IMD_Rainfall_Plots_Demo

December 22, 2022

0.1 Import all the required packages

[ ]:
```
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)

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

# Font properties (self-explanatory)
plot.rc['font.weight']='bold'
plot.rc['font.size']=14

# Tick propreties (self-explanatory)
plot.rc['tick.labelsize']=14
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']=False
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
plot.rc['label.size'] = 'medium'
```

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.**

[ ]:
```
#Opening multiple datasets using xarray's open_mfdataset command. To open a␣
↪single nc file, you can use the xr.open_dataset() comamnd.

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

#### Change the file name and folder accordingly ####
# Check the properties of the loaded dataset by printing the variable
ds
```

[ ]:
```
<xarray.Dataset>
Dimensions:  (lon: 35, lat: 33, time: 44195)
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2021-12-31
Data variables:
    rf       (time, lat, lon) float64 dask.array<chunksize=(365, 33, 35),
meta=np.ndarray>
Attributes:
    CDI:          Climate Data Interface version 1.6.3 (http://code.zmaw.de/p…
    Conventions:  CF-1.4
    history:      Mon Jun 01 15:26:15 2020: cdo splityear ../imd_rf_1x1_1901_…
    CDO:          Climate Data Operators version 1.6.3 (http://code.zmaw.de/p…
```

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.

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

[ ]:
```
<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

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

[ ]:
```
<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
```

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

[ ]:
```
<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
```

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

[ ]:
```
<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
```

0.4 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 operation.

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

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

[ ]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<getitem, shape=(33, 35), dtype=float64, chunksize=(33, 35),
chunktype=numpy.ndarray>
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
    time     datetime64[ns] 1995-05-17
Attributes:
    long_name:  GRIDDED RAINFALL
```

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

[ ]:
```
<xarray.DataArray 'rf' (time: 44195)>
dask.array<getitem, shape=(44195,), dtype=float64, chunksize=(1096,),
chunktype=numpy.ndarray>
Coordinates:
    lon      float64 82.5
    lat      float64 18.5
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2021-12-31
Attributes:
    long_name:  GRIDDED RAINFALL
```

We can also use the isel() function to use indices to select and subset the data + Selecting a specific date using the index (0): **for example the first time step which is 1901-01-01** + Selecting a specific location using the index (10,10)(latitude and longitude): **Latitude : 16.5, Longitdue: 76.5**

[ ]:
```
ds_sel_time_idx = ds.rf.isel(time=0) # time selection
ds_sel_loc_idx = ds.rf.isel(lat=10,lon=10) # location selection
```

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

[ ]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<getitem, shape=(33, 35), dtype=float64, chunksize=(33, 35),
chunktype=numpy.ndarray>
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
    time     datetime64[ns] 1901-01-01
Attributes:
    long_name:  GRIDDED RAINFALL
```

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

[ ]:
```
<xarray.DataArray 'rf' (time: 44195)>
dask.array<getitem, shape=(44195,), dtype=float64, chunksize=(1096,),
chunktype=numpy.ndarray>
Coordinates:
    lon      float64 76.5
    lat      float64 16.5
  * time     (time) datetime64[ns] 1901-01-01 1901-01-02 … 2021-12-31
Attributes:
    long_name:  GRIDDED RAINFALL
```

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

- Mean over time
- Variance over time
- Grouping over time
- Resampling over time

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

[ ]:
```
# Mean and Variance of the data along the time axis.
ds_mean = ds.rf.mean('time')
ds_var = ds.rf.var('time')
```

[ ]:
```
ds_mean
```

[ ]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<mean_agg-aggregate, shape=(33, 35), dtype=float64, chunksize=(33,
35), chunktype=numpy.ndarray>
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
```

[ ]:
```
ds_var
```

[ ]:
```
<xarray.DataArray 'rf' (lat: 33, lon: 35)>
dask.array<moment_agg-aggregate, shape=(33, 35), dtype=float64, chunksize=(33,
35), chunktype=numpy.ndarray>
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
```

We can also group the data along the time axis into either hours, days, months and years. The, we can apply methods such as mean and variance to the grouped dataset.

