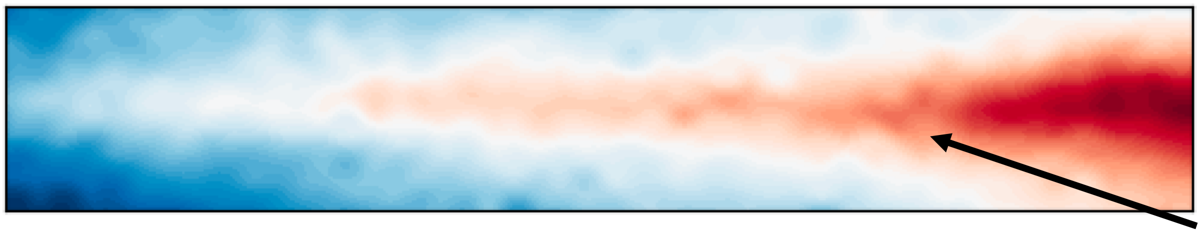
Lecture 3
Commonly used indexes in geoscience and python plotting

ENSO Index

El Nino Southern Oscillation (ENSO) is a climate oscillation occurring with a period of every 3 to 7 years.

We call an ”El Nino event” **the unusual warming of surface waters in the eastern equatorial Pacific Ocean**

Nino 3.4 area



Warm SST

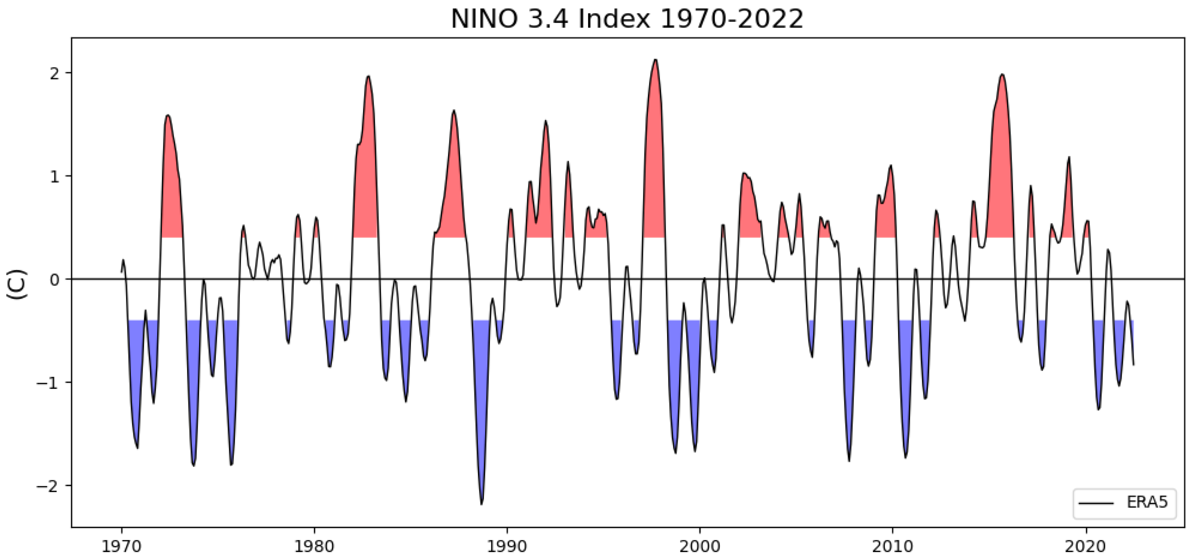
There exist several indexes (all based on SST) to estimate the strength of ENSO.
See: https://climatedataguide.ucar.edu/climate-data/nino-sst-indices-nino-12-3-34-4-oni-and-tni

*“Niño 3.4 (**5N-5S, 170W-120W**): The Niño 3.4 anomalies may be thought of as representing the average equatorial SSTs across the Pacific from about the dateline to the South American coast. The Niño 3.4 index typically **uses a 5-month running mean**, and El Niño or La Niña events are defined when the Niño 3.4 SSTs exceed **+/- 0.4C** for a period of six months or more.”*

ENSO Index

Steps to compute the NINO 3.4 index (from NCAR Climate Data Guide):

- Compute area averaged SST from Niño X region to get a SST timeseries;
- Compute a climatological mean (and its standard deviation) over 30 years of data (e.g., 1950-1979), this is called ‘reference period’;
- Subtract the climatology from the SST timeseries to obtain anomalies;
- Smooth the anomalies with a 5-month running mean;
- Normalize the smoothed values by the standard deviation over the climatological period.



