hunter, "Matplotlib: A 2D Graphics Environment," in Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, May-June 2007, *doi: 10.1109/MCSE.2007.55*.
3. Jordahl, K. (2014). GeoPandas: Python tools for geographic data. *URL: https://Github. Com/Geopandas/Geopandas*.
4. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D. Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*, 357–362. *https://doi.org/10.1038/s41586-020-2649-2*
5. Hoyer, S. and Hamman, J., 2017. Xarray: N-D labeled Arrays and Datasets in Python. *Journal of Open Research Software*, 5(1), p.10. *DOI: http://doi.org/10.5334/jors.148*
6. McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56).
7. National Center for Atmospheric Research Staff (Eds). Last modified 22 Jul 2013. "The Climate Data Guide: Binary." Retrieved from *https://climatedataguide.ucar.edu/climate-data-tools-and-analysis/binary*.
8. National Center for Atmospheric Research Staff (Eds). Last modified 05 Nov 2015. "The Climate Data Guide: GRIB." Retrieved from *https://climatedataguide.ucar.edu/climate-data-tools-and-analysis/grib*.
9. National Center for Atmospheric Research Staff (Eds). Last modified 31 Jul 2014. "The Climate Data Guide: netCDF Overview." Retrieved from *https://climatedataguide.ucar.edu/climate-data-tools-and-analysis/netcdf-overview*.
10. Srivastava, A.K., Rajeevan, M. and Kshirsagar, S.R. (2009), Development of a high resolution daily gridded temperature data set (1969–2005) for the Indian region. Atmosph. Sci. Lett., 10: 249-254. *https://doi.org/10.1002/asl.232(https://imdpune.gov.in/Clim_Pred_LRF_New/Grided_Data_Download.html*.)
11. Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

12. Gridded binary*:* *http://cola.gmu.edu/grads/gadoc/aboutgriddeddata.html#formats*
13. Grib: *https://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc/*
14. NumPy: *https://numpy.org/*
15. PyNGL & PyNIO: *https://www.pyngl.ucar.edu/index.shtml*
16. Cartopy: *https://github.com/SciTools/cartopy*; *https://scitools.org.uk/cartopy/docs/latest/*
17. Regionmask*:* *https://regionmask.readthedocs.io/en/stable/*
18. Matplotlib*:* *https://matplotlib.org/*
19. Geopandas*:* *https://geopandas.org/en/stable/*
20. Pandas*:* *https://pandas.pydata.org/*
21. Xarray*:* *http://xarray.pydata.org/en/stable/*