[ ]:
```
ds_year=ds.groupby('time.year') # Also, we can use 'time.month',  'time.day'␣
 ↪and 'time.dayofyear' for grouping.
ds_year_mean = ds_year.mean('time')
ds_year_mean # Annual mean rainfall
```

[ ]:
```
<xarray.Dataset>
Dimensions:  (lon: 35, lat: 33, year: 121)
Coordinates:
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * year     (year) int64 1901 1902 1903 1904 1905 … 2017 2018 2019 2020 2021
Data variables:
    rf       (year, lat, lon) float64 dask.array<chunksize=(1, 33, 35),
meta=np.ndarray>
```

Using the 'time.dayofyear' argument to get the daily climatology of rainfall

[ ]:
```
ds_daily_clim=ds.groupby('time.dayofyear').mean('time') # we get the mean of␣
 ↪each day across all years
ds_daily_clim
# notice that the final dimensions are (lat,lon,dayofyear)
```

[ ]:
```
<xarray.Dataset>
Dimensions:    (lon: 35, lat: 33, dayofyear: 366)
Coordinates:
  * lon        (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat        (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
  * dayofyear  (dayofyear) int64 1 2 3 4 5 6 7 8 … 360 361 362 363 364 365 366
Data variables:
    rf         (dayofyear, lat, lon) float64 dask.array<chunksize=(1, 33, 35),
meta=np.ndarray>
```

We can use the resample function to sample our data at a different resolution.

[ ]:
```
ds_res_month = ds.resample(time='1M') # Valid arguments are '1M','1D' and '1Y'.
# Then, we can apply mean, sum,variance etc.
```

```
ds_res_month.sum('time') # Monthly Rainfall Accumulation
```

[ ]:
```
<xarray.Dataset>
Dimensions:  (time: 1452, lon: 35, lat: 33)
Coordinates:
  * time     (time) datetime64[ns] 1901-01-31 1901-02-28 … 2021-12-31
  * lon      (lon) float64 66.5 67.5 68.5 69.5 70.5 … 97.5 98.5 99.5 100.5
  * lat      (lat) float64 6.5 7.5 8.5 9.5 10.5 … 34.5 35.5 36.5 37.5 38.5
Data variables:
    rf       (time, lat, lon) float64 dask.array<chunksize=(1, 33, 35),
meta=np.ndarray>
```

Plotting the rainfall data

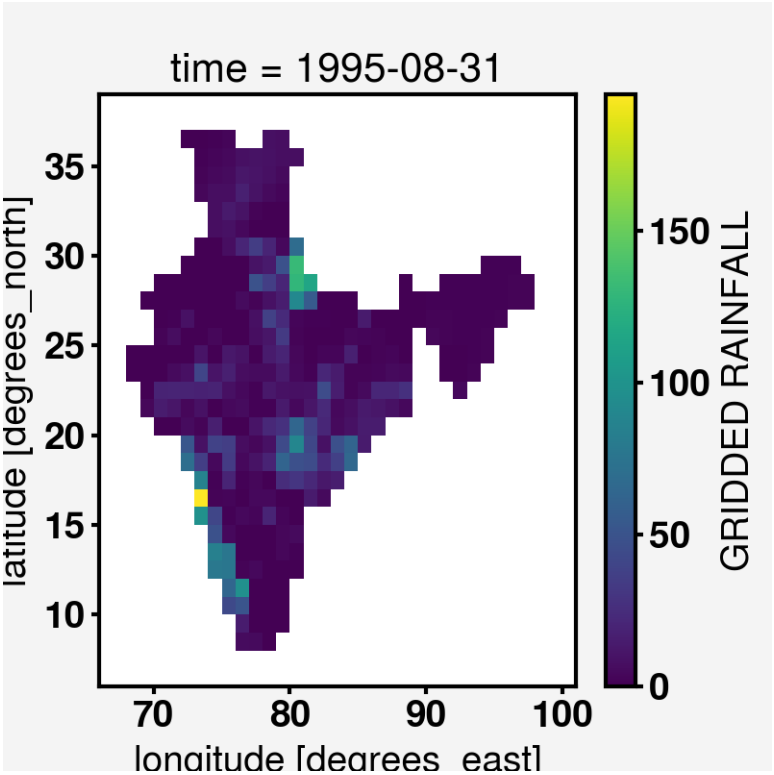
0.6 Default xarray plot commands (which use matplotlib) to generate plots.

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.

