# Python/NCL: free alternatives for Earth Scientists

Nicolas Barrier

September 4, 2013

# Contents

# Chapter 1

# Introduction

The job of an Earth-Scientist mainly consists in processing observations, model outputs and many other types of data. The verb “process” here has plenty of meanings. It can be the computation of statistics, the computation of other parameters from the datas or, more simply, to represent the data in a simple and accessible way.

There exists a lot of programmation langage that allows this processing and that are well used in the Earth-Science community. The most commonly used is Matlab (MAtrix LABoratory), which is a proprietary numerical computing environment that can be used in a multitude of domain (engineering, science and economics). Free clones of Matlab exist, the two commons being Scilab and Octave.

Python is an open-source programming language which can be used in many fields, basically the same as matlab. Moreover, it is fully compatible with Matlab outputs (.mat files). Its free characteristic can make it very popular for students who cannot afford a Matlab license.

The last processing tool I will discuss is NCL (NCAR Command Language). Contrary to Matlab or Python, NCL has been principally developped for scientific data analysis and visualization in the Earth-Science fields. It is very usefull to process NetCDF files, which is a common file format in Earth-Science and is free.

The aim of this book is to put into perspective those three langages, to describe their pros and con. Main features of each language will be described, and classical operations in Earth-Science (especially statistics) will be described.

Finally, an extended gallery will be proposed, following that of Python or NCL, so that the most classical figures will be accessible whatever your choice between the three languages.

# Chapter 2

# Programming

## 2.1 Array manipulation

The basis in programming, and especially in Earth Science, is the array manipulation. While it is pretty straigthforward in NCL, it can be more complicated in Python, as you can manipulate either `list` or `numpy array`. We will review, in this section, all what must be known about array manipulation

### 2.1.1 Creation

Python:

In Python, if you want to create an array of known values, you can do:

```
array=[0,1,2,3,4,5]
```

If you do a `print(type(array))`, you will get `<typelist >`. This is not the easiest type to manipulate. It is strongly advised to convert them into numpy arrays as follows:

```
import numpy as np
nparray=np.array(array)
```

The numpy package allows many ways to initialize arrays. You can either initialize them with random numbers, zeros or ones everywhere. This is done by doing:

```
array1=np.empty((4,6,8),np.int)
array2=np.zeros((4,6,8),np.float)
array3=np.ones((4,6,8),np.double)
```

The first argument is the size of the array. It is set as a “tuple”. The second argument is ‡the format of the array. You must remember that as we create numpy array, we must call the types available in the numpy package (numpy.int, numpy.float, etc).

NCL:

In NCL, if you want to create an array of known values, you can do:

```
array=(/0,1,2,3,4,5/)
```

Moreover, if you want to initialiaze a multidimensional array, you must use the `new` command as follows:

```
array1=new((/4,6,8/),integer)
array2=new((/4,6,8/),float)
array3=new((/4,6,8/),double)
```

Contrary to Python, there is no function to initialize the array with zeros or ones.

### 2.1.2 Find values

In this section, we will learn how to find indices and values of nonzero elements. Imagine we have an array A in which we have zeros values. We want to find the indexes of those values.

Python:

In Python, this is done by using the `nonzero` function.

```
i=numpy.nonzero(A==0)
```

`i` is a tuple of dimensions the number of dimensions of A. If A is mono-dimensional, you can print the zero values as follows:

```
print(A[i[0]])
```

If A is multidimensional (for example three dimensions), it is done as follows:

```
print(A[i[0],i[1],i[2]])
```

NCL:

In NCL, it is done using the `ind` function:

```
i=ind(A.eq.0)
print(A(ind))
```

Contrary to Python `nonzero`, NCL `ind` function does not handle multidimensional arrays.

### 2.1.3 Element masking

It is often usefull to mask an array when some conditions are verified. Imagine that we have a map M with 0 at land points that we want to mask.

Python:

In Python, this is done with the `masked_where` function of the `numpy.ma` module.

```
import numpy.ma as ma
M_masked_where(M==0,M)
```

NCL:

In NCL, the command to use is the `mask` function.

```
M_mask=mask(M,M.eq.0,False)
```

The last argument is a boolean that, when set to `False`, indicates that the values to mask are those that verify the condition.

### 2.1.4 Elements extraction

In this section, we look at the way to extract data from an array. Imagine that we have an array with elements from one to ten. In NCL and Python (contrary to Matlab), the index starts at 0;

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Elements | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Python:

In Python, to extract data from an array, you must do as follows:

```
extract=array[3:7])
```

You would notice that when printing this new array, you will obtain in Python [4567], while the element of index 7 is 8. This points toward a very important difference between Python and NCL (and also Matlab): such an extraction stops one element before the end of the stride. Here, the last element extracted has the index 6.

Another way to extract the elements of an array is to create an array with integers corresponding to the index to extract:

```
index=np.array([3,4,5,6,7])
extract=array[index]
```

To extract data with a certain stride (ie. one data out of three for example):

```
extract=array[::stride]
```

NCL:

In NCL, it is almost the same thing than in Python except that we must use parenthesis instead of brackets

```
extract=array(3:7)
```

It should be noted that extract contains [45678], ie. the last element selected has the end index indicated. You can also use an array of index, similarly than Python:

```
index=(/3,4,5,6,7/)
extract=array(index)
```

A very interesting feature with NCL is that instead of using indexes, we can directly use litteral values. This is possible if the data coordinates are set. Imagine for example that we a map M of dimensions (nlat,nlon) of coordinates (longitudes,latitudes). To extract the data in the domain $10°S - 70°N$ and $10°W - 80°E$, you can directly use:

```
extract=M({-10:70},{-10:80})
```

instead of looking for the indexes corresponding to the domain.

The extraction of data with a certain stride is done in the same way than with Python:

```
extract=array(::stride)
```

### 2.1.5 Evenly spaced arrays

In this section, we will explain how to create evenly spaced arrays, which can be usefull for example to defined contours intervals for contour plots.

Python:

In Python, the creation of evenly spaced arrays is simple. You must use the numpy function `arange`.

```
array_int=np.arange(0,10,1)
array_float=np.arange(0,10,0.1)
```

Again here, Python stops one elements before the effective end of the array: the `array_int` vector will stop at 9, while the `array_float` one will stop at 0.9.

NCL:

In NCL, the way to create evenly spaced arrays is different wether we generate integer or floats.

For integer arrays, we must use the `ispan` function:

```
array_int=ispan(0,10,1)
```

The ispan function takes as integer the start, end and stride of the array. In this case, the last element of `array_int` will be 10, contrary to Python.For float arrays, it is the `fspan` function that must be used:

```
array_float=fspan(0,10,101)
```

The fspan also takes as arguments the start and end elements of the array. But the last option is not the stride anymore but an integer that is the number of points of the array.

### 2.1.6 Changing the shapes

Python:

To change the shape of an array in Python, you have at least two options. If you have a $N \times M \times T$ array $A$ that you want to set as a one-dimensional array, you must use the numpy function `ravel` :

```
oned=np.ravel(A)
```

If you want to change the shape of A from $3D$ to $2D$, you can use the `reshape` function:

```
twod=np.reshape(A,(N*M,T))
```

The reshape arguments are the matrix to reshape and the dimensions of the new array.

NCL:

To change the shape of an array in NCL, you also have two functions. To moves from a $3D$ array to a $2D$ array, you must use `ndtooned` :

```
oned=ndtooned(A)
```

In NCL contrarary to Python, there are no function to reshape A from $3D$ to $2D$, you must combine the `ndtooned`and `onedtond` functions, the latter being the function that allows to reshape from one-dimensional to multidimensional arrays:

```
oned=ndtooned(A)
twod=onedtond(oned,(/M*N,T/))
```

The `onedtond` arguments are the matrix to reshape and the dimensions of the array.

### 2.1.7 Dimension reordering

Imagine you have an array $A$ of dimensions (time,lat,lon). It might be usefull to be able to change the dimensions order to create an array of dimensions (lat,lon,time).

Python:

In Python, it is the numpy `transpose` function that allows this:

```
reordered=np.transpose(A,[1,2,0])
```

The transpose function takes as an imput the array of which the dimensions are reordered, and the array that indicates how to permute the dimensions. Here, the second dimension becomes the first one, the third dimension becomes the second one and the first one becomes the last one.

NCL:

In NCL, there is no reordering function. Reordering is done as follows:

```
reordered=A(lat|:,lon|:,time|:)
```

The reordering is done via the dimensions of the arrays. You thus must have arrays in which dimensions are set. To do this, you must follow the procedure described in section 3.1.3.

### 2.1.8 Array conforming

Oftenly, you must conform the shape of one array to the shape of another array. Imagine we have a series of maps A of dimensions (lat,lon,time) and a mask M of dimensions (lat,lon). You want to create a new mask of same dimensions than A.

Python:

In Python, you must use the command `tile` as follows:

```
MC=np.tile(M,(ntime,1,1))
MC=np.transpose(MC,[1,2,0])
```

The first line defines a new array in which the mask is repeated ntime times. Thus the array has a size of (ntime,nlat,nlon). The second line allows to reorder the dimensions. The Python `tile` function only allows to repeat an array along the first dimesions. Had we done:

```
MC=np.tile(M,(1,1,ntime))
```

the array would have a dimension of (1,nlat, ntime*nlon).

NCL:

In NCL, it is the conform function that allows such a manipulation:

```
MC=conform(A,M,(/0,1/))
```

The conform function takes as argument the reference array (A), the array to conform (M), and an integer array corresponding to the dimensions of A that are shared with M. Here, M and A have the same latitudes (index 0) and the same longitudes (index 1).

## 2.2 Loops

### 2.2.1 Do loops

Python:

In Python, there are several ways to perform do loops. Let $F$ be a numpy array of $n$ elements. If you want to display all the elements of $F$, you can do:

```
for index in range(0,n):
    print(F[index])
```

In this case, we go over all the indexes of F (`index` will have values of 0, 1, ..., $n-1$) and `F[index]` retrives the element in $F$ of index `index`. The loop starts with a `for`. The line, where the loop is started, must be finished by a `:`. You also must notice the indent before the `print`. This is a peculiarity of Python. The start of do loop is indicated by a right indent, while the end of the loop is indicated by a left indent. Often, when a code that contains imbricated loops does not raise an error but does not return what was expected, the error is to be searched into a bad indent at some point. Another peculiarity of Python is that when going over a loop, the `index` will stop one element before the end indicated. Here, we told the loop to stop at `n` but it will implicitly stops at `n-1`. This peculiarity was also pointed out in the description of arrays.

Finally, Python allows another way to go over the elements of a list. One can directly go over them by doing:

```
for element in F:
    print(element)
```

NCL:

In NCL, do loops are performed as follows:

```
do index=0,n-1
  print(F(index))
end do
```

The loop starts with a `do` and stops with a `end do`. Moreover, the index effectively stops at the end index indicated in the loop. We thus need to stop the loop one element before the end of the array.

### 2.2.2 While loops

While loops are done similarly than do loops.

Python:

```
index=0
while(index<=n-1):
    print(F[index])
    index=index+1
```

NCL:

```
index=0
do while(index.le.n-1)
  print(F(index))
  index=index+1
end do
```

## 2.3 If statements

Python:

If statements in Python are done as follows:

```
if(condition==0):
    print('0')
else:
    print('Else')
```

They start by a `if` and end by `:` . Again, all the lines within the if statement must be indented. Moreover, you have the `elif` statement which allows multiple condition checking.

```
if(condition==0):
    print('0')
elif(condition==1):
    print('1')
else:
    print('Else')
```

NCL:

In NCL, the simplest if statement is done similarly than with Python.

```
if(condition.eq.0)
  print("0")
else
  print("Else")
end if
```

However, NCL does not have “official” elif statement. However, it is possible to combine `else` and `if` statements as follows:

```
if(condition.eq.0)
  print("0")
else if  (condition.eq.1)
  print("1")
else
  print("Else")
end if
end if
```

Note however the need to add a `end if` any times we open a line with a `if`, which is likely to make the code unreadable.

## 2.4 Date processing

In NetCDF files, the time variable is often given as an integer, in which the units of the files as well as the calendar are indicated. In this section, we discuss how to transform them into usable arrays. Imagine that we have an array of integers that we call `T`, which gives us the number of days since days since the 1st of January, 1950.

## Python:

To convert `T` into a date instance, you first must import the `utime` function from the `netcdftime` package.

```python
from netcdftime import utime
```

Then you must call this function as follows:

```python
units='days since 1950-01-01 00:00:00'
calendar='gregorian'
cdftime = utime(units,calendar=calendar)
```

Then, you must use the `num2date` function to convert the array `T` into a datetime object:

```python
date=cdftime.num2date(T)
```

Finally, if you want to retrieve the years, months or days of the date object, you must perform as follows:

```python
year=[]
month=[]
day=[]
for d in date:
    year.append(d.year)
    month.append(d.month)
    day.append(d.day)
```

To do the inverse operation (ie. move from a `datetime` object to an integer array) you must obviously use the `date2num` function as follows:

```python
T=[]
for d in date:
    T.append(cdftime.date2num(d))
```

## NCL:

In NCL, the function that allows to do this is `ut_calendar` .

```
date=ut_calendar(T,0)
```

It takes as an input the time array `T`, in which the attributes `units` and `calendar` must be set. The last argument is an integer, which indicates the format of the output. If 0, the ouput will have the format:

| | |
|---|---|
| date(:,0) | years |
| date(:,1) | months |
| date(:,2) | days |
| date(:,3) | hours |
| date(:,4) | minutes |
| date(:,5) | seconds |

If 1 (or −1), the output will have the YYYYMM format (either as a float or as an integer depending of the sign of the integer). The option set to 2 add the days, 3 adds the hours while 4 returns a format `YYYY.fraction_of_year`. To do the inverse operation, it is the function `ut_inv_calendar` that must be used:

```
units="days since 1950-01-01 00:00:00"
calendar="gregorian"
option=1
option@calendar=calendar
T=ut_inv_calendar(year,month,day,hour,minute,second,units,option)
```

It should be noted that the calendar is set as an attribute to an integer `option`. Moreover, the years, months, days, hours and minutes options must be integers.

## 2.5 Mathematics

### 2.5.1 Simple arithmetic operations

Table 2.5.0 shows how simple arithmetic operations are performed on a the first dimension of an array M, using either Python or NCL.

| Operation | Python | NCL |
|---|---|---|
| Average | `numpy.mean(M,axis=0)` | `dim_avg_n(M,0)` |
| Standard deviation | `numpy.std(M,axis=0)` | `dim_std_n(M,0)` |
| Variance | `numpy.var(M,axis=0)` | `dim_variance_n(M,0)` |
| Sum | `numpy.sum(M,axis=0)` | `dim_sum_n(M,0)` |
| Cumulated sum | `numpy.cumsum(M,axis=0)` | `dim_cumsum_n(M,0)` |
| Product | `numpy.prod(M,axis=0)` | `dim_product_n(M,0)` |
| Cumulated product | `numpy.cumprod(M,axis=0)` | - |
| Absolute value | `numpy.abs(M)` | `abs(M)` |
| Standardization | - | `dim_standardize_n(M,0)` |
| Minimum | `numpy.min(M,axis=0)` | `dim_min_n(M,0)` |
| Maximum | `numpy.max(M,axis=0)` | `dim_max_n(M,0)` |

Table 2.5.0: Simple arithmetic operations

It should be noted that by default, Python flattens the array so that the output should be a numeric. To force the operation along a specific dimension, we must add the argument `axis`.

