IMD_Rainfall_Plots_Demo

September 27, 2021

0.1 Import all the required packages

[1]:
```
import rioxarray as rio
import numpy as np
import xarray as xr
import proplot as plot
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
```

0.2 Customize the Proplot package (optional)

[3]:
```
plot.rc.reset()

# Font properties (self-explanatory)
plot.register_fonts('/home/sarat/anaconda3/pkgs/proplot-0.8.1-pyhd8ed1ab_0/
↪site-packages/proplot/fonts/IBMPlexSans-SemiBold.ttf')
plot.rc['font.name'] = 'IBM Plex Sans'
plot.rc['font.weight']='bold'
plot.rc['font.size']=10

# Tick propreties (self-explanatory)
plot.rc['tick.labelsize']=10
plot.rc['xtick.minor.visible'] =   False
plot.rc['ytick.minor.visible']=   False
plot.rc['tick.len']=2
plot.rc['tick.dir']= 'out'
plot.rc['xtick.major.size']=3
plot.rc['ytick.major.size']=3

# Grid properties (self-explanatory)
plot.rc['grid']=True
plot.rc['grid.linewidth']=0.25
plot.rc['grid.linestyle']=(0, (5, 10))

# Misc
plot.rc['meta.width']=1.5 # Line width in the plots
plot.rc['subplots.tight']= True # Tight layout for the subplots
plot.rc['colorbar.insetpad']='0.5em' # Insert whitespace around the colorbar
```

0.3 Using xarray to load the climate data

For this example, we will be using the Gridded Rainfall Data from Indian Meteorological Department (IMD) which is available as a netCDF (.nc) file. NetCDF is the most commonly used file format to store gridded climate data which is also CF compliant. Download the .nc files from the given link : Rainfall Data.

- **After you've downloaded the multiple .nc files, put them all in a folder of your choice.**
- **We will use xarray to read all the multiple files at once.**

[4]:
```
#Opening multiple datasets using xarray's open_mfdataset command.

ds = xr.open_mfdataset('/media/sarat/Study/IMD_data/rain1by1/*.nc')

#### Change the file name and folder accordingly ####
```

0.4 Check the properties of the loaded dataset

[5]:
```
ds
```

[5]:
```
<xarray.Dataset>
Dimensions:    (time: 42758, lat: 33, lon: 35)
Coordinates:
  * time       (time) datetime64[ns] 1901-01-01 1901-01-02 … 2019-12-01
  * lat        (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * lon        (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
Data variables:
    rainfall   (time, lat, lon) float32 dask.array<chunksize=(42746, 33, 35),
meta=np.ndarray>
    rf         (time, lat, lon) float32 dask.array<chunksize=(365, 33, 35),
meta=np.ndarray>
Attributes:
    creation_date:  Mon Jan  7 17:07:07 IST 2019
    story:          IMD 1x1 Monthly  data in mm/day
    source:         2018_1x1_rain.nc
    title:          IMD Observed Rainfalll  2018
```

This dataset has only variable: rf. We can access this variable simply by using ds.rf command The xarray package loads this as a Data Array which has three dimensions : + Latitude (lat) + Longitude (lon) + Time (time)

The picture below provides a useful visualization of how the gridded data is arranged. For more info on how xarray works, click here.

[19]:
```
ds.rf
```

[19]:
```
<xarray.DataArray 'rf' (time: 42758, lat: 33, lon: 35)>
dask.array<where, shape=(42758, 33, 35), dtype=float32, chunksize=(389, 33, 35),
chunktype=numpy.ndarray>
```

```
Coordinates:
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2019-12-01
  * lat      (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * lon      (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
Attributes:
    long_name:  GRIDDED RAINFALL
```

Checking the longitude, latitude and time dimensions in the loaded xarray dataset

[18]:
```
ds.lon
```

[18]:
```
<xarray.DataArray 'lon' (lon: 35)>
array([ 66.5,  67.5,  68.5,  69.5,  70.5,  71.5,  72.5,  73.5,  74.5,  75.5,
        76.5,  77.5,  78.5,  79.5,  80.5,  81.5,  82.5,  83.5,  84.5,  85.5,
        86.5,  87.5,  88.5,  89.5,  90.5,  91.5,  92.5,  93.5,  94.5,  95.5,
        96.5,  97.5,  98.5,  99.5, 100.5], dtype=float32)
Coordinates:
  * lon      (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
Attributes:
    units:      degrees_east
    long_name:  longitude
```