ENSO Index

Computing the NINO 3.4 Index in python:

```
#Load monthly SST data
D1=iris.load_cube('SST_ERA5_monthly_1970_2022.nc')
print (D1)

#Extract NINO3.4 area
where='NINO_3_4'
D1=extract_area(D1, where=where)
```

SAME FUNCTION OF LECTURE 4

```
#Alternatively
D1=iris.load_cube('SST_ERA5_monthly_1970_2022.nc',
                iris.Constraint(latitude= lambda lat: -5 <= lat <= 5,
                longitude= lambda lon: 190 <= lon <= 240  ))

#Call function to compute ENSO
ENSO=compute_ENSO(D1)

#Plot ENSO
Plot_ENSO(ENSO, label='ERA5', title='NINO 3.4 Index 1970-2022')
plt.show()
```

ENSO Index

```
################### CALCULATE ENSO INDEX ###############################

def compute_ENSO(data_in):

 #Compute area weighted mean
 data_in=area_weighted(data_in)

 #compute a climatological mean (and its standard deviation) over the first 30 years
 mean=data_in[:30*12].collapsed('time', iris.analysis.MEAN)
 std_dev=data_in[:30*12].collapsed('time', iris.analysis.STD_DEV)

 #calculate anomalies:
 ENSO=data_in-mean

 #apply a 5-month running mean:
 months=ENSO.shape[0]
 ENSO_5month=iris.cube.CubeList()

 for i in range (0, months-5):
   ENSO_5month.append(ENSO[i:i+5].collapsed('time', iris.analysis.MEAN))

 ENSO_5month= ENSO_5month.merge_cube()
```

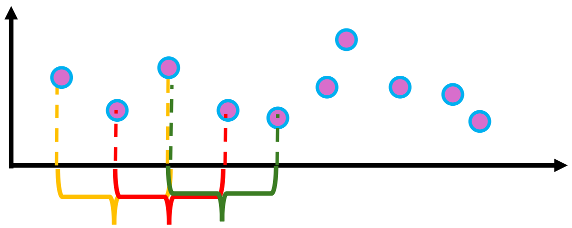
SAME FUNCTION OF LECTURE 4

Running Mean
/Moving Average

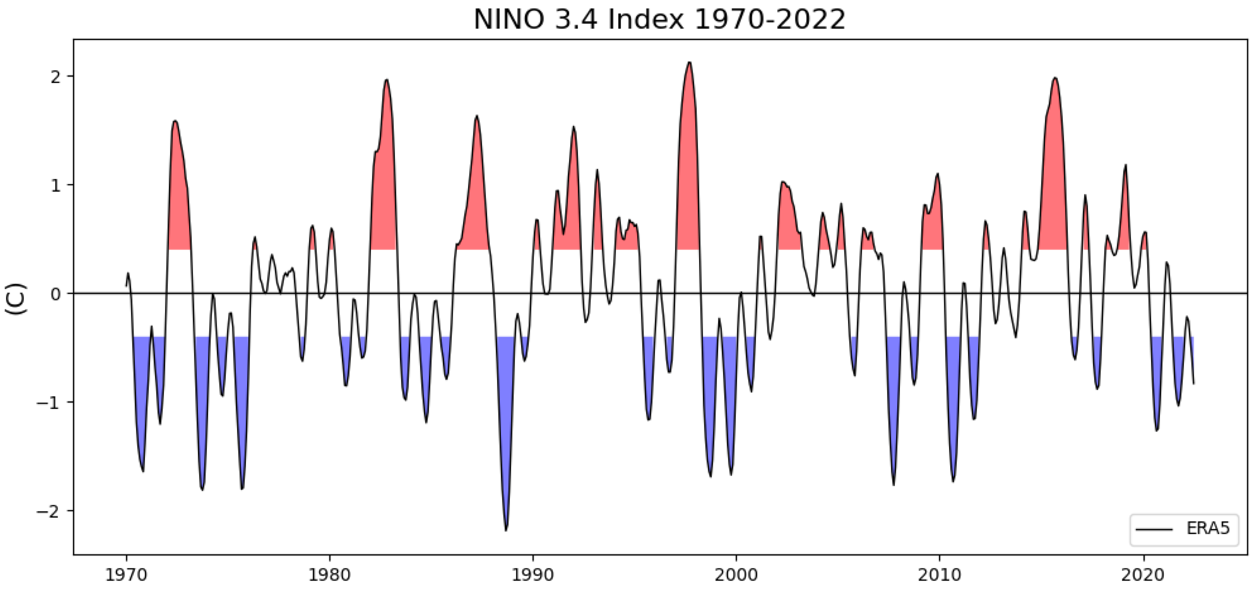
ENSO Index

```
CONTINUED …

#finally, normalize the timeseries by the standard deviation
 ENSO_norm=ENSO_5month/std_dev

 return(ENSO_norm)
##############################################################
```