Same is done with NCL. The function that allows operations along a specific dimension end by `_n`. There exists equivalent functions that do not end like this, such as `dim_avg`. This computes the average over the last dimension.

*Note*: In NCL, simple arithmetic operations tend to erase metadata, such as attributes, coordinates, etc. To keep metadata, the functions ending by `_Wrap` should be used (for example, `dim_avg_n_Wrap(M,0)` computes the average over the first dimension and will keep the metada of M).

### 2.5.2 Correlations

If we want to compute the correlation coefficient between two time-series `A` and `B`.

Python:

In Python, you must use the function `corrcoef`:

```
C=numpy.corrcoef(A,B)
```

`C` will be a matrix of size $2 \times 2$, with $C(0,0) = C(1,1) = 1.0$ (the autocorrelation between A and A and between B and B). The correlation coefficient between A and B is $C(0,1) = C(1,0)$.

There are no Python functions to compute lead-lag correlations. The latter must be done by hand. The correlation coefficient is estimated by:

$$r_{AB} = \frac{\sum_{i=1}^{n}(A_i - \overline{A})(B_i - \overline{B})}{(n-1)S_A S_B}$$

where $n$ is the number of data in $A$ and $B$, $\overline{A}$ and $S_A$ are the mean and standard deviation of $A$, respectively.

Thus, lead lag correlations between $A$ and $B$ can be computed from the simple program:

```
def LeadLagCorr(A, B,nlags=10):
    coefs=numpy.empty(nlags+1)
    cpt=0
    for p in range(-(nlags/2),nlags/2,1):
        if p<0:
            Aint=A[:p]
            Bint=B[-p:]
        elif i==0:
            Aint=A
            Bint=B
        else:
            Aint=A[p:]
            Bint=B[:-p]

        n=len(Aint)
        sA=np.std(Aint)
        sB=np.std(Bint)
        mA=np.mean(Aint)
        mB=np.mean(Bint)

        r=np.sum((Aint-mA)*(Bint-mB))/(n-1*sA*sB)

        coefs[cpt]=r
        cpt=cpt+1

    return coefs
```

NCL:

In NCL, correlations are computed by the command:

```
C = escorc(A,B)
```

In this case, we obtain a scalar which effectively corresponds to the correlation coefficient between the two time-series.

Moreover, NCL permits to compute quiete easily lead-lag correlations by using the command `esccr`.

```
C = esccr(A,B,nlags)
```

In this case, C is an array of dimensions nlags+1 that computes the correlation coeffient between the two time-series at lags 0 to 10, with $A$ leading. However, negative lags are not handled. And to have the correlations when B leads, you must compute:

```
Cbis = esccr(B,A,nlags)
```

To concatenate the two results:

```
Cfinal = new(2*mxlag+1,float)
Cfinal(0:nlag-1)=Cbis(1:nlag-1)
Cfinal(mxlag:)=C(0:mxlag)
```

In this case, negative lags in Cfinal implies that B leads.

### 2.5.3 EOF analysis

Empirical Orthogonal Function (EOF) decomposition is often used in climate science. It indeed permits to easiyly assess the variability of a multidimensional field. Imagine we have a $3D$ field, M, of dimensions (time,depth,latitude). We want to decompose it into EOFs.

Python:

One way to simply compute EOFs is to use the `pyclimate` package (`http://starship.python.net/crew/jsaenz/pyclimate/`).

This is done as follows:

```
from pyclimate import svdeofs
(PCs, eigenvalues ,EOFs)=svdeofs.svdeofs(M, pcscaling=1)
```

The function `svdeofs` takes as an input the matrix M. The `pscaling` option allows to determine wether the PCs or the EOFs are normalized. Here we chose `pcscaling=1`, which implies that the PCs will be dimensionless while the EOFs will have the same units as M.

The output is a tuple, with `PCs` being the principal components of dimensions (time,eofs), `eigenvalues` being the eigenvalues of dimension (eofs) and `EOFs` being the spatial patterns of the EOFs of dimension (depth,latitude,eofs). To obtain the amount of variance explained by each EOFs, you must remember that:

$$Var(k) = \frac{\lambda_k}{\sum_{i=1}^{n} \lambda_i}$$

Thus the variance (in %) explained by each EOF can be determined by:

```
Var=100*eigenvalues/np.sum(eigenvalues)
```

NCL:

In NCL, the EOF patterns are easily obtained:

```
 deof=ntime
optEOF=True
optEOF@pcrit = 100
eof_map = eofunc(M(depth|:,latitude|:,time|:),deof,optEOF) ; Carte des eofs
```

The two first lines determine the option of the EOF decomposition. The `pcrit` option indicates the percentage of non-missing value (along the time dimension at each grid point) that must have the array so that EOF is computed. The `eofunc` takes as arguments the array M (with time as the last dimension), the number of EOFs to retain and the option boolean.

The output `eof_map` contains the normalized EOF patterns and has dimensions of (eofs,depth,latitude). It has the attributes `eval` (eigenvalues) and explained `var` (explained variance). To dimensionize the EOF maps, you must do:

```
eof_map_dim=eof_map
do beof=0,deof-1
  eof_map_dim(beof,:,:) = (/eof_map(beof,:,:)*sqrt(eof_map@eval(beof))/)
end do
```

Finally, to obtain the principal components:

```
eof_pc= eofunc_ts_Wrap(fld,eof_map,False)
eof_pc_dim=dim_standardize_Wrap(eof_pc,0)
```

The first line determines the principal components with the same units than $M$. The second line allows to determine the normalized principal components.

# Chapter 3

# Input/Output (IO)

## 3.1 NetCDF

### 3.1.1 What is a NetCDF?

NetCDF stands for **Net**work **C**ommon **D**ata **F**orm. It is a portable, auto-documented data format and is common among Earth-Scientists. Many data-sets are available in this format in this format. Moreover, many models output are also returned in NetCDF. Thus, it is paramount that any Earth Scientist should be able to work with this format. This is the goal of this section. But beforehand, the architecture of a NetCDF file needs to be described.

In a NetCDF file, you will find variables, dimensions, coordinates and attributes. The variable is the information you want to extract of the file. The dimensions obviously returns the name of the dimensions of the variables as well as the number of data along each of them. The coordinates are monotonic one dimensional arrays, each corresponding to one dimension, which returns the values of each dimension points. The attributes can be viewed as informations that can be attached to the variables or to the file itself. Those attributes can be of any types (string, arrays, etc).

Imagine we have a NetCDF file, the variable of which is a succession of daily global maps of one degree resolution and for a complete year. We call this variable *SST*. The dimensions of *SST* are (*time*,*latitude*,*longitude*), with *time* having 365 elements, *latitude* having 181 elements, and *longitude* having 361 elements. The coordinates of *SST* will be the sucession of time steps, the vector of latitudes and the vector of latitudes. Usually, the names of the coordinates and of the dimensions are identical. Moreover, it should be pointed out that in the NetCDF file, the coordinates will appear as additional variables.

This is this file that we will try to read and write with our programming tools.

### 3.1.2 Reading a NetCDF

The common features in reading a NetCDF is that you have to know the names of the fields you are trying to read. An easy way to obtain those informations is to use the following command: **ncdump -h My__NetCDF.nc**.

According to the definition of *SST*, we should obtain the following result:

```
netcdf My_NetCDF {
dimensions:
        time = UNLIMITED ; // (365 currently)
        latitude = 181 ;
        longitude = 361 ;
variables:
        float sst(time, latitude, longitude) ;
                sst:Name = "Sea-Surface Temperature" ;
                sst:_FillValue = 9.96921e+36f ;
        int time(time) ;
        int latitude(latitude) ;
        int longitude(longitude) ;

// global attributes:
                :Description = "Example of NetCDF File" ;
}
```

The number of dimensions of time is UNLIMITED, which means that this dimension is the record dimension. This means that other data of similar latitude and longitude dimensions can be concatenated along the time dimensions. We will see that this characteristic allows multiple files opening.

Python:

In Python, you have many librairies which allow to read NetCDF files. We will only described one of them, which, according to us, allows the most capabilities. This librairy is netCDF4.

```python
import netCDF4 # Import the netCDF4 Module
f=netCDF4.Dataset('My_NetCDF.nc','r') # Read the NetCDF file
field=f.variables['sst'] # Read the variables
time=f.variables['time']
lat=f.variables['latitude']
lon=f.variables['longitude']
name=field.Name # Retrieve the variable attribute
des=f.Description # Retrieve the file attribute
field=field[:] # Convert the data into numpy arrays
time=time[:]
lat=lat[:]
lon=lon[:]
```

NCL:

NCL, which has been developped for Earth Science, is the most efficient among the three in the processing of NetCDF. To open a file, you simply use the command **addfile**

```
f = addfile("My_NetCDF.nc","r") ; Read the NetCDF file
field=f->sst ; Read the variables
time=field&time ; Read the coordinates
lat=field&latitude ; Another possibility
lon=field&longitude ; lat=f->latitude
name=field@Name ; Retrieve the variable attribute
des=f@Description ; Retrive the file attribute
```

### 3.1.3 Writting a NetCDF

Python:

```python

from netCDF4 import Dataset # Load the librairies
import numpy as np #
import os

# Generate data arrays
sst=np.zeros((365,181,361),np.float)
```

```
lon=np.arange(-180,181)
lat=np.arange(-90,91)
days=np.arange(1,366,1)

#Compute dimension size
nlon=len(lon)
nlat=len(lat)

os.system('rm '+'My_NetCDF.nc') # Remove the output file
fout=Dataset('My_NetCDF.nc','w') # Create the output file
fout.Description='Example of NetCDF File' # Write the file attribute

# Create the dimensions of the files
fout.createDimension('longitude',nlon)
fout.createDimension('latitude',nlat)
fout.createDimension('time',None)

#Create the variables of the files
varout=fout.createVariable('sst','f',('time','latitude','longitude'))
varout.Name="Sea-Surface Temperature" # Attach the variable attribute
varout=fout.createVariable('latitude','i',('latitude',)) # Create the
coordinates variable
varout=fout.createVariable('longitude','i',('longitude',))
varout=fout.createVariable('time','i',('time',))

# Write out the data arrays into the file
fout.variables['sst'][:] = sst
fout.variables['latitude'][:] = lat
fout.variables['longitude'][:] = lon
fout.variables['time'][:] = days

fout.close() # Close the file
```

NCL:

In NCL, the writting is as easily as the reading of a NetCDF.

```
begin

; Generate data arrays
sst=new((/365,181,361/),float)
lon=ispan(-180,180,1)
lat=ispan(-90,90,1)
days=ispan(1,365,1)

; Define the dimensions of the variable
sst!0="time"
sst!1="'latitude"
sst!2="'longitude"

; Attach the coordinates to the dimensions
sst&time=days
sst&longitude=lon
sst&latitude=lat

;Compute dimension size
nlon=dimsizes(lon)
nlat=dimsizes(lat)

; Attach the variable attribute
sst@Name="Sea-Surface Temperature"

system("rm My_NetCDF.nc") ; Remove the file
fout=addfile("My_NetCDF.nc","c") ; Create the output file
fout@Description="Example of NetCDF File"

filedimdef(fout,(/"time","latitude","longitude"/),(/-1,nlat,nlon/),(/True,False,False/))
; Set the time dimension as the record dimension
```

```
fout->sst=sst  ; Write out the data arrays into the file

end
```

### 3.1.4 Multiple file opening

Python:

It can happen that a variable is stored in many files (for example, one file for each time step). It can thus be usefull to know how to use multiple file opening. Sadly, Matlab does not allow such operations. In python, the procedure is very similar to single file opening but with the commande MFDaset. It takes as in input a string which defines where to find the find (it basically is the string you will use in a linux "ls" query).

```
from netCDF4 import MFDataset
f=MFDataset('sst_*')
sst_aggregated=f.variables['sst'][:]
lat=f.variables['latitude'][:]
```

NCL:

In NCL, you have to use the command systemfunc to obtain the names of the file to open. The results of this command will be used in the addfiles command.

```
filenames=systemfunc("ls sst_*")
f=addfiles(filenames,"r")
sst_aggregated=f[:]->sst
lat=f[0]->latitude
```

## 3.2 Text files

Imagine that you want to read an ascii file with the following format:

```
Maximum    13.58449     Sv latitude    45.81936     depth =    1061.401
Minimum   -1.709398     Sv latitude    45.81936     depth =    3530.419
Maximum    12.94729     Sv latitude    45.81936     depth =    1061.401
Minimum   -1.989103     Sv latitude    45.81936     depth =    3530.419
```

This the output of a CDFTOOLS which compute the maximum overturning streamfunction. You want to retrive the values of the maximum, which are in this case 13.58449, 12.94729, etc.

Python:

In Python, the script that will allow you to to this is:

```
import numpy as np # Import numpy

file=open('REF_Yearly_maxovt.txt','r') # Read the file
lines=file.readlines() # Read the file lines by lines
nlines=len(lines) # Read the number of lines

maxmoc=[] # Initialize the output
```

```

delim='    ' # String delimiter

for l in lines: # Run the different lines
    s=l.split(delim) # Split the line according to the delimiter
    if(s[0].strip()=='Maximum'): # Test wether the first string is Maximum
        maxmoc.append(float(s[1])) # Add the output to the list

maxmoc=np.array(maxmoc) # Convert the list into an array
```

The command that allows to open the file is simply open, while to read the file line by line, it is readlines. The output is an array, the elements of which are the string lines. The next step is to run all the lines (for loop) and to split the string according to the delimiter delim. If you print s, you will obtain:

```
['  Maximum', ' 13.58449', '', 'Sv latitude', ' 45.81936', '', 'depth =', ' 1061.401', ' \n']
```

The if condition check wether we are reading a line starting with 'Maximum'. The strip function allows to get rid of spurious spacing at the beginning of the string (due to the fact that the lines starts with a spacing). Then the elements is converted into a float and added to the list maxmoc. As list cannot be used with numpy, we finally convert it into a numpy array.

This method assumes that the file is not corrupted (ie. no missing data, all the column have the same shape, etc). However, in some cases (for example a CSV file of GPS trames), the file might have some incomplete lines. In this case, our simple script will not work.