[17]:
```
ds.lat
```

[17]:
```
<xarray.DataArray 'lat' (lat: 33)>
array([ 6.5,  7.5,  8.5,  9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
       18.5, 19.5, 20.5, 21.5, 22.5, 23.5, 24.5, 25.5, 26.5, 27.5, 28.5, 29.5,
       30.5, 31.5, 32.5, 33.5, 34.5, 35.5, 36.5, 37.5, 38.5], dtype=float32)
Coordinates:
  * lat      (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
Attributes:
    units:      degrees_north
    long_name:  latitude
```

[16]:
```
ds.time
```

[16]:
```
<xarray.DataArray 'time' (time: 42758)>
array(['1901-01-01T00:00:00.000000000', '1901-01-02T00:00:00.000000000',
       '1901-01-03T00:00:00.000000000', …, '2019-10-01T00:00:00.000000000',
       '2019-11-01T00:00:00.000000000', '2019-12-01T00:00:00.000000000'],
      dtype='datetime64[ns]')
Coordinates:
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2019-12-01
Attributes:
    long_name:  time corresponding to 1st encountered time of current month
```

0.5 Performing operations on the rainfall data

First, we will slice/select the data according to our needs + Selecting a specific date, **for example 1995-05-17** + Selecting a specific location (latitude and longitude): **Latitude : 18.5, Longitdue: 82.5**

To perform operations on all the variables in the dataset, we can directly use the original dataset variable (ds). To perform operations on a specific variable, such as rainfall (rf), we can explicitly pass the variable name (ds.rf) before performing any operation.

- In our case, since we have only one variable, we can also directly operate on **ds** without specifying the variable.
- However, to be as general as possible, we will explicitly pass the **rainfall (ds.rf)** variable before performing any operatiom.

[32]:
```
ds_sel_time = ds.rf.sel(time='1995-05-17T00:00:00.000000000')
ds_sel_loc = ds.rf.sel(lat=18.5,lon=82.5)
```

[33]:
```
ds_sel_time # This is a 2-D array of rainfall values on that particular time␣
↪value.
```

[33]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<getitem, shape=(33, 35), dtype=float32, chunksize=(33, 35),
chunktype=numpy.ndarray>
Coordinates:
    time     datetime64[ns] 1995-05-17
  * lat      (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * lon      (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
Attributes:
    long_name:  GRIDDED RAINFALL
```

[22]:
```
ds_sel_loc # This is 1-D time-series of rainfall values for that particular␣
↪location.
```

[22]:
```
<xarray.DataArray 'rf' (time: 42758)>
dask.array<getitem, shape=(42758,), dtype=float32, chunksize=(389,),
chunktype=numpy.ndarray>
Coordinates:
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2019-12-01
    lat      float32 18.5
    lon      float32 82.5
Attributes:
    long_name:  GRIDDED RAINFALL
```

0.6 Now, we will perform operations on the time axis of the rainfall dataset.

- Mean over time
- Variance over time

These operations can be perorfmed along any dimension other than time.

[27]:
```
ds_mean = ds.rf.mean('time')
ds_var = ds.rf.var('time')
```

[29]:
```
ds_mean
```

[29]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<mean_agg-aggregate, shape=(33, 35), dtype=float32, chunksize=(33,
35), chunktype=numpy.ndarray>
Coordinates:
  * lat      (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * lon      (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
```