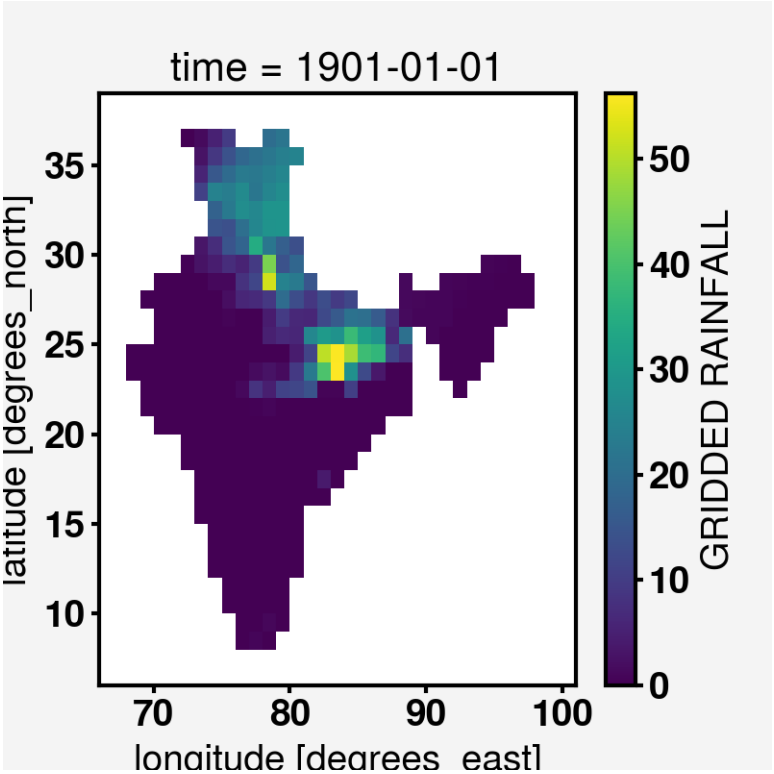
[ ]:
```
ds.rf.sel(time='1995-08-31T00:00:00.000000000').plot() # extracting a specific␣
↪time slice and plotting it.
```

[ ]:
```
<matplotlib.collections.QuadMesh at 0x7fae9094d8e0>
```



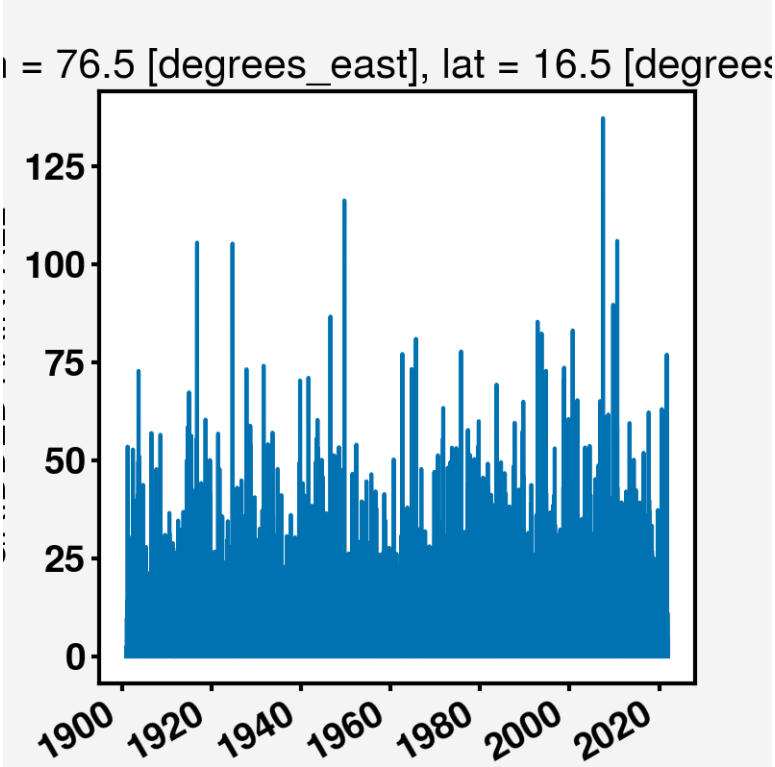
[ ]:
```
ds_sel_time_idx.plot() # Same as above but here we directly load the variable␣
↪that we extracted earlier.
```

[ ]: <matplotlib.collections.QuadMesh at 0x7fae8ecffdc0>



[ ]:
```
ds_sel_loc_idx.plot()
```

[ ]: [<matplotlib.lines.Line2D at 0x7fae9109ecd0>]



Using the globals() functions is to generate the variable names within a loop. **We will be rquiring to call variables within a loop as follows : if a variable is stored as ds_clim, we can use globals()['ds_clim'] to get the ds_clim variable *This command is very useful while plotting the data.***

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.