In Python, there is a package that can handle this. This is the **re** (for Regular Experessions) package. Imagine we slightly modify our text file:

```
  Maximum    13.58449      Sv latitude    45.81936      depth =    1061.401
  Minimum   -1.709398      Sv latitude    45.81936      depth =    3530.419
  Maximum    0       Sv latitude    45.81936      depth =    1061.401
  Minimum   -1.989103      Sv latitude    45.81936      depth =    3530.419
  Maximum    13.97433      Sv latitude    45.81936      depth =    1061.401
  Minimum   -1.899664      Sv latitude    45.81936      depth =    3530.419
  Maximum    13.42229      Sv latitude    45.81936      depth =    1061.401
  Minimum   -1.606414      Sv latitude    45.81936      depth =    3530.419
```

You see in line 3 that something went wrong, as the data is 0. When reading the file, we are only interested in good data. To do so, you must use the re package as follows:

```
import re # Importation of the re package
import numpy as np

file=open('REF_Yearly_maxovt_REGEXP.txt','r')
lines=file.readlines()
nlines=len(lines)

pattern=' *Maximum *[1-9]{2}\.[0-9]{5}' # Definition of the pattern to match
regexp=re.compile(pattern) # Compilation of the pattern

maxmoc=[]

for l in lines:
    if(regexp.search(l)): # Test wether the line matches or not.
       print('We have a good line here')
       s=l.split('    ')
       maxmoc.append(float(s[1]))
```

The pattern string is the so-called regular expression, which is a pattern that must be matched. The ' *Maximum *' indicates that we can have 0 or any number of white spaces before and after the 'Maximum' string. The `'[0-9]{2}\.[0-9]{5}'` means that after the white spaces, we must have two digits from 0 to 9, a point .[1], and 5 digits from 0 to 9 after the point. This is the format of the good data (for example 13.58449). We must then compile this pattern using the compile function. Now, similarly with the previous example, we run the lines and compare it with the pattern, using the search command. If this pattern is found in the data, then they will be added in the list. As 0 does not match this pattern , it will be discarded.

Regular expressions also permit to split a string with a delimiter matching a pattern. In our textfiles, we have spacing between the column which is irregular. But this spacing, between the data, is greater than 3. We can thus do the following:

```
pattern=' {3,}'
regexp=re.compile(pattern)

for l in lines:
    s=regexp.split(l)
    print(s)
```

Note that the syntax of the split here is different from that of the first example. The ' 3,' indicates that the splitting must be done for spacing from 3 to infinity. If you print s, you will have:

```
['  Maximum', '13.58449', 'Sv latitude', '45.81936', 'depth =', '1061.401', '\n']
```

Compared to the first example, we notice that single spaces (column 3 and 6) have disappeared.

NCL:

Here is the NCL script that will allow this:

```
  delim=" " ; definition of the string delimiter
  finname="REF_Yearly_maxovt.txt" ; name of the file to read

  lines = asciiread(finname,-1,"string") ; Read the file per lines
  nfields = str_fields_count(lines(1), delim) ; Number of column of
  the file
  nlines=dimsizes(lines) ; Number of lines

  maxmocint=new(nlines,float) ; Initialize the output

  cpt=0 ; Counter to increment
  do indl=0,nlines-1 ; Line iteration
    test=str_get_field(lines(indl),1,delim) ; Retrieve the first string
    if(test.eq."Maximum") ; Test wether it is maximum
      maxmocint(cpt)=stringtofloat(str_get_field(lines(indl),2,delim)) ; Retrive the
          maximum value, convert it into float
      cpt=cpt+1 ; Counter incrementation
    end if
  end do

  maxmoc=maxmocint(ind(.not.ismissing(maxmocint))) ; Remove all missing values
```

[1]As the point is considered as a special character for regular expressions, we must add a backslash before the point

The asciiread command will read the file line by line, and returns an array of string, the elements of which are the string lines. The str_fields_count function counts the number of columns that are separated by the delimiter. nlines is the total number of lines in the file, including the lines we are not interested in. We also initialize the vector in which the values will be stored. However, we initialize it with too many dimensions. The cpt variable is a counter wich will be incremented every time we write an element in the array. The command to retrive a value of a SVD line is str_get_field, which takes as arguments the input string, the position of the element we want to extract, and the delimiter used in the separation. The test variable is used to test wether the line we read is that of the maximum overturning. If it is, we extract the values and convert it into a float before adding to the array maxmocint. The cpt variable is then incremented.

At the end, we have a big array with, at the beginning, the values we are interested in, and at the end missing values. The last step, using the ismissing function, is to get rid of those spurious missing values.

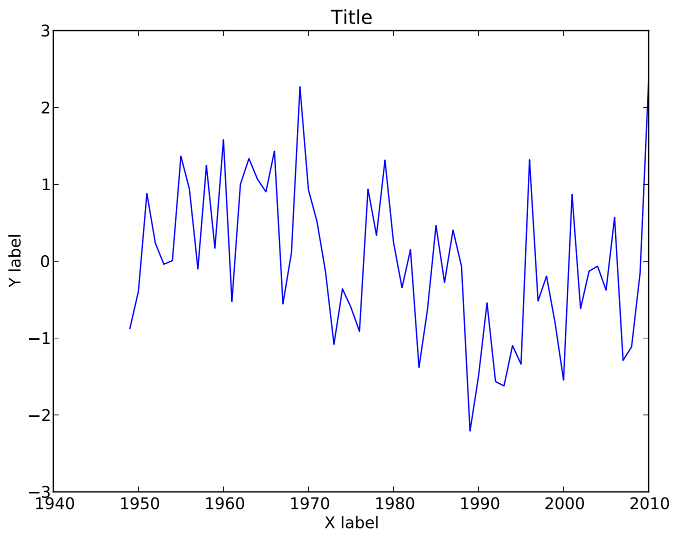
As you may have noticed, we have less flexibility with NCL than with Python for reading ASCII files.

# Chapter 4

# Plots

## 4.1 XY Plots

### 4.1.1 First plot



Python:

```
from pylab import * # import all pylab functions
import numpy as np # import numpy library

figure() # define a figure instance
plot(years,indexes[0,:]) # Plot function
xlabel('X label') # definition of xlabel
ylabel('Y label') # definition of ylabel
title('Title') # definition of title
savefig('fig/plot_ts_1_py.pdf', bbox_inches='tight') # save figure
```

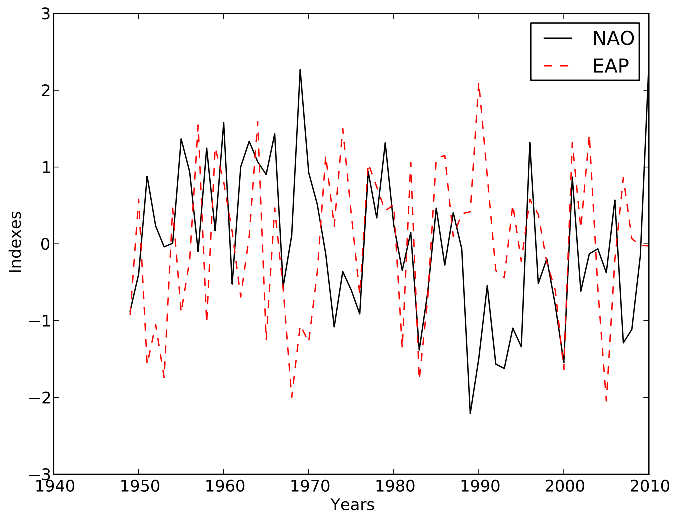
In Python, the *pylab* library handles all the functions necessary to draw figures. First, we define the *figure* instance. The time-series are plotted with the *plot* function. As no axes instance has been defined, it implicitely creates one. The x-axis tile, y-axis title and figure title are given by *xlabel*, *ylabel* and *title*, respectively. The figure saving is done with *savefig*. The *tight* option allows to get rid of white space around the figure.

NCL:

```
; Import NCL functions
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_csm.ncl"
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/contributed.ncl"
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/shea_util.ncl"

begin
  wks    = gsn_open_wks (''pdf'',''fig/''+get_script_prefix_name()+''_ncl'') ; open
      workstation

  res                    = True ; Initialize ressource
  res@tiXAxisString        = "X Label" ; definition of xlabel
  res@tiYAxisString        = "Y Label" ; definition of ylabel
  res@tiMainString        = "Title" ; definition of title

  plot  = gsn_csm_xy (wks,years,indexes(0,:),res) ; Plot function
end
```

In NCL, it is often usefull to load the *csm* scripts beforehand. The *gsn_open_wks* command allows to open the *workspace*, which in this case is a .pdf file. Now we initialize the resources of the plot by setting *tiXAxisString*, *tiXAxisString* and *tiMainString*, which are x-axis tile, y-axis title and figure title, respectively.

### 4.1.2 Multiline plots



Python:

If you want to superpose two time series with Python, you have two ways. Either you define a $N \times 2$ vector, and the *plot* function described above will work. But I would advise the following.

```
l1=plot(years,indexes[0,:],color='k',linestyle='solid') # draw the first line, black and
     solid
l2=plot(years,indexes[1,:],color='r',linestyle='dashed') # draw the second line, red and
     solid
```

These two lines define two *Lines2D* instances. The *color* and *linestyle* options handle the color and linestyles of each individual lines (dashed, solid, dotted, etc).

To add a legend, you must use:

```
prop = matplotlib.font_manager.FontProperties(size=9) # Define the legend fontsize
legend((l1[0],l2[0]),('NAO','EAP'),loc=0,prop=prop) # Add the legend: l1->NAO, l2->EAP,
    localisation=best
```

The *matplotlib.font_manager.FontProperties* handle the font properties of the legend strings. The *legend* function draws the legend. It takes as an input two tuples: the individual lines of the plot (not the use of brackets, which is necessary). We added the *loc* property, which handles the location of the legend (0 means best position) and the FontProperties instance prop.

NCL:

In NCL, two-line plots need the definition of a $2 \times N$ array (note the difference with Python). The colors and dashed patterns of each line are defined in an array:

```
res@xyLineColors     = (/"black","red"/) ; Line colors vector
res@xyDashPatterns = (/0,2/) ; Line patterns vector: 0->solid , 2->dashed
```

The handling of the legend is much complicated than with Python.

```
res@pmLegendDisplayMode = "Always" ; Force the legend display
res@xyExplicitLegendLabels = (/"  NAO ","  EAP "/) ; Legend vectors
res@pmLegendWidthF          = 0.12 ; Legend width
res@pmLegendHeightF         = 0.07 ; Legend
res@pmLegendParallelPosF    = .85  ; Move legend along xaxis
res@pmLegendOrthogonalPosF  = -1.14 ;  Move legend along yaxis
```

The pmLegendDisplayMode resource forces the legend drawing. xyExplicitLegendLabels defines the lines names. pmLegendWidthF and pmLegendHeightF handle the width and height of the legend box, respectively. Finally, the last two resources handle the parallel (along X) and orthogonal (along Y) positions of the legend. It is the setting of these four resources that makes difficult legend handling with NCL. To finally draw the time series:

```
plot  = gsn_csm_xy (wks,years,indexes(0:1,:),res)
```

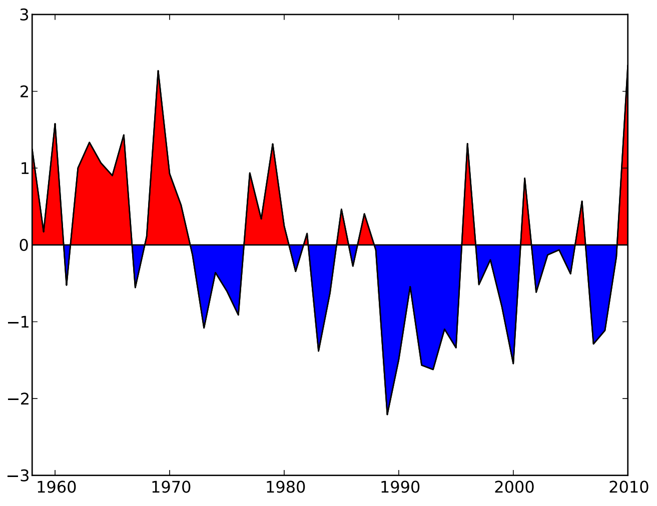
### 4.1.3 Filled XY plots

Python:

To draw filled plots, like NAO index as found in the internet:

```
fill_between(years,0,indexes[0,:],where=indexes[0,:]>=0,facecolor='red',interpolate=True
    ) # fill in red positive values
fill_between(years,0,indexes[0,:],where=indexes[0,:]<0,facecolor='blue',interpolate=True
    )
# fill in blue negative values
plot(years,indexes[0,:],color='k') # Overlay the line plot
```

The fill_between function fills the space between two time series. In this example, one of the time series is set to 0. The where option defines the condition that must be verified for the



drawing. Here, the first line indicates that the drawing, with the filling in red, must be done only where the index is positive. The second one, with blue filling, is only done if the index is negative. Note the use of the option interpolate set to True, to have a nice rendering.

In this example, we also resctricted the xaxis and yaxis limits:

```
xlim(1958,2010) # Limitation of xaxis
ylim(-3,3) # Limitation of yaxis
```

NCL:

In NCL, it is the same function that is used for filled plots. The only difference is the addition of three resources:

```
  res@gsnYRefLine           = 0.0  ; Set the Y reference line
  res@gsnAboveYRefLineColor = "red" ; fill in red positive values
  res@gsnBelowYRefLineColor = "blue" ; fill in red positive values
```

The first one defines the ordinate of the reference line, in our case the absic axis. The last two define the above and below fill colors, respectively.

To limit the plot domain, the resources that must be set are:

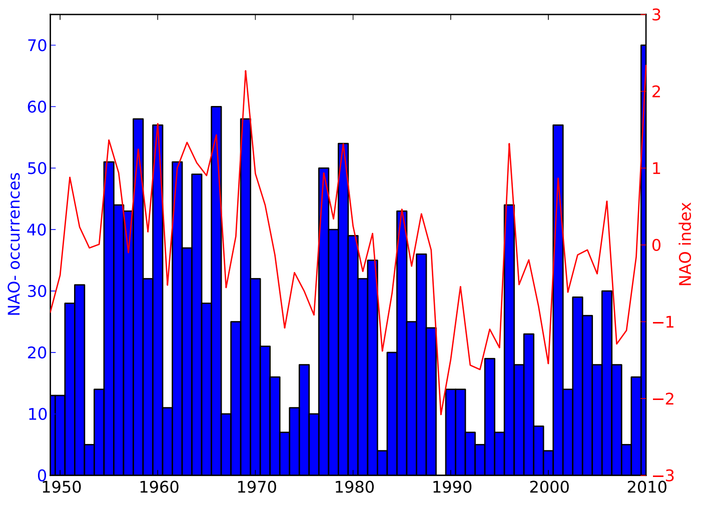
```
  res@trYMinF  = -3.0 ; min value on y-axis
  res@trYMaxF  =  3.0 ; max value on y-axis
  res@trXMinF  = 1949 ; min value on y-axis
  res@trXMaxF  =  2010 ; max value on y-axis
```

### 4.1.4 Twin xaxis

Python:

To draw two time-series with different units (in this case, a number of days and a dimensionless index, you have to use the command twinx. First, we set-up the bar histogramm.

```
ax=gca() # Initialize axes instance
bar(years-0.5,occu[3,:],width=1,bottom=0,color='b') # Plot histogram on 'ax'
ax.set_ylabel('NAO- occurrences', color='b') # Set 'ax' ylabel
```



```
setp(ax.get_yticklabels(),color='b') # Set the color of ticklabels
setp(ax.yaxis.get_ticklines(),color='b') # Set the color of ticklines
```

To draw histogramms, it is the command bar, and not plot, that is to use. It takes as an input the leftmost coordinates of each bar, the width, the bottom coordinate of the bar (in our case, 0). As the bars are blue, we also change the color of this axes ylabel, yticklabels and yticklines. This is done with the setp function, which takes as an input the instance of which we are willing to change the property, and this property.