Running Mean:
It is a way to smooth (or filter) a time-series by removing high-frequency components (the ‘noise’) to highlight timescale we are interested in.
Common window length for rolling means:
3/5 months (to remove very high frequencies in the dataset)
12 months (to filter intra-annual variations)
11 year (this is the length of a solar cycle)



ENSO Index



Shading selected values in a time-series in python (see also DA_exercises_2_solutions.py):

```
plt.fill_between(date, 0.4, np.ma.masked_where(data_set.data <= 0.4, data_set.data) , alpha=0.5,
facecolor='red')
plt.fill_between(date, -0.4, np.ma.masked_where(data_set.data >= -0.4, data_set.data) , alpha=0.5,
facecolor='blue')
```

Maps

Python Basemap provides 24 projections, see documentation below:

https://matplotlib.org/basemap/stable/users/mapsetup.html#:~:text=Basemap%20provides%2024%20different%20map,the%20map%20projection%20will%20describe.

Gall Stereographic Projection ‘gall’

```
#Plot global map
bmap=Basemap(projection= 'gall', llcrnrlat= -90,  urcrnrlat= 90,
llcrnrlon=0,  urcrnrlon= 360, resolution='l')

llcrnrX: lower left corner
urcrnrX: upper right corner


#Plot selected area (North Atlantic)
 bmap=Basemap(projection= 'gall', llcrnrlat= 0,  urcrnrlat= 60, llcrnrlon=
280,  urcrnrlon= 360, resolution='l')
```

Maps

Polar Stereographic Projection ‘npstere’, ‘spstere’

```
#Plot Southern Hemisphere
bmap=Basemap(projection='spstere',boundinglat=-55,lon_0=180,resolution='l')
```

```
boundinglat: ‘cutting’ latitude
lon_0 : sets the orientation of your map (“the longitude at 6 o’clock”)
```

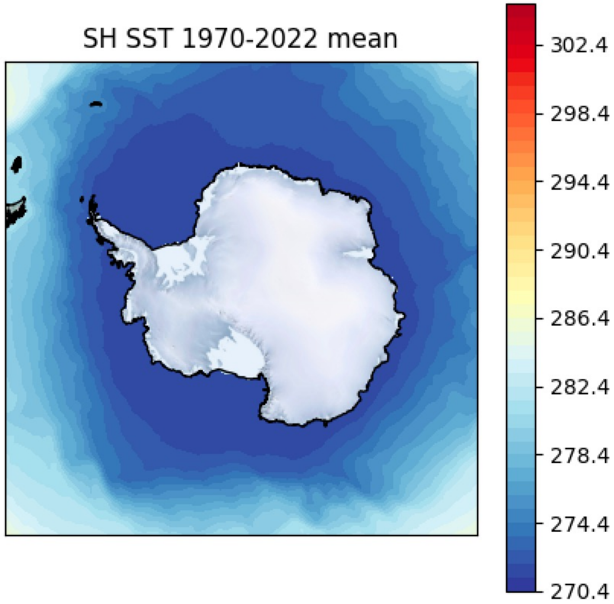
Maps

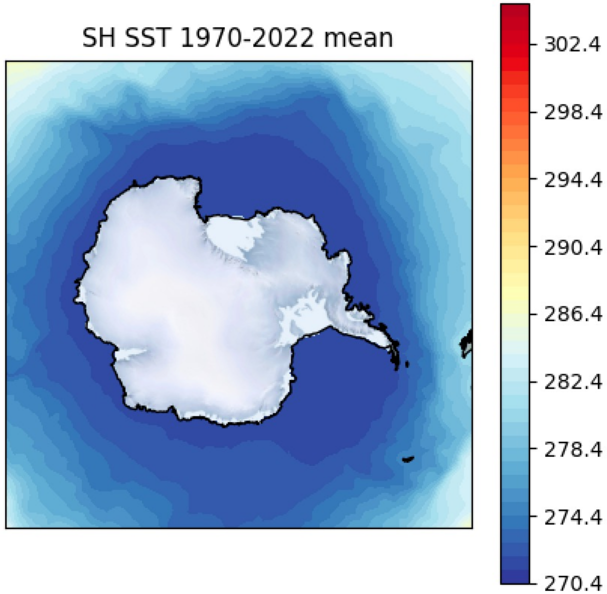
Polar Stereographic Projection `'npstere'`, `'spstere'`