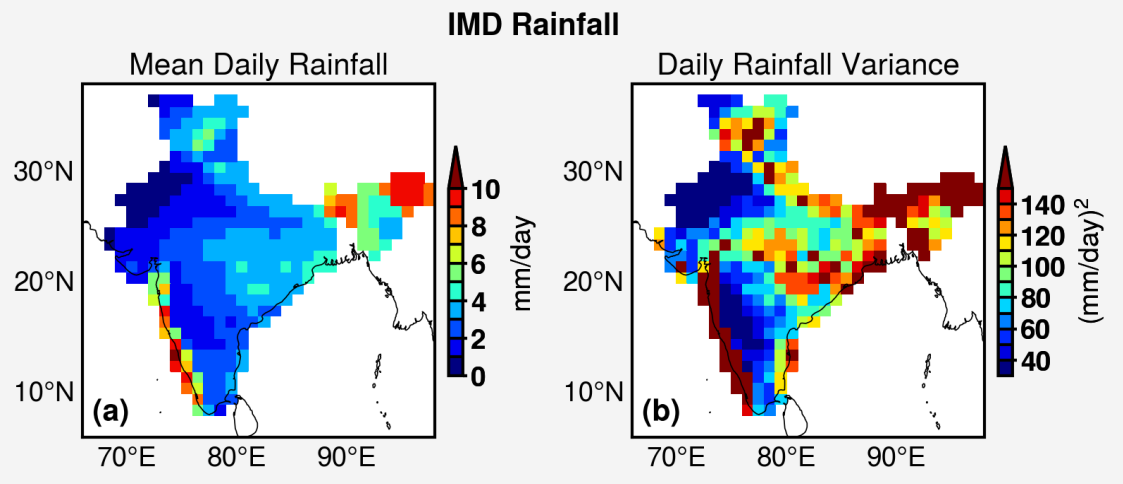
More info on proplot can be found here.

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

##### proj = 'cyl' is the Cylindrical Equidistant Map projection used by␣
 ↪Cartopy ###
```

```
lat_min = 6 # # Minimum latitude on the map
lat_max = 38 # Maximum latitude on the map
lon_min = 66 # # Minimum longitude on the map
lon_max = 98 # # Maximum longitude on the map

cm = 'jet' # 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=10, lonlines=10,
           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

# Subtitles for each plot
# We can declare the subtitles now and use it further
title=['Mean Daily Rainfall','Daily Rainfall Variance' ]

# We can declare varaible names now and use it further
vars =['ds_mean','ds_var']
levels_mean= np.arange(0,11,1)  # generates a sequence of numbers from 0 to 10␣
 ↪with  a spacing of 1
levels_var= np.arange(30,160,10)
levels =['levels_mean','levels_var'] # for indexing later
labels = ['mm/day','$(mm/day)^2$'] # for indexing later

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

#contourf for contours

#pcolormesh for psuedo color plot

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

```
# We can loop over each plot by indexing it as axs[i] where i is the indexing␣
 ↪variable

for i in range(0,2):
        ########################

        m=axs[i].pcolor(globals()[vars[i]], # globals() and vars[i] are used to␣
 ↪index and generate the variable name
        cmap=cm, # Colormap
        transform=ccrs.PlateCarree(), # cartopy map projection
        extend=ex,  #colorbar style
        levels = globals()[levels[i]] # using globals to get levels variable
        )
        axs[i].format(title=title[i])
        axs[i].coastlines() # adds coastlines
        axs[i].colorbar(m,loc='r',drawedges=True, width = 0.10 , length=0.65,␣
 ↪label=labels[i])



# Colorbar

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

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

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

```
/home/carbform/anaconda3/lib/python3.9/site-
packages/dask/array/numpy_compat.py:41: RuntimeWarning: invalid value
encountered in true_divide
  x = np.divide(x1, x2, out)
```


IMD Rainfall
Mean Daily Rainfall
Daily Rainfall Variance
30°N
20°N
10°N
70°E
80°E
90°E
(a)
(b)
10
8
6
4
2
0
mm/day
140
120
100
80
60
40
(mm/day)²