[30]:
```
ds_var
```

[30]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<moment_agg-aggregate, shape=(33, 35), dtype=float32, chunksize=(33,
35), chunktype=numpy.ndarray>
Coordinates:
  * lat      (lat) float32 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * lon      (lon) float32 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
```

Plotting the rainfall data

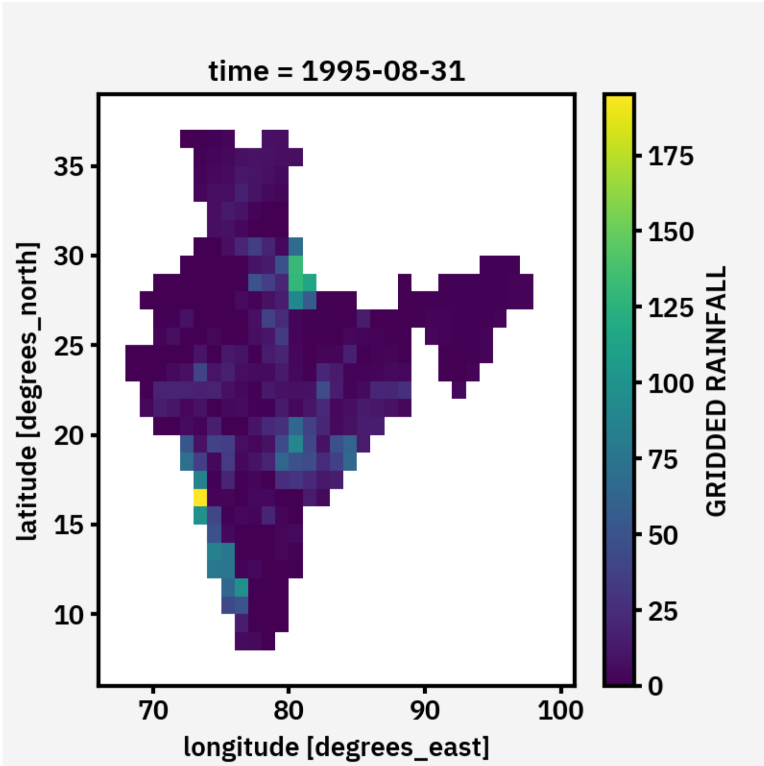
0.7 Default xarray plot commands (which use matplotlib) to generate plots.

Remember that we can only plot arrays upto 2 dimensions only.

So, we can either select a slice of the original dataset or plot the 2-D arrays and 1-D time series that we generated earlier.

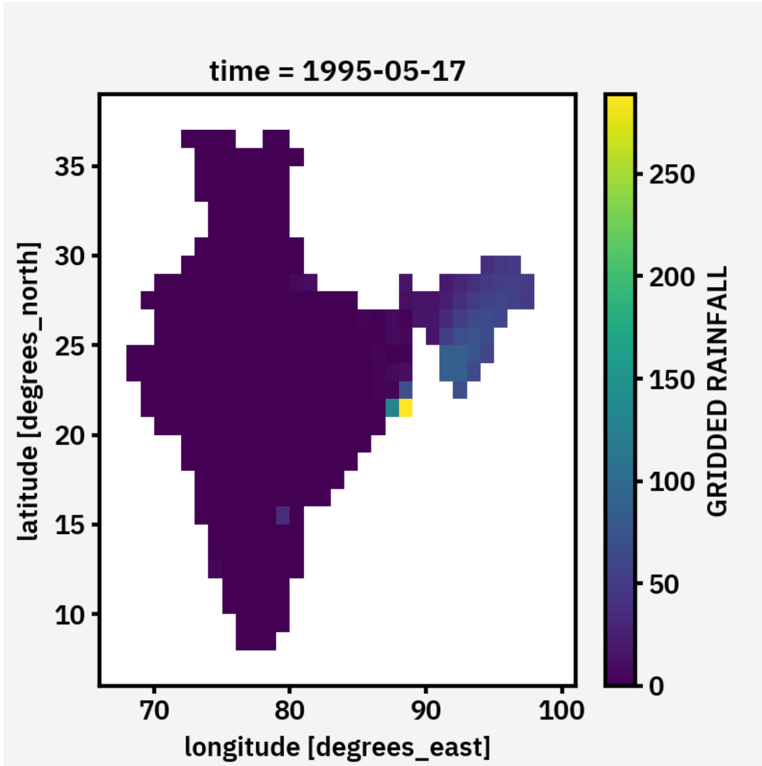
[38]:
```
ds.rf.sel(time='1995-08-31T00:00:00.000000000').plot() # extracting a specific␣
↪time slice and plotting it.
```

[38]:
```
<matplotlib.collections.QuadMesh at 0x7f5142977160>
```



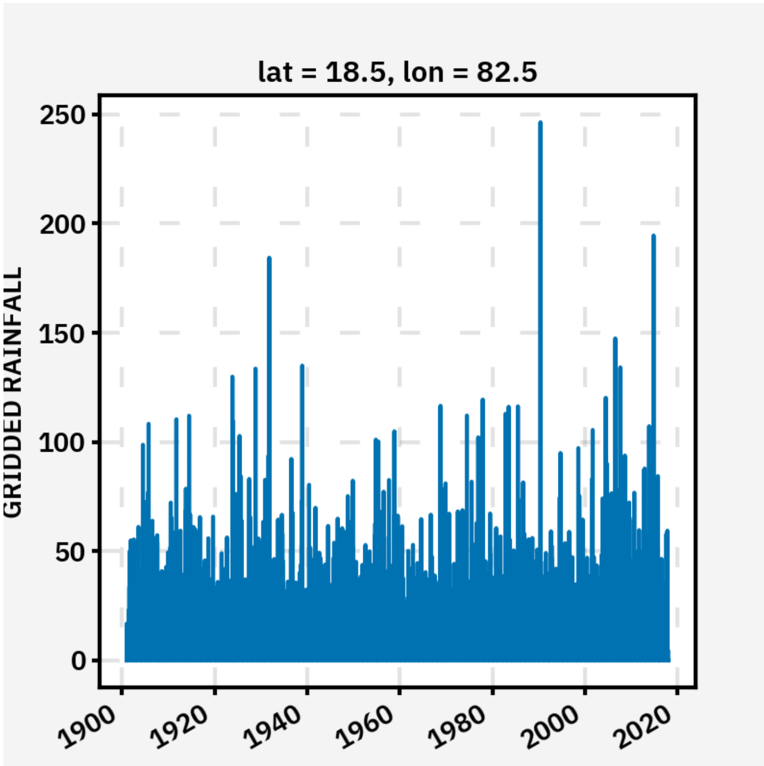
[35]:
```
ds_sel_time.plot() # Same as above but here we directly load the variable that␣
↪we extracted earlier.
```

[35]: <matplotlib.collections.QuadMesh at 0x7f51412d3ca0>



[39]:
```
ds_sel_loc.plot()
```

[39]: [<matplotlib.lines.Line2D at 0x7f514284ebb0>]



1 Using the Proplot package to generate publication qualilty plots

Matplotlib is an extremely versatile plotting package used by scientists and engineers far and wide. However, matplotlib can be cumbersome or repetitive for users who…

- Make highly complex figures with many subplots.
- Want to finely tune their annotations and aesthetics.
- Need to make new figures nearly every day.

More info on proplot can be found here.

[61]:
```
# Generate the figure and axis with nrows and ncols for subplots ###
fig, axs=plot.subplots(ncols=2,nrows=1, proj='cyl', dpi=300,
                        tight=True)