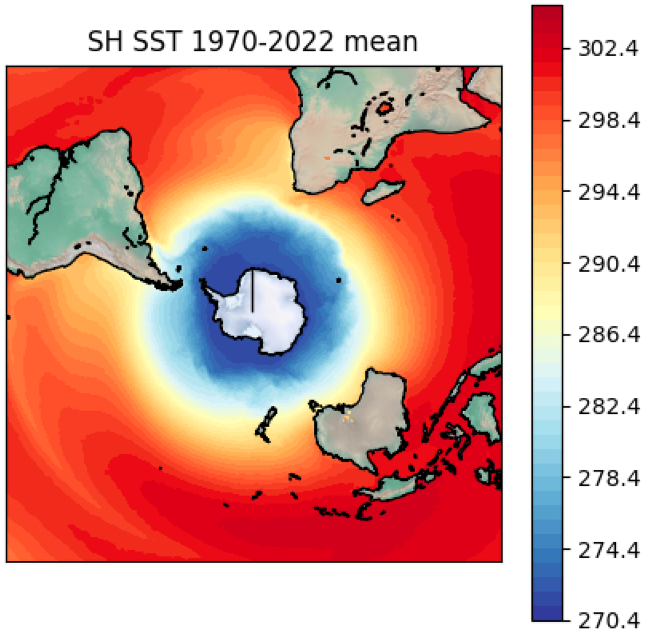
```
boundinglat=-55
lon_0=180
```

```
boundinglat=-55
lon_0=0
```

```
boundinglat=0
lon_0=180
```