To define the second axes, used for the line, we use

```
ax2=ax.twinx() # Initialize the second axes instance
```

Then, the plotting is done in the exact same way as previously. It has to be noted that to limit the xaxis limit of this figure, you have to do it on ax2, not ax.

NCL:

To do this figure with NCL, you have to define two ressources: one for the histogramm (resB), one for the line. As it is basically twice the same thing, we will only describe the resB ones.

```
  resB=True ; Initialise the ressource for the histogram
  res@trYMinF=0 ; Yaxis minimum
  resB@trYMaxF=75 ; Yaxis maximum
  resB@trXMinF=1949-0.5 ; Xaxis minimum
  resB@trXMaxF=2010+0.5 ; Xaxis maximum
  resB@tiYAxisString="NAO- occurrences" ; YAxis label
  resB@gsnXYBarChart = True ; Histogram mode on
  resB@gsnYRefLine=0 ; Define the Y reference line
  resB@gsnAboveYRefLineColor="blue" ; Blue fill above 0
  resB@tiYAxisFontColor="blue" ; Y title color
  resB@tmYLLabelFontColor="blue" ; Y Left labels colors
  resB@tmYLMajorLineColor="blue" ; Y Left Major ticks colors
  resB@tmYLMinorLineColor="blue" ; Y Left Minor ticks colors
```

The gsnXYBarChart is set to true to obtain a histogramm. However, this one will not be filled. We have to use gsnYRefLine and gsnAboveYRefLineColor, as previously, to fill the bars.

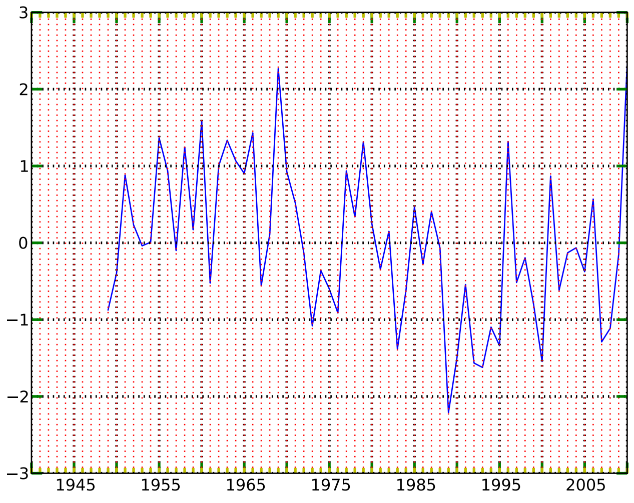
Now, to change the colors of the leftmost ticklabels (YL:YLeft, opposed to YR: YRight), we must set tmYLLabelFontColor, while to change the ticklines color, we must define tmYLMajorLineColor and tmYLMinorLineColor for major and minor ticks, respectively.

To overlay both figures, we must use the command:

```
    plot=gsn_csm_xy2(wks,years,occu(3,:),indexes(0,:),resT,resL) ; Plot function
```

It takes as an input the workspace, the x coordinates, the first y coordinates, the second y coordinates, and the resources of both plots.

### 4.1.5 Grid/labels handling



Python:

To handle tickmarks, we have to import MultipleLocator from the matplotlib library.

```
from matplotlib.ticker import MultipleLocator # Tick library

majorLocator   = MultipleLocator(5) # Major locator at multiple of 5
ax.xaxis.set_major_locator(majorLocator) # Define xticks major locator

minorLocator   = MultipleLocator(1) # Minor locator at multiple of 1
ax.xaxis.set_minor_locator(minorLocator) # Define xticks minor locator
```

This ensembple of function allows to draw, in the xaxis, a major tickmark at each multiple of 5, and a minor one at each coordinate (multiple of 1).

To add the grid:

```
grid(True,which='major',color='k',linewidth=2) # Draw grid at major locators
grid(True,which='minor',color='r',linewidth=1) # Draw grid at minor
locators
```

The which option allows to define whereas the grid is to be drawn on major or major ticks. It should be noted that Python automatically adds a label at each major tickmarks. If you want to draw a ticklabel out of 2:

```
for t in ax.get_xticklabels()[::2]: # hide one label out of 2
    setp(t,visible=False)
```

Finally, if you want to handle the tick properties, such as the color:

```
tick_params(which='major',color='g',width=2,length=8) # major tick lines properties
tick_params(which='minor', color='y',width=2,length=4) # minor tick lines properties
```

NCL:

In NCL, to do similar things only needs to set many resources:

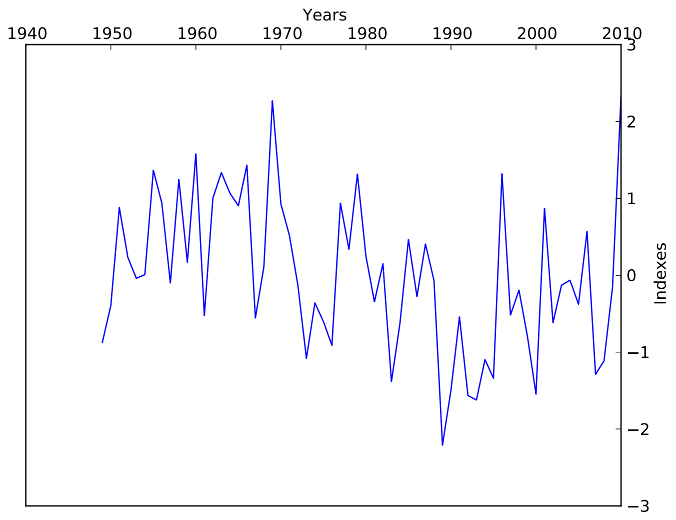
```

 res@tmXBMode="Manual" ; Draw ticks manually

 ; X Bottom Tick handling resources
 res@tmXBTickStartF=1945 ; XBottom tick start
 res@tmXBTickSpacingF =5 ; Xbottom tick spacing
 res@tmXBTickEndF=2010 ; XBottom tick end
 res@tmXBMinorPerMajor=4 ; 4 minor ticks for one major
 res@tmXBLabelStride=2 ; # hide one label out of 2
 res@tmXBMajorLineColor="green"
 res@tmXBMinorLineColor="orange"

 ; X Major Grid handling resources
 res@tmXMajorGrid=True ; Draw X major grid
 res@tmXMajorGridLineColor="black" ; X Major grid color
 res@tmXMajorGridLineDashPattern=2 ; X Major grid in dashed
 res@tmXMajorGridThicknessF=2 ; X Major grid thickness

 ; X Minor Grid handling resources
 res@tmXMinorGrid=True ; Draw X minor grid
 res@tmXMinorGridLineColor="red" ; X Minor grid color
 res@tmXMinorGridLineDashPattern=2 ; X Minor grid in dashed
 res@tmXMinorGridThicknessF=1 ; X Major grid thickness

 ; Y Left Tick handling resources
 res@tmYLMajorLineColor="green"
 res@tmYLMinorLineColor="orange"

 ; Y Major Grid handling resources
 res@tmYMajorGrid=True
 res@tmYMajorGridLineDashPattern=2
 res@tmYMajorGridThicknessF=2

 ; Similar tick properties in the top Xaxis and the right Yaxis
 res@tmYUseLeft=True
 res@tmXUseBottom=True
```

I do not go into the details, as the resource names are rather obvious. The most important are the last two ones. They force the XT (XTop) tick properties (colors, etc) to be equal to the XB (XBottom) ones, and the YR (YRight) tick properties to be equal to the YL (YLeft).

### 4.1.6 Labels positioning

Python:

To handle label and ticklabels position in python:



```
ax.yaxis.set_label_position('right') # YLabel in the right
ax.yaxis.set_ticks_position('right') # YTickLabels in the right

ax.xaxis.set_label_position('top') # XLabel at the top
ax.xaxis.set_ticks_position('top') # XTickLabels at the top
```

NCL:

To do the same thing in NCL,:

```
    res@tmYLLabelsOn=False ; Hide YLeft TickLabels
    res@tmYRLabelsOn=True ; Force YRight TickLabels
    res@tmXBLabelsOn=False ; Hide XBottom TickLabels
    res@tmXTLabelsOn=True ; Force XTop TickLabels

    res@tiXAxisSide="Top" ; Position of XLabel
    res@tiYAxisSide="Right" ; Position of YLabel
```

The first four resources hide the YLLabels/XBLabels, and to force the other labels. The last two change the positions of the axes labels.

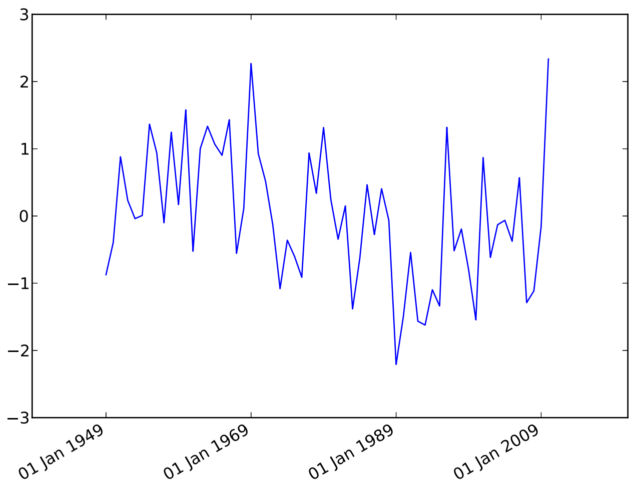
### 4.1.7 Time ticking

Python:

Oftenly, time in netcdf times is an integer, which gives the number of days, hours or second since a specific date. To nicely label your time axes, you first have to convert it into a datetime object (see previous section):

```
units='days since 0049-09-01 00:00:00'
calendar='gregorian'
cdftime = utime(units,calendar=calendar)
date=cdftime.num2date(time)
```

The next step is to generate the string array that will be used for the labelling:



```
stF=[]
for p in range(0,len(date)):
    stF.append(date[p].strftime('%d %b %Y'))
stF=np.array(stF)
```

The strftime command takes as an option a string formatter. Here, the format '%d %b %Y' stands for days as an integer, months as an integer and years as a 4 digit integer.

Then, you plot the index as a function of the time variable:

```
fig=figure()
ax=gca()
plot(numf,indexes)
```

So far, the xaxes will have the labels corresponging to the time. To add the labels corresponding to the strings previously generated:

```
stride=20
ticks=ax.set_xticks(numf[::stride])
ax.set_xticklabels(stF[::stride])
```

This will draw the labels with a stride of 20. Those labels will appear horizontally. To rotate them in a nice way, you must use the following command:

```
fig.autofmt_xdate()
```

NCL:

To do the same thing in NCL, you have to add the following ressources:

```
 stride=20
res@tmXBMode = "Explicit"
res@tmXBValues = time(::stride)
res@tmXBLabels = ut_string(time(::stride),"%d %f %Y")
```

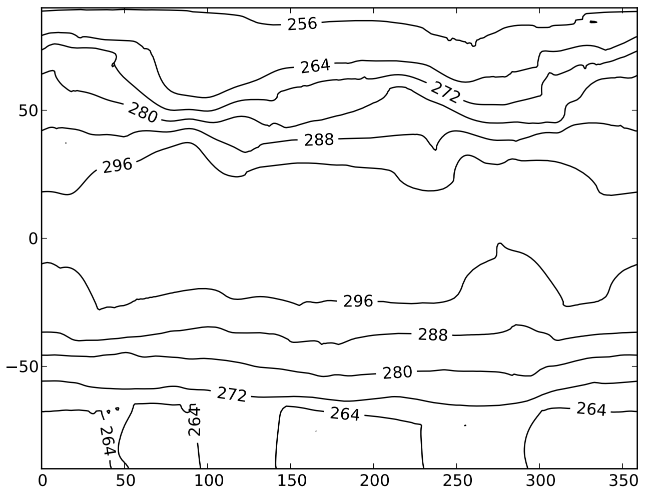
This adds the labels in text formatting but horizontally. To rotate them similarly as described, you must add the following lines:

```
    res@tmXBLabelAngleF=30
    res@tmXBLabelJust="TopRight
```

The first one rotates the labels, while the second one shifts the label to the left so that its end coincides with the ticks locations.

## 4.2 Contours

### 4.2.1 First contour plot



Python:

To draw our first contour, we have a one dimensional array for x coordinates, a one dimensional array for y coordinates and a two dimensional array of temperature. To draw a simple contour plot:

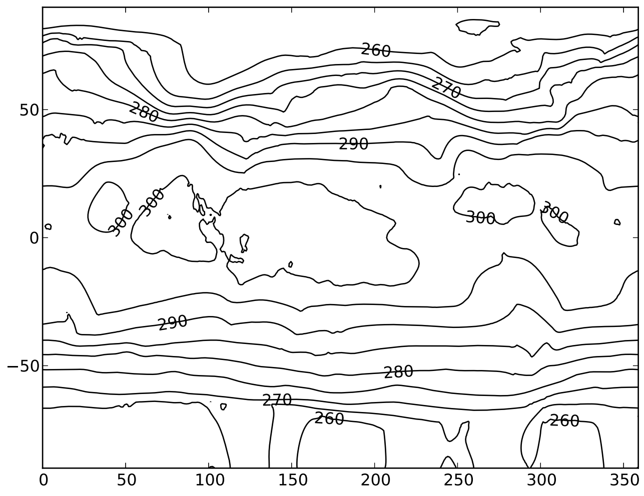
```
figure()
cs=contour(lon,lat,t2,colors='k') # Contour plots (not filled)
clabel(cs,fmt='%d') # Add labels to the lines as integer
```

NCL:

In NCL, to draw a contour plot, you first have to define the ressources that will contain the coordinates of the contour plot:

```
    res=True
    res@sfXArray = lon ; X array
    res@sfYArray = lat ; Y array
    res@cnInfoLabelOn=False ; Hide the informations on the contour
    plot=gsn_csm_contour(wks,t2,res) ; Contour command
```

The cnInfoLabelOn resource allows to hide the information of the contour plot (contour from 256 to 300 by step of 4). that is automatically added at the bottom of the plot.



### 4.2.2 Levels handling

Python:

It might be usefull to define the levels that must be drawn. In Python, you can do that with the option levels of the contour plots commands"

```
levels=np.arange(250,305,5)
cs=contour(lon,lat,t2,levels=levels,colors='k')
```

Moreover, in the previous section, we notice that the contours are masked under the labels. To decactivate this option, you have to set the option inline in the clabel function:

```
clabel(cs,levels[::2],inline=False,fmt='%d')
```

We notice that the second argument to the clabel function has been added. It corresponds to the array of levels at which the labels must be drawn.