##### proj = 'cyl' is the Cylindrical Equidistant Map projection used by␣
 ↪Cartopy ###
#### dpi = 300 ( recommended ) , 600 , 1200

lat_min = 6 # Change accordingly
lat_max = 38 # lat max
lon_min = 66 ###
```

```
lon_max = 98
levels=np.arange(0,10,1) # generates a sequence of numbers from 0 to 10 with  a␣
 ↪spacing of 1
cm = 'RdYlBu' # Colormap 'rainbow' , 'viridis', 'RdYlBu', 'RdBu' etc..
ex= 'max' # Color bar arrow ,'min', 'max', 'none','both'

#Now, we can format all the axes at once using these commands

axs.format(lonlim=(lon_min, lon_max),
           latlim=(lat_min, lat_max),
           labels=True,
           innerborders=False,
           latlines=4, lonlines=2,
           abc='(a)', abcloc='ll',
           gridminor=False,
           suptitle='IMD Rainfall' )

######## Limits as above; ### labels = True for lat lon labels,
###### inner borders = False , If True, it will show rivers #####
###latlines=1, lonlines=1  spacing ########
#abc=False, It abc='(a)', it will automatically give subplot (a),(b),(c) etc....
####abcloc='ll', abc location
#### gridminor=False; if true it will show all gridlines of lat , lon


##########Subplots ###############

#contourf for contours

#pcolormesh for psuedo color plot

#Each subplot axis is numbered as axs[0] or axs[1] etc....]

# 1st subplot

m=axs[0].contourf(ds_mean,     # Data to be plotted
                    cmap=cm,  # Colormap
                  extend=ex,
                 transform=ccrs.PlateCarree(), # cartopy map projection
                  levels=levels )

axs[0].format(title='Mean Rainfall Contour')


# 2nd subplot

n=axs[1].pcolormesh(ds_mean,
```

```
                    cmap=cm,
                    extend=ex,
                    transform=ccrs.PlateCarree(),
                    levels=levels )

axs[1].format(title='Mean Rainfall Pcolormesh')

# Colorbar

fig.colorbar(m,loc='b',drawedges=True, width = 0.10 , length=0.45, label='mm/
 ↪day')

#fig.colorbar will ive 1 common colorbar for all plots. But for common colorbar␣
 ↪give explict levels.

#Use axs[0].colorbar for individual colorbars ########

# axs[1].colorbar(n,loc='b',drawedges=True, width = 0.10 , length=0.65, label=␣
 ↪'Rainfall')
```

```
/home/sarat/anaconda3/lib/python3.8/site-packages/dask/array/numpy_compat.py:39:
RuntimeWarning: invalid value encountered in true_divide
  x = np.divide(x1, x2, out)
```

[61]: <matplotlib.colorbar.Colorbar at 0x7f50f8b6a310>