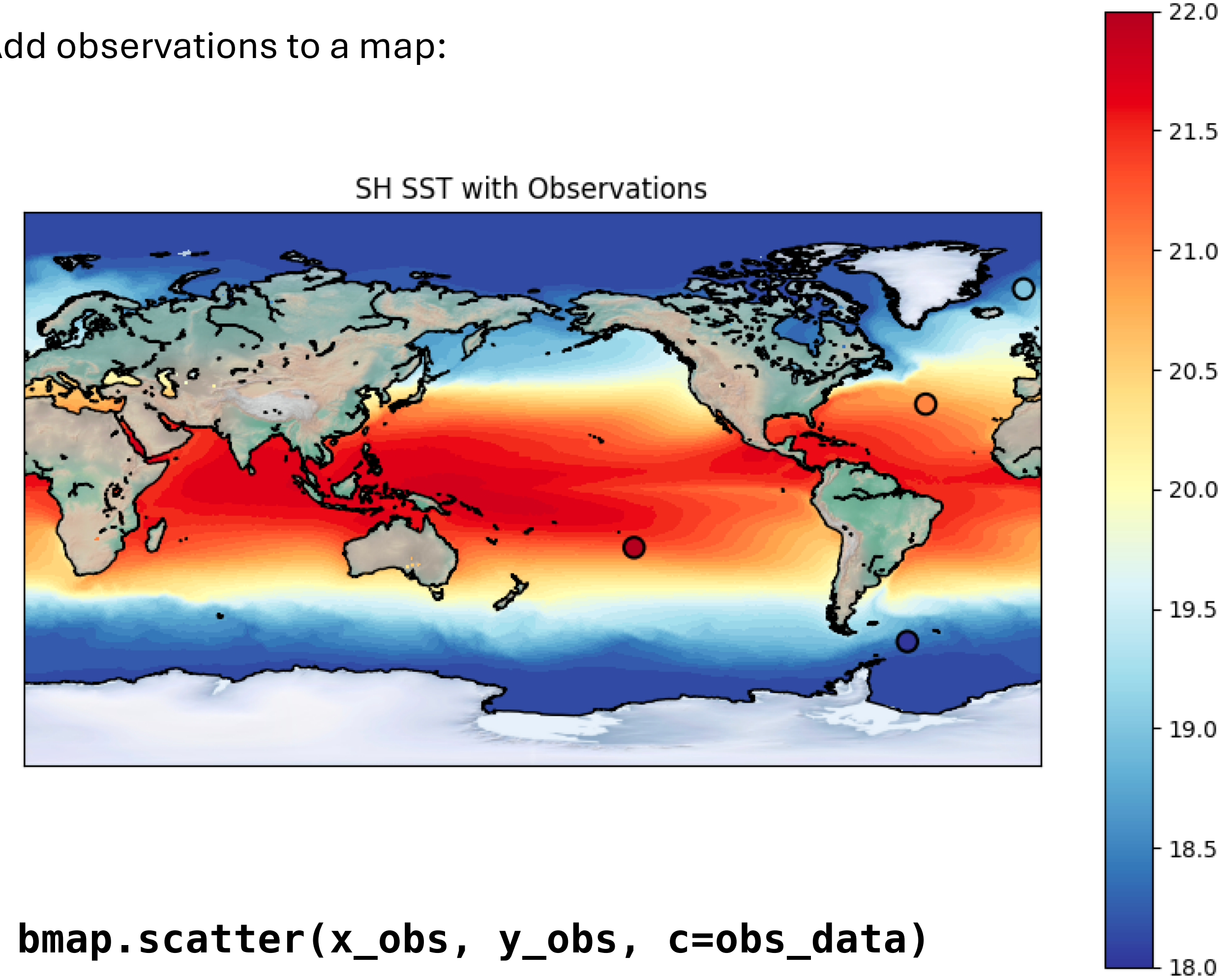
Maps

A generic plotting function in python:

```
################### Plotting Function ############################################

def plot_map(data_in, cmap, levels, title):

 fig=plt.figure(figsize=(5,5))
 ax=plt.gca()

 #bmap=Basemap(projection= 'gall', llcrnrlat= -90,  urcrnrlat= 90, llcrnrlon=0,  urcrnrlon= 360, 
resolution='l')
 bmap=Basemap(projection='spstere',boundinglat=-55,lon_0=180,resolution='l')

 lon= data_in.coord('longitude').points
 lat= data_in.coord('latitude').points
 x,y=bmap(*np.meshgrid(lon,lat))
 contours=bmap.contourf(x,y, data_in.data, levels, cmap=cmap)

 bmap.shadedrelief(scale=0.5)
 bmap.drawcoastlines()
 plt.colorbar()
 plt.title(title)

 return()
```

Maps

Add observations to a map:




```
bmap.scatter(x_obs, y_obs, c=obs_data)
```

Maps

Add observations to a map:

```
#Make up some 'observational' data (lat, lon, SST value) and plot it on a map
lats=[71.78, 34.12, -23.85, -57.71]
lons=[360-6, 360-40.65, 360-143.90, 360-47.17]
sst=[19,21,22,18]
sites=[lats,lons,sst]

plot_map_obs(D1, sites, cmap='RdYlBu_r', levels=50, title='SH SST with Observations')
```

```
def plot_map_obs(data_in, sites, cmap, levels, title):

 fig=plt.figure(figsize=(10,10))
 ax=plt.gca()

 #define latitude, longitude and data value of observations
 obs_lat=sites[0]
 obs_lon=sites[1]
 obs_sst=sites[2]
```

Maps

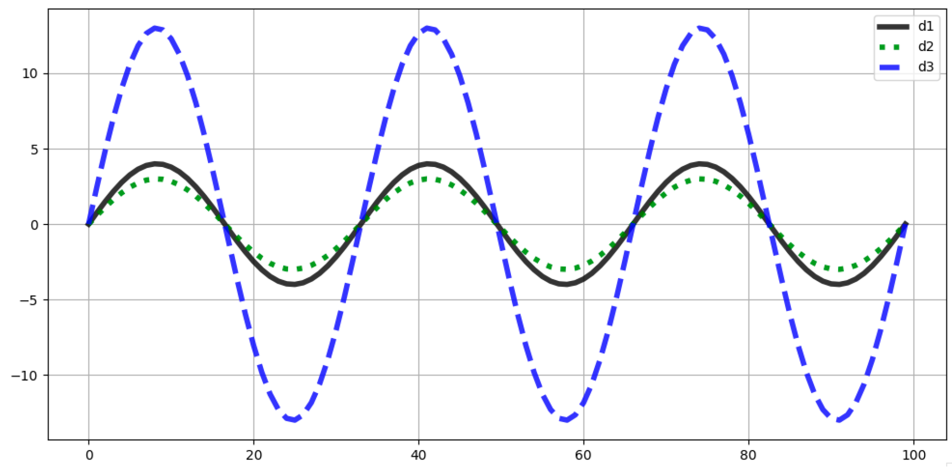
Add observations to a map:

```
 bmap=Basemap(projection= 'gall', llcrnrlat= -90,  urcrnrlat= 90, llcrnrlon=0,  urcrnrlon= 360,
resolution='l')

 lon= data_in.coord('longitude').points
 lat= data_in.coord('latitude').points
 x,y=bmap(*np.meshgrid(lon,lat))
 contours=bmap.contourf(x,y, data_in.data, levels, cmap=cmap)

 x_obs,y_obs = bmap(obs_lon,obs_lat) #transform coordinates into same projection used for model data
 obs=bmap.scatter(x_obs, y_obs, c=obs_sst, s=80, marker='o', linewidth=1.5,
edgecolors='black',  cmap=cmap)

 bmap.shadedrelief(scale=0.5)
 bmap.drawcoastlines()
 plt.colorbar()
 plt.title(title)

 return()
```