NCL:

In NCL, the selection of contour levels to be drawn is done by setting:

```
levels=ispan(250,300,5)
res@cnLevelSelectionMode="ExplicitLevels"
res@cnLevels=levels
```

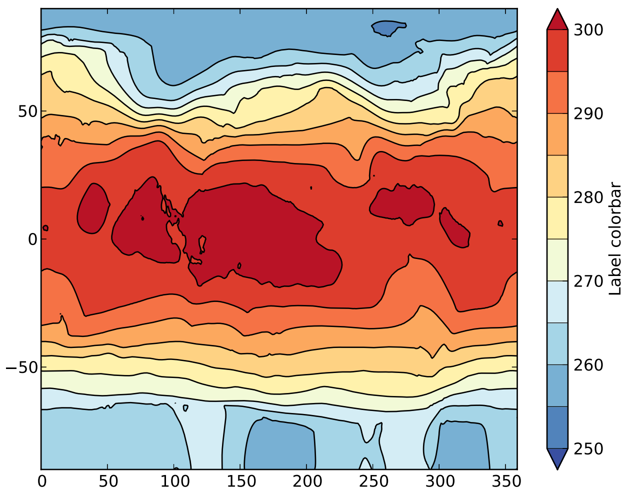
To add the labels to specific levels only, it is slightly more complicated than in Python. The first step is to define an array of strings, that we call flags. The second one is to set the cnLabelDrawOrder ressource:

```
  nlevels=dimsizes(levels)
  flags=new(nlevels,string) ; Initialisation of flags as a string array
  flags(:)="LineOnly" ; Initialize the array with this string.
  flags(::2)="LineAndLabel" ; Change the array with a stride of 2
  res@cnLevelFlags=flags ; Set the resource
```

This cnLevelFlags resource has the same length than the levels vectors and indicates how to draw each specific contours. In this case, we want a line everywhere, and an additional label for one line out of two. Finally, to erase the white space under the label, you just have to force the label to be drawn first, by setting

```
   res@cnLabelDrawOrder="PreDraw"
```

### 4.2.3 Filled contour plot



Python:

Usually, it is nicer to add some colors to the figures. Here we will learn how to draw filled contour plots. In Python, you must use contourf instead of contour:

```
cs=contourf(lon,lat,t2,levels=levels,cmap=cm.get_cmap('RdYlBu_r'),extend='both')
```

The difference with the previous contour function is the cmap option, which is the choice of the colormap, and the extend option set to both. Python automatically mask the values that are greater than the maximum or lower than the minimum of the levels option. This option allows to disable this. This sole line does not draw black lines. They must be added with a contour command:

```
cs2=contour(lon,lat,t2,levels=cs.levels,colors='k')
```

It should be noted that the levels option here is set with the levels used in the contourf funtction, so that the black lines correspond to the interface of two colors.

Finally, to add the colorbar, it is simply the colorbar option:

```
cb=colorbar(cs,orientation='vertical',drawedges=True)
```

The drawedges option defines wether line should be drawn between the colors in the colorbar. Finally, to add the title to the colobar:

```
cb.set_label('Label colorbar')
```

NCL:

In NCL, you first have to define a colormap, which is done by using:

```
gsn_define_colormap(wks,"RdYlBu_r")
```

It is the same command as described previously that is used for filled contour plots. except that additional ressources need to be added. For filled contours:

```
   res@cnFillOn=True
   res@gsnSpreadColors=True
```

The gsnSpreadColors forces NCL to use the entire colormap for the figure. If set to False, if the number of levels to draw is lower than the number of colors in the colormap, NCL will select the first colors of the colormap so that the numbers of colors equals the number of levels. In NCL, the colorbar is automatically drawn. However, to have a nice rendering, it must be improved a little, by setting some resources.

```
   res@lbOrientation="Vertical"
   res@lbBoxLinesOn=False
```

Those resources define the orientation of the colormap and wether lines between the boxes must be drawn. To add a title to the colorbar:

```
   res@lbTitleString="Label colorbar"
```

In this case, the title will be drawn at the top of the colorbar, horizontally. We must change the position, direction and angle of the title so that is is drawn as in the previous figure. This is done by:

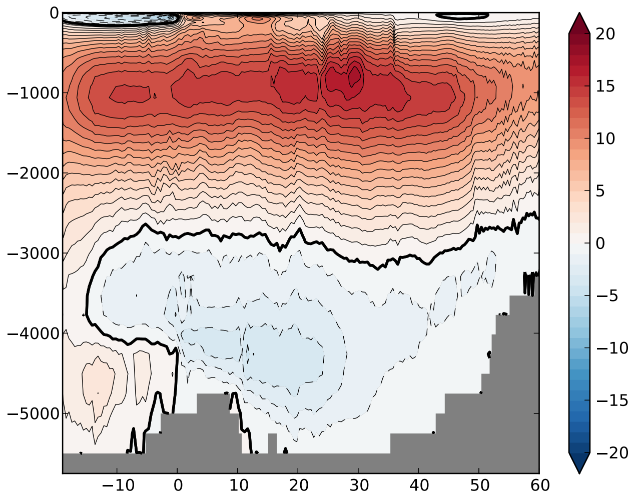
```
   res@lbTitlePosition="Right"
   res@lbTitleDirection="Across"
   res@lbTitleAngleF =90
```

### 4.2.4 Gray filled values

Python:

Here we will draw a sligthly more complicated contour. First of all, we notice that where data are missing, the color is gray. As contourf method draws transparently missing values, we must set the axes background color:

```
ax=gca()
ax.set_axis_bgcolor('gray')
```



Now, we want to have dashed lines for the negative contour lines. This is done by changing the rc default parameters:

```python
import matplotlib
matplotlib.rcParams['contour.negative_linestyle'] = 'dashed'
```

Now, we want to overlay the filled contours, thin contour lines and a thick one for the zero contour. This is done by using:

```python
cs=contourf(lat,z,moc,levels=levels,cmap=cm.get_cmap('RdBu_r'),extend='both')
# Contourf command
cs1=contour(lat,z,moc,levels=levels,colors='k',linewidths=0.5) # Thin contours
cs2=contour(lat,z,moc,levels=[0,0],colors='k',linewidths=2) # Thick zero contour line
```

Now, we might want to change the stride at which the labels are added to the colorbar. This is done by the following:

```python
cb.set_ticks(levels) # Draw one tick at each level. Handle irregular ticks
stride=5
setp(cb.ax.get_yticklabels(),visible=False) # Hide all the labels
setp(cb.ax.get_yticklabels()[::stride],visible=True) # Draw each 5 levels
```

However, we can notice that little tickmarks appear in the colorbar. They are due to the fact that a colorbar is drawn on an axes instance, which has tickmarks. While in the previous plot, it was not a huge problem, when doing this, it makes the colorbar unreadable. To draw those ticklines next to the labels:

```python
setp(cb.ax.get_yticklines(),visible=False)
setp(cb.ax.get_yticklines()[::2*stride],visible=True)
setp(cb.ax.get_yticklines()[1::2*stride],visible=True)
```

Moreover, Python allows to add lines to the colorbar at a given levels. Imagine you want to highlight that the thick contour is the zero line, you can add:

```python
cb.add_lines(cs2)
```

with cs2 being the contour instance for the zero line.

NCL:

As always with NCL, you do not have to change the function but only the resources. To define the gray color used for the continents, you must use:

```
    g=0.7
    newindex=NhlNewColor(wks,g,g,g)
```

This command add the color, the rgb of which are put in argument, to the workspace colormap. However, when drawing the contouring, the last contour will be drawn in gray because of the newcolor added (via NhlNewColor). To avoir the contouring to use this color, you must set

```
res@gsnSpreadColorEnd=newindex-1
```

To change the color in which are drawn the filled values, you must set

```
res@cnMissingValFillColor=newindex
```

Now to change the dash pattern of the negative contours and the zero line thickness:

```
    res@gsnContourNegLineDashPattern=2
    res@gsnContourZeroLineThicknessF=3
    res@lbOrientation="Vertical"
```

Finally, to change the stride of the colorbar labels:

```
    res@lbLabelStride=5
```

However, it should be noted that contrary to Python, NCL does not automatically interpolate the data onto a regular grid. In this case, we have many levels in the first few meters, which will cause a stretchin of the plot at the surface. To have an identical plit, we must force NCL to interpolate the data. This is done by adding the following resources:

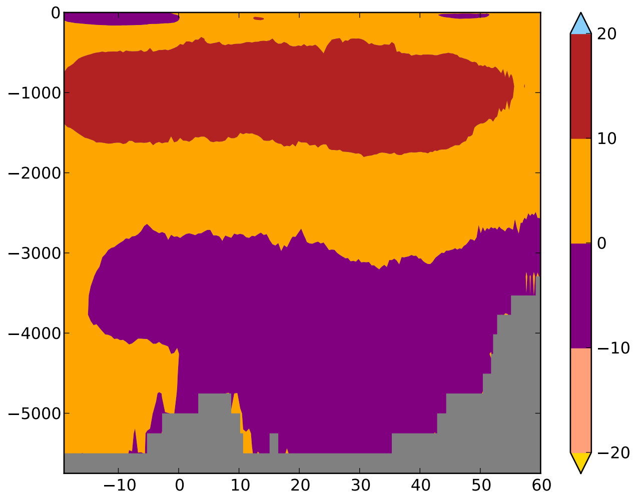
```
    res@gsnXAxisIrregular2Linear=True
    res@gsnYAxisIrregular2Linear=True
```

### 4.2.5 Homemade colormaps

Python:

It is often usefull to define our own colormap. Imagine we have a colormap that we want to draw from svg colors (such as "LightSkyBlue" for example). In python, one has to use the function ListedColormap:

```
from matplotlib.colors import ListedColormap
vect=['gold','lightsalmon','Purple','orange','firebrick','lightskyblue']
cmap=ListedColormap(vect)
```



This however has a drawback, as one must choose the number of levels identical to the number of colors. I suggest the use of another function, LinearSegmentedColormap('cmapname',dict,N=256). This function permits to generate a colormap made by linear interpolation on 256 value. The values used in the interpolation are contained in the dictionnary dict, which is as follows:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

The tuple contains the mapping value, the percentage of a color at the benning and at the end (usually, the two last values are identical). In this example, described in the matplotlib website, the colormap contains red that increases from 0 to 1 over the bottom half, the green of which does the same over the middle half, and blue over the top half.

NCL:

In NCL, it is done exactly in the same way:

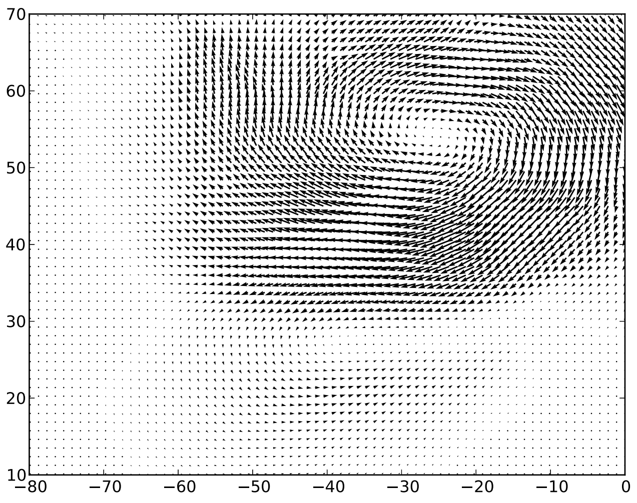
```
  cmap=(/"white","black","gold","LightSalmon","purple","orange","firebrick","
      lightskyblue"/)
  gsn_define_colormap(wks,cmap)
```

Compared to Python, we must start the colormap with white (background color) and black (foreground color).

*Note: I have chosen here to describe the method with named svg colors, but it works exactly in the same way with $N \times 3$ arrays. Moreover, the cpt-city website (`http://soliton.vm.bytemark.co.uk/pub/cpt-city/`) contains a huge amount of colormap. Python and NCL functions that permit to use them for filled contours can be downloaded in `https://dl.dropboxusercontent.com/u/99128427/Site/barriernicolas/Python_NCL.html` (you will* *also find examples on how to use them). The Python function also permits to deal with numpy arrays or list of named colors*

## 4.3 Vectors

### 4.3.1 First quiver plot



Python:

In Python, a quiver plot is done by using the quiver command:

```
quiver(lon, lat, u, v)
```

The first two options are the x and y coordinates (it can be either 1D or 2D), and the two last are the zonal and meridional components of the field.

NCL:

In NCL, you first have to define the x and y coordinates by setting the ressources:

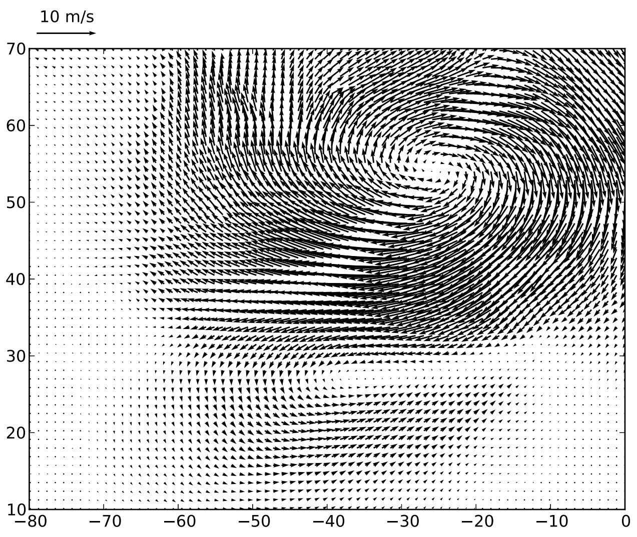
```
  res=True
  res@vfXArray = lon
  res@vfYArray = lat
```

Note that it is not the same ressources than for contour plots.
Finally, the plotting is done by:

```
 plot=gsn_csm_vector(wks,u,v,res)
```

### 4.3.2 Quiver length handling

Python:



To change the size of the arrows in Python, you must add the scale option to the quiver function:

```
qu=quiver(lon,lat,u,v,scale=100)
```

The scale gives you the size of the array as a function of the size of the domain. In our case, it means that the length of the array will be 1/100 the size of the domain.

Similarly to the colorbar, the legend of the quiverplot must be added to the figure. This is done by using the quiverkey function:

```
qk = quiverkey(qu,-75, 72,10,'10 m/s',coordinates='data')
```

This takes as arguments the quiverkey instance (qu), the position of the reference vector (here, in data coordinates, ie. longitudes and latitudes), the value of the reference vector and the corresponding string.