Working with many datasets

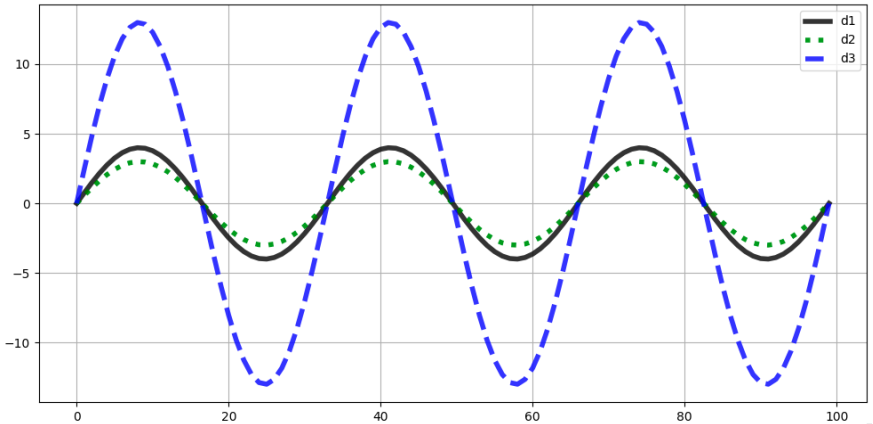
Efficient way of plotting when working with many datasets:

```
def plot_data(data_list, style, color, label):


 fig=plt.figure(tight_layout=True, figsize=(10, 5))
 ax=plt.gca()

 for i in range(0, len(data_list)):
   plt.plot(data_list[i], c=color[i], linewidth=4, linestyle=style[i], alpha=0.8, label=label[i])

 plt.legend()
 plt.grid()

 return()
```



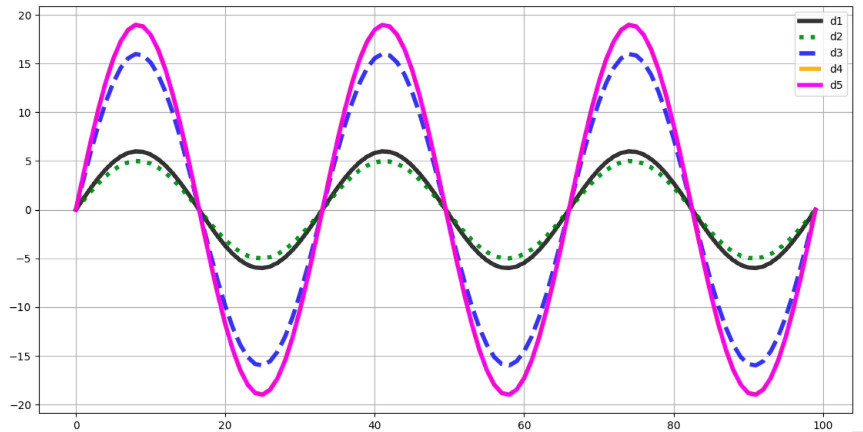
Working with many datasets

Efficient way of plotting when working with many datasets:

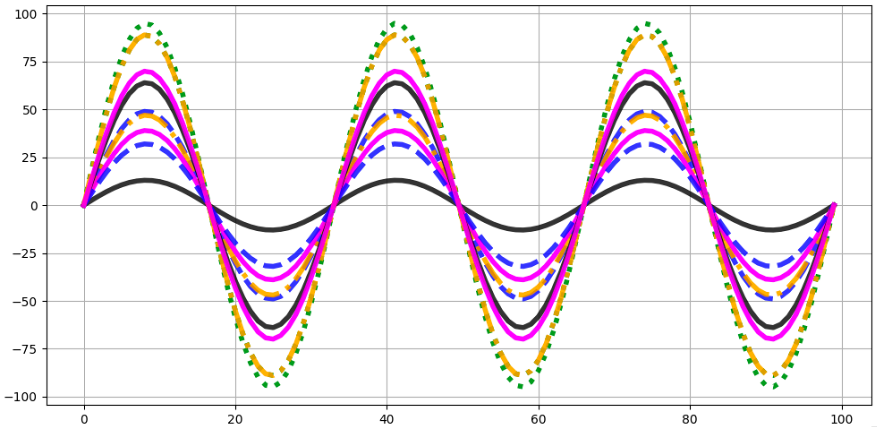


`len(data_list)=3`

`len(data_list)=5`



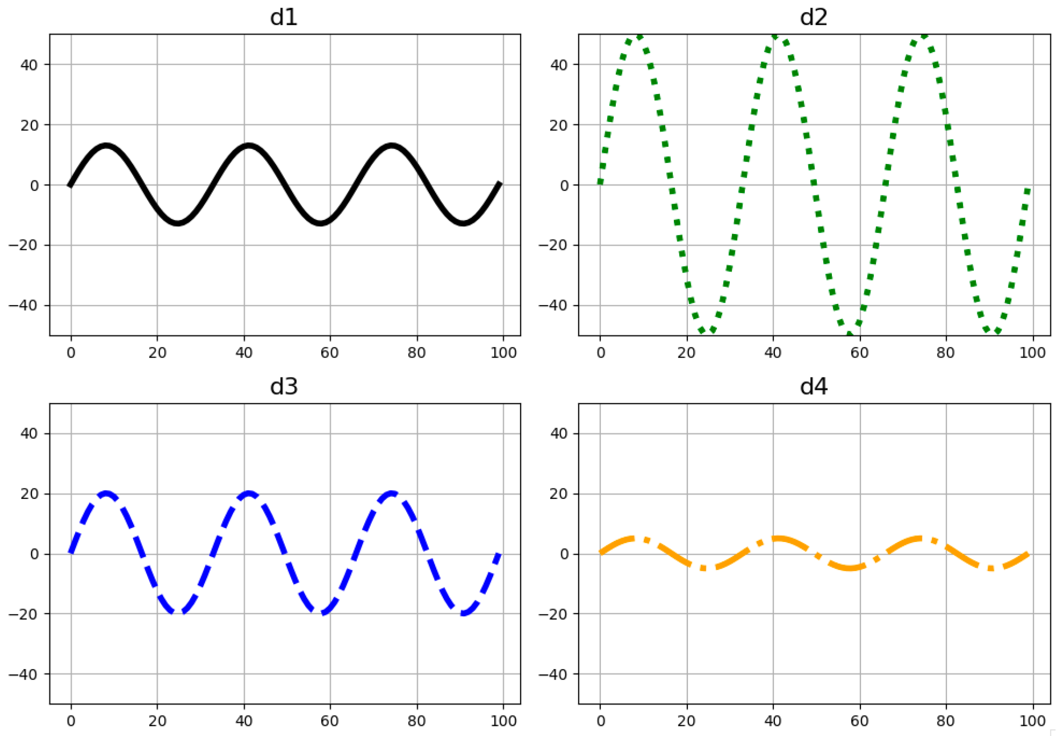
`len(data_list)=10`



Working with many datasets

Efficient way of plotting when working with many datasets:

```
def plot_panels(data_list, style, color, title):

 fig=plt.figure(figsize=(10,7),tight_layout=True)

 n_r = 2 #number of rows
 n_c = 2 #number o columns


 for i in range (0, len(data_list)):
                  #n_of_row, n_of_columns, plot_id
  ax = fig.add_subplot(int(n_r), int(n_c), i+1)
  ax=plt.gca()
  plt.plot(data_list[i], c=color[i],
         linewidth=4, linestyle=style[i])

  plt.grid()
  plt.ylim(-50,50)
  plt.title(title[i], fontsize=16)


 return()
```



Command-line arguments

When processing large datasets you might have to process them in stages (e.g.. in batches of 10 years) and you might want/need to run python scripts on a cluster as a batch job.

To simplify this process, you could consider pass on to python keyword arguments via the command line (i.e. not in your python script, but in your ' .sh' submission script).
Advantage : *you can modulate the script behavior without having to modify the source code every time.*

End year

Logical argument

```
>> python3 process_ERA5.py 0 10 False
```

Start year

```
import sys

start=int(sys.argv[1])
end=int(sys.argv[2])
condition=sys.argv[3] #True/False
```

```
sys.argv[0] Note the zero argument is the name
of the script itself
```

On-click events

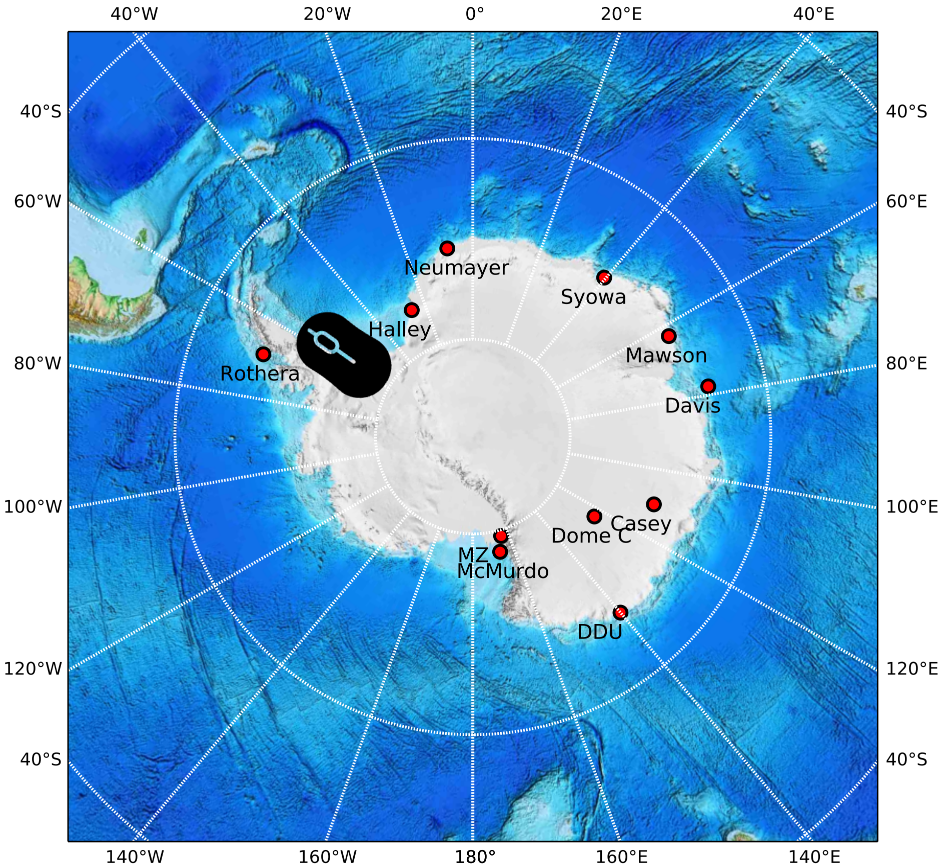
Relevant Documentation: https://matplotlib.org/stable/users/explain/figure/event_handling.html

You can create interactive maps/figures by defining within your code a series of ‘onclick events’.
Event type (a few examples):

One click	`event.button`
Double click	`event.dblclick`

Left Click	`event.button==1`
Middle Click	`event.button==2`
Right Click	`event.button==3`
Scroll Up	`event.button==4`
Scroll Down	`event.button==5`

On-click events



python3 interactive_map.py

single click: button=1, x=254, y=439, xdata=-2242070.049719, ydata=-5582140.158033

\\\
(lon,lat) on click: -68.4841152905 -67.1495205054

Nearest Station: Rothera (lon,lat) -68.57 -67.12

Processing data for Rothera
\\\

On-click events

```
################### Create an Interactive Map ######################


fig=plt.figure(figsize=(10,10),tight_layout=False)
ax=plt.gca()

#Plot map
bmap=Basemap(projection='spstere',boundinglat=-50,lon_0=180,resolution='l')
bmap.drawparallels()
bmap.drawmeridians()
bmap.etopo()

#convert site coordinates to map projection coordinates
data_in=np.loadtxt('station_coordinates.txt', skiprows=1, usecols=(1,2)) #skip header
lon, lat = data_in[:,0], data_in[:,1] # Location of station in degrees
#plot sites
x,y = bmap(lon,lat)
bmap.scatter(x, y, s=80, marker='o', color='red', linewidth=2, edgecolors='black')


…
```

On-click events

```
continued …

def onclick_map(event):
 print('%s click: button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          ('double' if event.dblclick else 'single', event.button,
           event.x, event.y, event.xdata, event.ydata))

 if event.button == 1: #if RIGHT CLICK: plot Brunt-Vaisala frequency N for model and obs

   #convert projection coordinates to geographical coordinates (these are the coordinates of our click)
   lon_cl,lat_cl = bmap(event.xdata,event.ydata, inverse=True)

   print '\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\'
   print '(lon,lat) on click:',  lon_cl,lat_cl

   #using the ‘click coordinates’ look for the closest station
   for i_site in range (0, len(lon)):
    if  (lon[i_site]-1)<= lon_cl <= (lon[i_site]+1):     #within +/- 1 degree lon of site location
     print 'Nearest Station:', labels[i_site], '(lon,lat)', lon[i_site], lat[i_site]

     #finally, call the function to plot model-observation comparisons
     process_data(lat[i_site],lon[i_site])
   print
'\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\’

cid = fig.canvas.mpl_connect('button_press_event', onclick_map)

plt.show()
```

Exercises

Using the provided data set (`SST_ERA5_monthly_1970_2022.nc`) compute the NINO 3.4 Index.
In order to do this, follow the instructions at page 3 and the examples at pages 4-7.
Remember, for the computation of the mean of your "reference period" use the first 30 years of data (i.e. 1970-2000).

ICTP