NCL:

In NCL, you set the length of the arrows by setting:

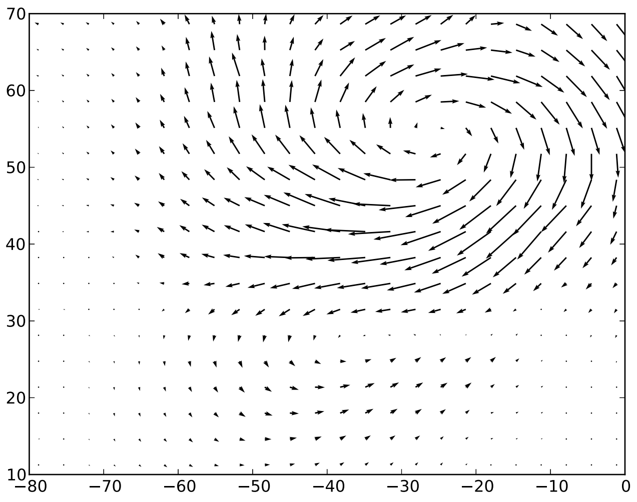
```
   res@vcRefMagnitudeF          = 10              ; make vectors larger
   res@vcRefLengthF             = 0.07            ; ref vec length
```

The first one controls the magnitude of the reference vector, the second one its length. In NCL, the reference vector is plotted automatically. However, to control it, you must set:

```
   res@vcRefAnnoOn = True ; Force the annotation
   res@vcRefAnnoString1 = "10 m/s"  ; Annotation string
   res@vcRefAnnoString2On = False ; Hide the 'reference vector'
   automatically added
   res@vcRefAnnoOrthogonalPosF   = -1.2 ; Y position of the reference vector
   res@vcRefAnnoParallelPosF   = 0.3 ; X position of the reference vector
   res@vcRefAnnoPerimOn=False ; Hide the bar around the reference
   vector
```

Note that it is not the same ressources than for contour plots.

### 4.3.3 Distance between the quivers



Python:

Oftenly, you must skip some arrows to have a nice rendering. One way to do it in Python is do draw only a few arrows:

```
stlat=3
stlon=3
qu=quiver(lon[::stlon],lat[::stlat],u[::stlon,::stlat],v[::stlon,::stlat],scale=100)
```

This will draw one arrow out of 3.

NCL:

In NCL, to do the same thing is very easy. You just have to set:

```
    res@vcMinDistanceF = 0.025
```

This resource controls the minimum distance to let between two consecutive arrows.

### 4.3.4 NCL curly vectors

One limitation with quiver plots in Python or Matlab is that they do not handle curly vectors. One strong advantage with NCL is that it does. You just have to set the resource:

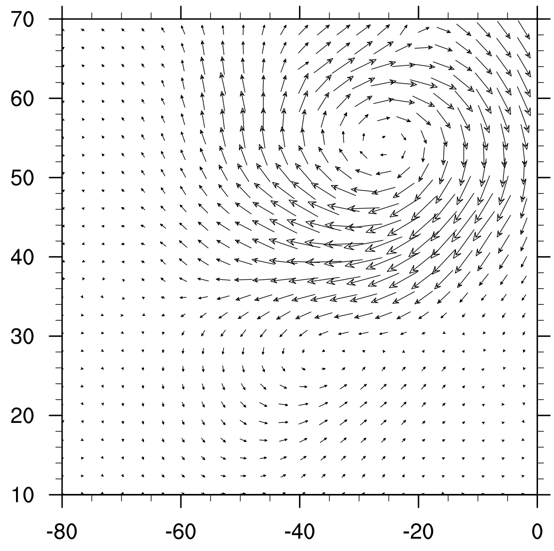
```
    res@vcGlyphStyle = "CurlyVector"
```

In figure 4.3.1, we compare the previous figure when curly vectors are activated or not.

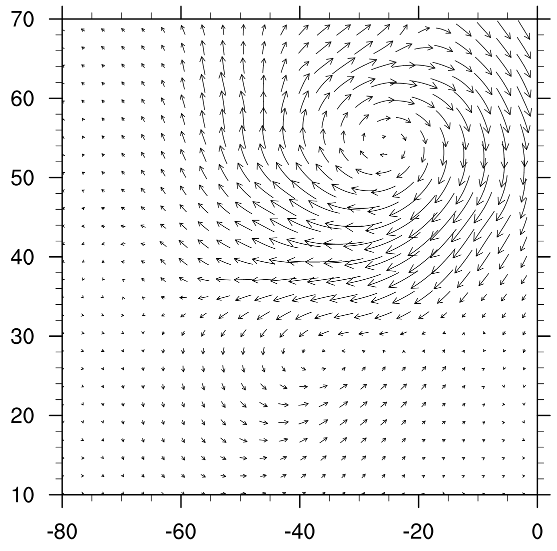
## 4.4 Panels

### 4.4.1 First panel plot

Python:

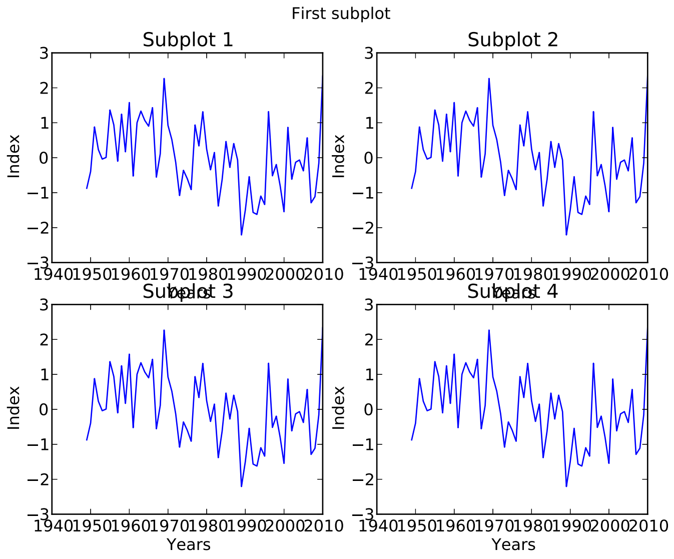


(a) Normal vectors)



(b) Curly vectors

Figure 4.3.1: Examples of quiver plots with NCL



```python
figure()

ax=subplot(2,2,1) # First panel
plot(years,indexes[0,:])
xlabel('Years')
ylabel('Index')
title('Subplot 1')

ax=subplot(2,2,2) # Second panel
plot(years,indexes[0,:])
xlabel('Years')
ylabel('Index')
title('Subplot 2')

ax=subplot(2,2,3) # Third panel
plot(years,indexes[0,:])
xlabel('Years')
ylabel('Index')
title('Subplot 3')

ax=subplot(2,2,4) # Four panel
plot(years,indexes[0,:])
xlabel('Years')
ylabel('Index')
title('Subplot 4')

suptitle('First subplot') # Global title
```

To draw panel plot in Python, it is simple. You have to use the subplot function to define your new axis. It takes as an input the number of lines, the number of column of your panel, and the number of the panel you want to draw. It should be noted that the use of the tight option when saving the figure erases the suptitle.

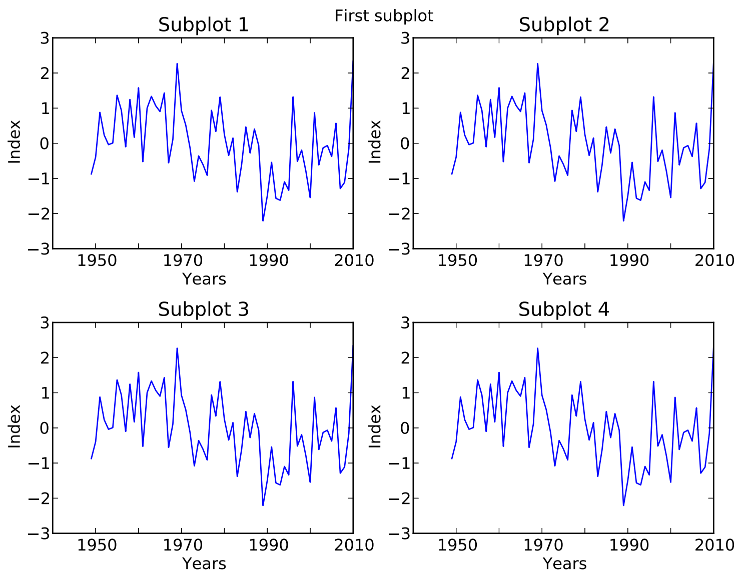
NCL:

```
 wks   = gsn_open_wks ("pdf","fig/"+get_script_prefix_name()+"_ncl")

  panel=new(4,graphic) ; Initialisation panel

  ; Define the ressources of each individual plots
  res                  = True
  res@gsnDraw=False
  res@gsnFrame=False
  res@tiXAxisString     = "Years"
  res@tiYAxisString     = "Index"

  res@tiMainString     = "Subplot 1" ; Change the title of the figure
  panel(0)  = gsn_csm_xy (wks,years,indexes(0,:),res) ; Add the figure to the panel

  res@tiMainString     = "Subplot 2"
  panel(1)  = gsn_csm_xy (wks,years,indexes(0,:),res)

  res@tiMainString     = "Subplot 3"
  panel(2)  = gsn_csm_xy (wks,years,indexes(0,:),res)

  res@tiMainString     = "Subplot 4"
  panel(3)  = gsn_csm_xy (wks,years,indexes(0,:),res)

  resP=True ; Resource for the set of panels
  resP@txString="First subplot" ; Global title
  gsn_panel(wks,panel,(/2,2/),resP) ; Draw the panel
```

NCL

In NCL, you have to define a new graphic instance, which will store each individual figure. This is done in line 3. Note the resources gsnFrame and gsnDraw set to False. If they were not, each individual figure will be drawn on a single pdf file. When you have attributed to each case of panel the individual plots, you define the resources of the panel plot (resP). The title is by txString resource and the panel plot is drawn with the gsn_panel panel function. It takes as an input the workspace, the graphic instance, the panel shapes (in this case, two lines, two columns) and the panel resources.

### 4.4.2 Space control



Python:

As you can see in the previous figure, it is sometimes necessary to control the spacing between the subplots. In Python, it is the subplots_adjust command which allows this.

```
figure()
subplots_adjust(wspace=0.2,hspace=0.35,top=1-0.07,bottom=0.07,left=0.07,right=1-0.07)
```

wspace is the white space along the X dimension (in percent), hspace is the white space along the Y dimension while the four other options are the start and end coordinates along the X and Y dimensions.

NCL:

```
    resP=True
    resP@txString="First subplot"
    resP@gsnPanelXWhiteSpacePercent=5 ; Add 5% of white spacing in X direction
    resP@gsnPanelYWhiteSpacePercent=5 ; Add 5% of white spacing in Y direction
    resP@gsnPanelLeft=0.1 ; Leftmost limit
    resP@gsnPanelRight=0.9 ; Rightmost limit
    resP@gsnPanelBottom=0.1 ; Lowermost limit
    resP@gsnPanelTop=0.9 ; Uppermost limit
    gsn_panel(wks,panel,(/2,2/),resP)
```

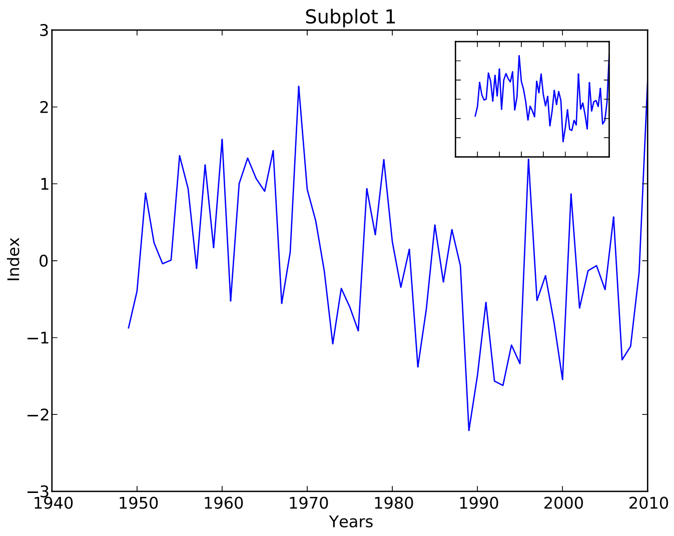
In NCL, you must set the gsnPanel resources described above, which have exactly the same purpose than the one described for Python.

To debug the paneling, you can set:

```
resP@gsnPanelDebug = True
```

This will print out the coordinates of each panel in the terminal.

### 4.4.3 Hand positionning



Python:

Sometimes, you must place the panels yourself for a better rendering. This can be the case if you try to subplot a time-series and a map. In Python, you must define an axes instance (when you use plot, it implicitely creates one by using the command gca()).

```
figure()
ax=axes([0.65,0.68,0.2,0.2])
plot(x,indexes[0,:])
```

It takes as an input the X and Y locations, the width and the height of the figure. When you use any of the matplotlib command, it is drawn on the last axes instance. You can also do this on a specific axes by doing:

```
figure()
ax=axes([0.65,0.68,0.2,0.2])
ax.plot(x,indexes[0,:])
```

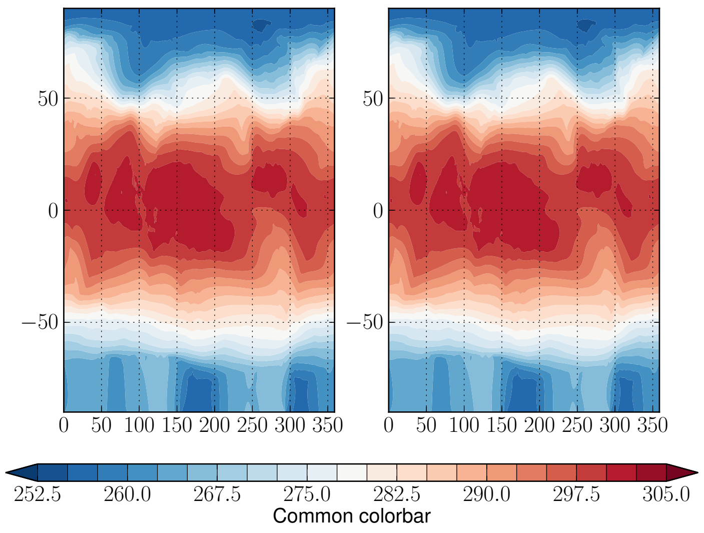
NCL:

In NCL, the positionning of the panels is done by setting the resources

```
  res@vpXF=0.6  ; X position of the left part of the inside figure
  res@vpYF=0.78  ; Y position of the top part of the inside figure
  res@vpWidthF=0.15  ; Width of the inside figure
  res@vpHeightF=0.15  ; Height of the inside figure
```

It should be notice that the vpYF resource is the location of the top of the panel, while in Python it is the bottom of the figure.

### 4.4.4 Shared colormap



Python:

If you want to compare two 2D fields, it might be useful to know how to define one common colorbar.

```
subplot(1,2,1) # First subplot
cs1=contourf(lon,lat,t2,20) # First contourf

subplot(1,2,2) # Second subplot
cs2=contourf(lon,lat,t2,levels=cs1.levels) # Second contourf, same levels

cax=axes([offx,0.08,1-2*offx,0.03]) # Define axes for the colorbar
cb=colorbar(cs2,cax,orientation='horizontal') # Draw horizontal colorbar
cb.set_label('Common colorbar') # Label colorbar
```

As we try to compare two fields, it is important that both contours share the same levels. To retrive the levels of the first contours, you just have to use cs1.levels command. Then, you must define the position in which the common colorbar will be drawn. This is done by using the axes instance described in the previous subsection. Then, when using colorbar function, you must specify the axes that will be used for the colorbar drawing.

NCL:

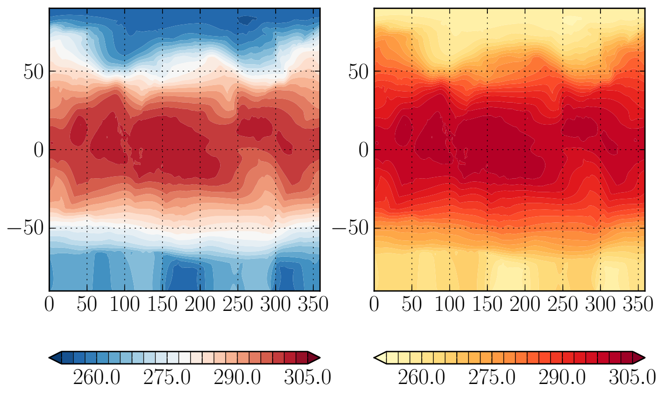
In NCL, you first have to deactivate the drawing of individual colormaps.

```
  res@lbLabelBarOn = False ; Hide individual colorbars
```

Then you have to set to true the Panel resource:

```
    resP=True ; Panel resources
    resP@gsnPanelLabelBar = True ; Force panel colorbarq
```

### 4.4.5 Two colormaps



Python:

If you want to subplot two figures with different colorbar (for example to compare a mean, and an anomaly), it is easy in Python. You draw your subplots independently, you draw the colorbar and you are all set. However, the colorbar labels might not be visible. You thus must hide som of the labels:

```
subplot(1,2,1)
cs1=contourf(lon,lat,t2,20,cmap=cm.get_cmap('RdBu_r')) # First contourf
cb=colorbar(cs1,orientation='horizontal') # Drawing of the first colorbar
for t in cb.ax.get_xticklabels()[::2]: # Hiding of one colorbar label out of 2
    setp(t,visible=False)

cs2=contourf(lon,lat,t2,20,cmap=cm.get_cmap('RdYlBu_r')) # Second contourf
cb=colorbar(cs2,orientation='horizontal') # Drawing of the second colorbar
```

NCL:

In NCL, it is far more complicated to do such a thing. Fist of all, we must merge the two colormaps.

```
gsn_merge_colormaps(wks,"BlueDarkRed18","cmp_flux")
```

This will give one big colorbar. Then you have to draw the first plot by defining the resources gsnSpreadColorStart and gsnSpreadColorEnd to the first and last elements of the first colorbar:

```
    res@gsnSpreadColors=True
    res@gsnSpreadColorStart=2
    res@gsnSpreadColorEnd=19
    panel(0) = gsn_csm_contour (wks,t2,res)
```

Then, you do exactly the same thing but after changing the values of the previous resources:

```
res@gsnSpreadColorStart=20
res@gsnSpreadColorEnd=32
panel(1)   = gsn_csm_contour (wks,t2,res)
```

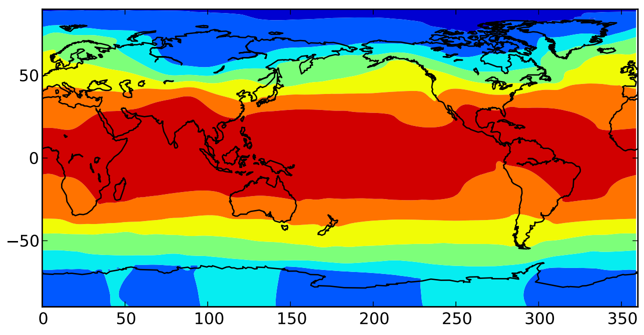
To help with the colors indexes to put, a very helpflul function is

```
gsn_draw_colormap(wks)
```

which will draw the colorbar defined in the wks, with the numbers corresponding to each color.

## 4.5 Maps

### 4.5.1 First map



Python:

In Earth Science, it is necessary to be aware of how to draw data on a map. In Python, you first must import the Basemap function

```
from mpl_toolkits.basemap import Basemap
```

Now the next step is to define the map object, which is done by the Basemap command.

```
m=Basemap(llcrnrlat=-90,llcrnrlon=0,urcrnrlat=90,urcrnrlon=360,suppress_ticks=False)
```

Here we set the lower left and upper right coordinates of the map. In this case, longitudes are from 0 to 360. The next step is to define the map coordinates (instead of longitude and latitude coordinates) that will be used in the contouring:

```
lonI,latI=np.meshgrid(lon,lat)
x,y=m(lonI,latI)
```

It should be noted that for the drawing, we must have 2D coordinates. This is done with the numpy meshgrid function. Now, we can do contour, contourf, pcolor or quiver plots on the map as follow:

```
m.contourf(x,y,t2)
```

Now, we must add the coastlines, which is done as follow.

```
m.drawcoastlines()
```

NCL:

In NCL, it is very similar than to do a contourf plot. You first have to set-up the resources:

```
    res@sfXArray=lon
    res@sfYArray=lat
    res@cnFillOn             = True
    res@gsnSpreadColors=True
    res@gsnAddCyclic=False
```

The gsnAddCyclic is set to False because in this case, you have data that spans the entire globe and thus need no additional cyclic point. So far, NCL will automatically draw the map with the Atlantic in the center of the plot. However, we have data in which the Pacific is in the center. You thus must add the ressource:

```
    res@mpCenterLonF = 180
```

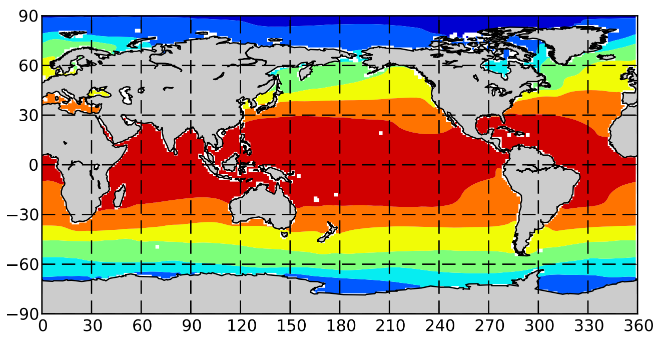
The last step is to draw the contour map. This is done with the gsn_csm_contour_map command:

```
    plot=gsn_csm_contour_map(wks,t2,res)
```

NCL automatically draws continental outlines. To hide them, you must set the resource:

```
    mpOutlineOn=False
```

### 4.5.2 Grid handling



Imagine that data are masked on non-ocean grid points, and that we want to add coordinate grids. It should be noted that the classical grid method described in the first section, as they are working on axes objects, are not applicable here.

Python:

In Python, you first have to define the meridians and the parallels in which you want to add the map grid.

```
mer=np.arange(0,360+30,30)
par=np.arange(−90,90+30,30)
```

Now, to effectively draw the grid, you must use the following commands

```
m.drawparallels(par,dashes=[10,5])
m.drawmeridians(mer,dashes=[10,5])
```

The dashes options specifies the length of the lines (10) and the lines of the blanks (5) for the grid. However, in this case, the xticks and yticks do not coincide with the grid lines. We thus have to set

```
ax.set_xticks(mer)
ax.set_yticks(par)
```

Finally, to fill the continents:

```
m.fillcontinents()
```

NCL:

Here, we assume that the gray color was defined as in the previous section (newindex). The grid handling is done with the following resources

```
  res@mpGridAndLimbOn          =  True ; Grid on
  res@mpGridLineDashPattern       = 2 ; dashed grid
  res@mpGridLatSpacingF = 30 ; Meridional grid spacing
  res@mpGridLonSpacingF = 30 ; Zonal grid spacing
  res@gsnMajorLatSpacing=30 ; Major meridional grid spacing
  res@gsnMajorLonSpacing=30 ; Meridional grid spacing
```

Finally, to change the color of the land:

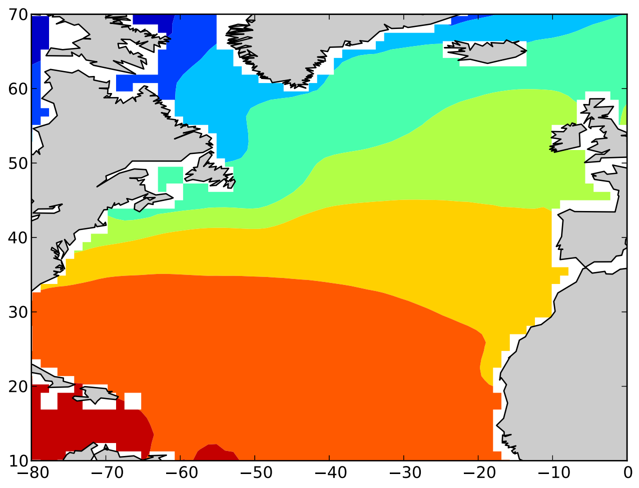
```
  res@mpLandFillColor         = newindex
```

### 4.5.3 Shifting longitudes

In the previous maps, longitudes span 0 to 360, which implies that the map will be centered in the Pacific. Here, we will learn how to shift the longitudes so that they span -180 to 180. We will also learn how to limit the spatial domain.

Python:

In Python, the shifting of the grid is done by using the shiftgrid command; which must be imported beforehand.



```python
from mpl_toolkits.basemap import shiftgrid
t2,lon = shiftgrid(180.,t2,lon,start=False)
```

The first argument is the starting longitude if start option is set to True, and the ending longitude otherwise. It takes as an input the field to shift and the corresponding longitudes. We have already shown previously how to limit the spatial domain of the map:

```python
m=Basemap(llcrnrlat=10,llcrnrlon=-80,urcrnrlat=70,urcrnrlon=0,suppress_ticks=False)
```

NCL:

In NCL, the switching of the longitudes is performed with the lonFlip command, the argument of which is the field assuming that it has longitudes coordinates attached to it.

```
t2=lonFlip(t2)
```

Now, the limitation of the spatial domain of the grid is done by setting the ressources:

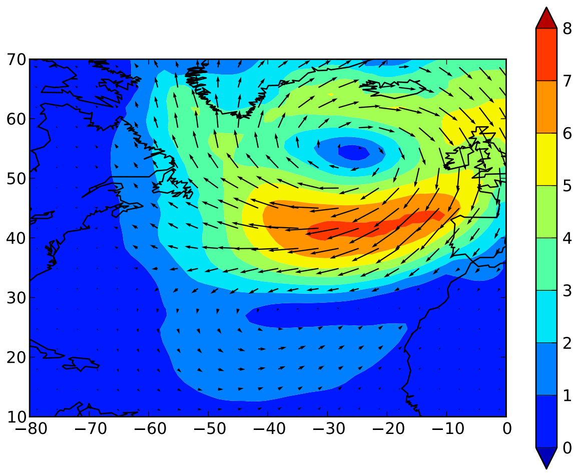
```
  res@mpLimitMode="LatLon"
  res@mpMinLatF               = 10
  res@mpMaxLatF               = 70
  res@mpMinLonF               = -80
  res@mpMaxLonF               = 0
```

The first one defines how we limit the spatial domain. Here, we set Longitudes and Latitudes limits, which is the easiest, but there are many other ways.

### 4.5.4 Plot Overlays

Python:

In Python, the overlay is implicitely done. The second plot is drawn on top of the first one. For example, if we want to overlay a quiver plot on top of a contourf plot:



```
cs=m.contourf(x,y,ws)
qu=m.quiver(x[::stlon,::stlat],y[::stlon,::stlat],u[::stlon,::stlat],v[::stlon,::stlat],
    scale=100)
colorbar(cs)
```

NCL:

In NCL, overlays on a map are slightly more complicated. First, you have to set-up the resources for the contourf plot on a map (see section ??).

```
resC=True
resC@mpLimitMode="LatLon"
resC@mpMinLatF              = 10
resC@mpMaxLatF              = 70
resC@mpMinLonF              = −80
resC@mpMaxLonF              = 0
resC@gsnFrame=False
resC@gsnDraw=False
resC@sfXArray = lon
resC@sfYArray = lat
resC@gsnAddCyclic=False
resC@cnInfoLabelOn=False
resC@cnFillOn=True
resC@gsnSpreadColors=True
```

The seconde step is to define the resources of the vector plot. **Here, we do not add resources relative to maps.**

```
resV=True
resV@gsnFrame=False
resV@gsnDraw=False
resV@vfXArray = lon
resV@vfYArray = lat
resV@vcRefAnnoOn=False
resV@vcMinDistanceF          = 0.022        ; thin out vectors
resV@vcRefMagnitudeF         = 10          ; make vectors larger
resV@vcRefLengthF            = 0.07        ; ref vec length
resV@vcRefAnnoOn=False
resV@vcLineArrowThicknessF = 2
resV@vcLineArrowColor=0
```

Note that in each case, the gsnFrame and gsnDraw resources are set to False. Then, you must draw the contourf and the vector plot:

```
  plotc=gsn_csm_contour_map(wks,wspeed,res)
  plotv=gsn_csm_vector(wks,u,v,resV)
```

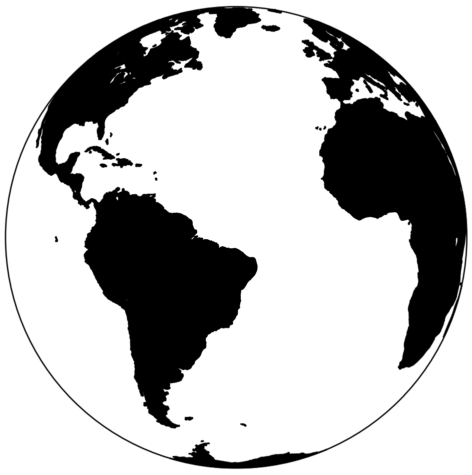
You would have notice that in the first case, we draw on a map contrary to the second case. The last step is to overlay both figures:

```
  overlay(plotc,plotv)
```

This will project the plotv onto the map of the plotc vector. The next step is to draw the contourf (and as the plotv is now attached to it, it will also be drawn) and to open a frame:

```
draw(plotc)
frame(wks)
```

### 4.5.5 Orthographic projection



Python:

In Python, to change the projection of the map, you must set the projection option in the Basemap function:

```
m=Basemap(resolution='l',projection='ortho',lon_0=-40,lat_0=0)
```

Moreover, you will notice that we have added two more options, which are the longitude and the latitude at the center of the map. Now, we must fill the continents (or draw the coastlines):

```
m.fillcontinents(color='k')
```

So far, the Earth is not "closed", meaning that the continents seem to drift in the paper. To close the Earth (ie. to draw the circle that will limit the Earth), you must use the drawmapboundary function:

```
m.drawmapboundary()
```

NCL:

In NCL, the choice of the projection and of the longitudes/latitudes of the center of the map are done with the ressources:

```
    res@mpLimitMode="LatLon"
    res@mpProjection          = "Orthographic"
    res@mpCenterLonF  =  −40.
    res@mpCenterLatF  =  0.
```

Now, if we want to close the Earth, we must set the following resources:

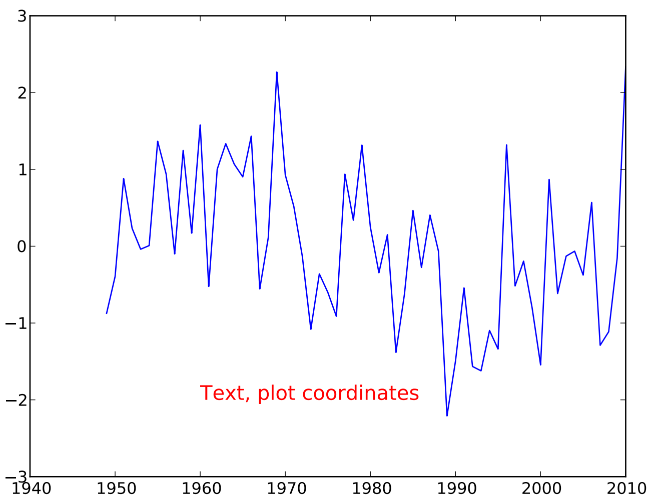
```
    res@mpGridAndLimbOn          =  True
    res@mpGridMaskMode           = "MaskFillArea"
    res@mpPerimOn=False
```

The first one allows to draw the circle that will close the Earth, but it will also add the Earth Grid. The second resource allows to hide the grid and to let only the circle. Automatically, NCL draws a box around the map, which can be hidden by the last resource. Finally, the drawing of the map only (without contours or quivers) is done by:

```
    plot=gsn_csm_map(wks,res)
```

## 4.6 Text

### 4.6.1 Plot coordinates



Python:

In Python, to add text in map coordinates is pretty straigthforward. You simply need to use the `text` command. If we want to add text on figure ??, you must add:

```
text(1960,-2,'Text, plot coordinates',fontsize=15,color='r')
```

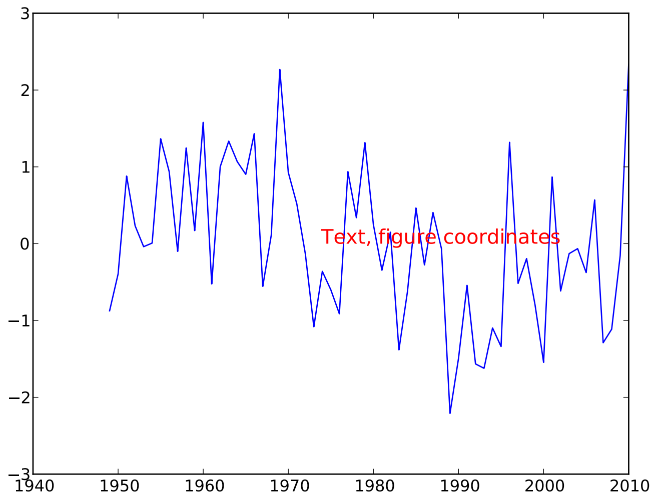
NCL:

In NCL, you must use the command `gsn_text` .

```
txres                  = True                    ; text mods desired
txres@txFontHeightF = 0.03                    ; font smaller. default big
txres@txFontColor="red"
gsn_text(wks,plot,"Text, plot coordinates",1960,-2,txres)
```

`txres` are the resources of the text (ie. fontsize, font color). The function takes as an input the workspace and the plot in which the text will be added, the text its X and Y coordinates and finally the text resources.

### 4.6.2 Figure coordinates



Python:

It can be usefull to add text in figure coordinates. In Python, it is done by using `figtext` instead of `text` .

```
figtext(0.5,0.5,'Text, figure coordinates',fontsize=15,color='r')
```

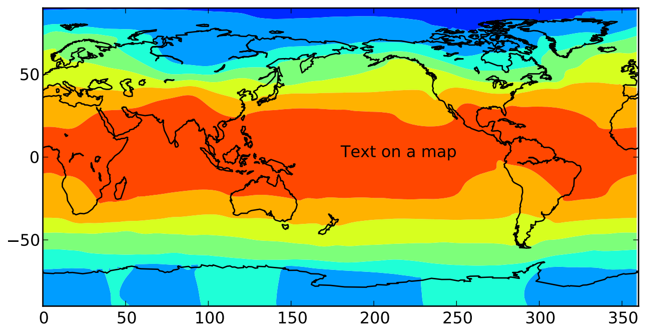
NCL:

In NCL, you also need to use another function:

```
gsn_text_ndc(wks,"Text, figure coordinates",0.5,0.5,txres)
```

This command ressembles the `gsn_text` command, but takes as an input the coordinates of the figure.

### 4.6.3 Text on a map



Python:

Imagine that you want to add text on the map of figure ??. Similarly with the previous example, you must use the text command. However, you first must convert the longitude/latitude coordinates into map coordinates as follows:

```
m=Basemap(llcrnrlat=-90,llcrnrlon=0,urcrnrlat=90,urcrnrlon=360)
xt,yt=m(180,0)
text(xt,yt,"Text on a map")
```

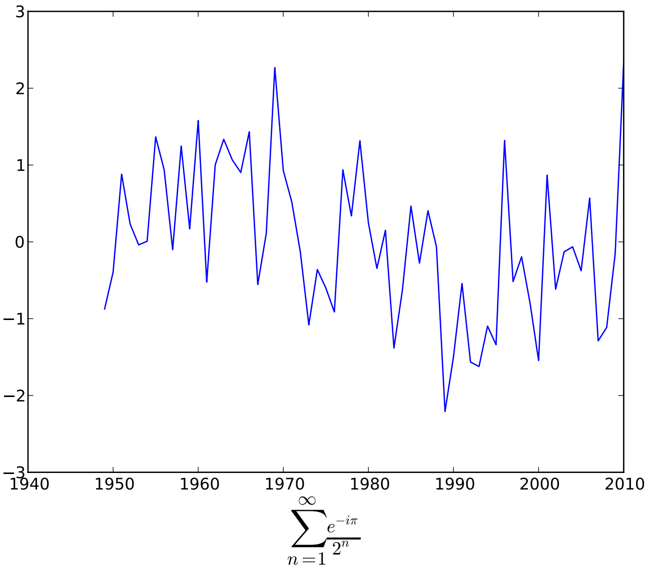
NCL:

In NCL, you must use another function to add text on a map, `gsn_add_text`.

```
   text=gsn_add_text(wks,plot,"Text on a map",180,0,txres)
```

It works identically to the `gsn_text` command.

### 4.6.4 Equations



Python:

You might need to add equations to a figure. In Python, it is quiete easy as it follows the LaTeX syntax.

```
xlabel(r'$\sum_{n=1}^{\infty} \frac{e^{-i\pi}}{2^n}$',fontsize=20)
```

You notice that before writing the string, we have add a `r`, which force Python to use raw strings. Its omission will raise an exception. The mathmode is opened and closed by `$` signs, as in LaTeX.

*Note*: It should be noted that Python allows to use TeX rendering. This is done as follows:

```
import matplotlib
matplotlib.rcParams['text.usetex']=True
```

The use of TeX rendering slightly change the syntax of the equation:

```
xlabel(r'\TeX\  $\displaystyle \sum_{n=1}^{\infty} \frac{e^{-i\pi}}{2^n}$',fontsize=20)
```

Indeed, the mathmode is now opened by a `$` followed by `\displaystyle`.

NCL:

One major limitation of NCL compared to Python is the handling of equations. Indeed, you must place everything by hand, which can become quiete restristing. This is done by using numerous *function codes*. Those are the symbols located between `::`

```
res@tiXAxisString      = ":F34:e   :H-65:  :V-30:  :Z50:  :F21:  n=1
:Z100:  :H-70:  :V50:  :H10:  :F18:x  :H-20:  :F21:e  :S:  -i  :H-10:  :F8:p  :N:
:V-5:  :X300:  :H-190:  :F21:_  :H-120:  :V-30:  :Z90:  :H-10:  :F21:  2:S:n"
```

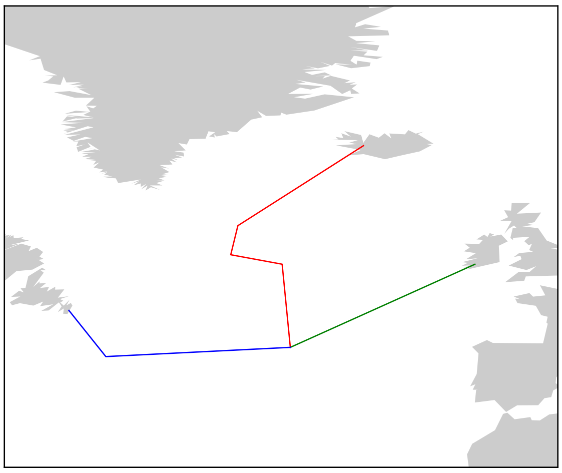
This line permits to draw as best as possible the string axis. .They allow string manipulations. For example, `:F34:e` draw the $\sum$ symbol. `:H-65:` offsets the following string to the left, `:V-30:` offsets the following string to the top, `:Z50:` diminishes the size of the following string by 50%, `:F21:` changes the font, `:F18:x` is the $\infty$ symbol, `:S:` implies that the following strings are superscripts, `:N:` forces to return to normal levels, `:F8:p` is the $\pi$ symbol and finally `:X300:` forces the following string widths to increase by 300%.

Obviously, the syntax in NCL is far more complicated than the syntax in Python and for a worst rendering. I would advise not to add complicated equations to a NCL plot.

## 4.7 Polylines

### 4.7.1 Polylines on a map

Python:



Polylines are often usefull on a map, to draw sections for examples. In python, the plot command permits to do it simply. The first step is to define longitude and latitude arrays that contains the points locations of the line:

```
marx=[-21.065,-34.685,-35.459,-29.887,-29.]
mary=[64.86,56.189,53.036,52.011,43.]
qdex=[-29,-9]
qdey=[43,52]
qdwx=[-29,-49,-53]
qdwy=[43,42,47]
```

If m has been initialized as a basemap object,

```
m=Basemap(llcrnrlon=-60,llcrnrlat=30,urcrnrlon=0,urcrnrlat=80)
```

Then, each longitude and latitude needs to be converted into map coordinates, as done for contouring on a map, prior the use of the plit function:

```
xp,yp=m(marx,mary)
m.plot(xp,yp,color='r')
xp,yp=m(qdex,qdey)
m.plot(xp,yp,color='g')
xp,yp=m(qdwx,qdwy)
m.plot(xp,yp,color='b')
```

NCL:

In NCL, the philosophy is slightly different. We attach polylines object to an existing plot. We first draw, let's say, a contour plot:

```
  res=True
  res@gsnFrame=False
  res@gsnDraw=False
  plot=gsn_csm_contour_map(wks,t2,res)
```

with res the resources of the contour plot. Note that gsnFrame and gsnDraw need to be set to False in order to prevent multiplot plots (one without and one with the polylines).

Similarly with Python, we define the coordinates of the lines:

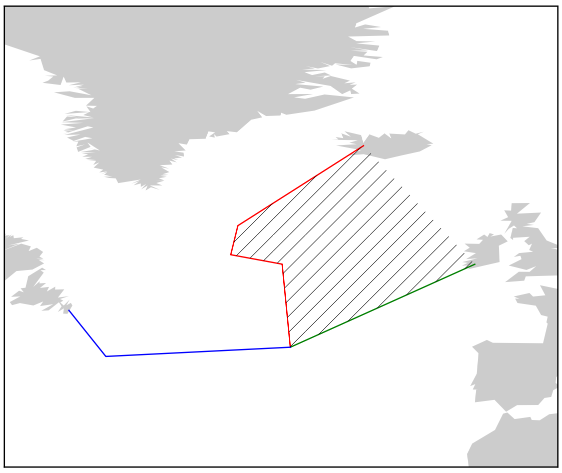
```
  marx=(/-21.065,-34.685,-35.459,-29.887,-29./)
  mary=(/64.86,56.189,53.036,52.011,43./)
  qdex=(/-29,-9/)
  qdey=(/43,52/)
  qdwx=(/-29,-49,-53/)
  qdwy=(/43,42,47/)
```

Then, we define the resources of the polylines and use them in the gsn_add_polyline function.

```
  lres=True
  lres@gsLineThicknessF = 4.0              ; line thickness

  lres@gsLineColor       = "Red"        ; color of lines
  dum1 = gsn_add_polyline(wks,plot,marx,mary ,lres)

  lres@gsLineColor       = "Blue"        ; color of lines
  dum2 = gsn_add_polyline(wks,plot,qdex,qdey,lres)

  lres@gsLineColor       = "Green"        ; color of lines
  dum3 = gsn_add_polyline(wks,plot,qdwx,qdwy,lres)

  draw(plot)
  frame(wks)
```

One limitation with NCL is the need to have different names for the attached polylines (dum1, dum2, etc).

### 4.7.2 Filled polygon on a map



Python:

To show area of importance, it might be needed to draw filled polygons. In Python, this is done thanks to matplotlib patches and artist. The first step is to import the Polygon patches:

```
from matplotlib.patches import Polygon
```

Then, similarly with what has been done previously, we define the polygon coordinates as a $2 \times N$ array (xy), with the first dimension being the x coordinates, and the second dimension being the y coordinates.

```
    px=[-21.065,-34.685,-35.459,-29.887,-29.,-29,-9]
    py=[64.86,56.189,53.036,52.011,43.,43,52]
    xp,yp=m(px,py)
    xy=np.transpose(np.array([xp,yp]))
```

Then, we create the polygon artist:

```
    pol=Polygon(xy,closed=True,fill=False,hatch='/',edgecolor='k',linewidth=0.0,zorder=20)
    ax.add_artist(pol)
```

The closed option indicates that we want closed polygones, edgecolor indicates the hatching color, the hatch option defines the hatching pattern, the fill option specifies wether the polygons is filled or not, the linewidth option specifies the polygon boundaries width (here, we do not have boundaries) and zorder specifies the order of the drawing (the polygon will be the $20^{th}$ object drawn.

Finally, the artist needs to be added to the axes object. If we assume that ax=gca():

```
    ax.add_artist(pol)
```

NCL:

In NCL, adding polygons is much more simple and very alike adding polylines. This is done using the gsn_add_polygon function. We first define the polygon vectors:

```
    x=(/-21.065,-34.685,-35.459,-29.887,-29.,-29,-9/)
    y=(/64.86,56.189,53.036,52.011,43.,43,52/)
```

We then set the resources of the polygon plot.

```
    polres=True
    polres@gsFillColor    ="Black"
    polres@gsFillIndex=2 ; change the hatching patterns
```

The gsFillColor and gsFillIndex set the color and fill pattern of the polygon plot, respectively. We finally use the function to draw the polygon:

```
    dum4=gsn_add_polygon(wks,plot,x,y,polres)
```