Python Programming
for
Data Processing and Climate Analysis

Jules Kouatchou and Hamid Oloso
Jules.Kouatchou@nasa.gov and Amidu.o.Oloso@nasa.gov

Goddard Space Flight Center
Software System Support Office
Code 610.3

April 8, 2013
We want to introduce:

- **Basic concepts of Python programming**
- **Array manipulations**
- **Handling of files**
- **2D visualization**
- **EOFs**
Based on the feedback we have received so far, we plan to have a hand-on presentation on the following topic(s):

**F2Py:**
- Python interface to Fortran
- Date: April 29, 2013 at 1:30pm

**iPython Notebook:**
- A web-based interactive computational environment
- Tentative Date: TBD
Obtaining the Material

Slides for this session of the training are available from:

https://modelingguru.nasa.gov/docs/DOC-2322

You can obtain materials presented here on discover at

/discover/nobackup/jkouatch/pymonsTrainingGSFC.tar.gz

After you untar the above file, you will obtain the directory pythonTrainingGSFC/ that contains:

Examples/
Slides/
Settings on *discover*

To use the Python distribution:

```bash
module load other/comp/gcc-4.5-sp1
module load lib/mkl-10.1.2.024
module load other/SIVO-PyD/spd_1.9.0_gcc-4.5-sp1
```

To use uvcdat:

```bash
module load other/uvcdat-1.2-gcc-4.7.1
```
What Will be Covered Today

1. Basic Introduction to EOF
2. Data Source & EOFs with NCL
3. Two Approaches for Doing EOFs
   - CDAT
   - Numpy
Useful Links on EOFs


What is EOFs Analysis?

- Empirical Orthogonal Function (EOF) analysis attempts to find a relatively small number of independent variables (predictors; factors) which convey as much of the original information as possible without redundancy.

- EOF analysis can be used to explore the structure of the variability within a data set in a objective way, and to analyze relationships within a set of variables.

- EOF analysis is also called principal component analysis or factor analysis.
What Does EFO Analysis Do?

Uses a set of orthogonal functions (EOFs) to represent a time series in the following way

\[ Z(x, y, t) = \sum_{k=1}^{N} P_k(t) \times E_k(x, y) \]

- \( Z(x, y, t) \) is the original time series as a function of time \((t)\) and space \((x, y)\).
- \( E_k(x, y) \) show the spatial structures \((x, y)\) of the major factors that can account for the temporal variations of \( Z \).
- \( P_k(t) \) is the principal component that tells you how the amplitude of each EOF varies with time.
What Does EOF Analysis Provides?

1. A set of EOF **loading patterns** (eigenvectors)
2. A set of corresponding **amplitudes** (temporal scores)
3. A set of corresponding **variances accounted for** (eigenvalues)
Data

- We use Sea Level Pressure from NCEP/NCAR Reanalysis 1
- Monthly means from 1948 to present
- $2.5 \times 2.5$ grid resolution
- Global data
Verification

- We can use NCAR Command Language (NCL) to perform EOF calculations.
- Show good demonstration of typical steps (data sub-setting, seasonal time averaging, plotting of EOFs, etc)
- For plots: http://www.ncl.ucar.edu/Applications/eof.shtml
- For source code:
  http://www.ncl.ucar.edu/Applications/Scripts/eof_1.ncl
NCL: Seasonal SLP Plot
What is eof2?

- Python package for performing EOF analysis on spatial-temporal data sets
- Suitable for large data sets
- Transparent handling of missing values
- Work both within a CDAT environment or as a stand-alone package
- Provides two interfaces (supporting the same sets of operations) for EOF analysis:
  1. Numpy arrays
  2. cdms2 variables (which preserves metadata)
- For more information: http://ajdawson.github.com/eof2/
eof2 Solver

- Uses SVD
- Expects as input a spatial-temporal field represented a an array (Numpy array or cdms2 variable) of two or more dimensions.
- Internally, any missing values in the array are identified and removed.
- The EOF solution is computed when an instance of `eof2.Eof` (for cdms2) or `eof2.EofSolver` (for Numpy) is initialized.
Here are some functions associated with the solver:

- **eofs**: Array with the ordered EOFs along the first dimension.
- **eofsAsCorrelation**: EOFs scaled as the correlation of the PCs with the original field.
- **eofsAsCovariance**: EOFs scaled as the covariance of the PCs with the original field.
- **eigenvalues**: Eigenvalues (decreasing variances) associated with each EOF.
- **pcs**: Principal component time series (PCs). Array where the columns are the ordered PCs.
How to Use eof2?

```python
from eof2 import Eof
# from eof2 import EofSolver
...
# Initialize and Eof object.
# Square-root of cosine of latitude weights are used.
solver = Eof(myData, weights='coslat')

# coslat = np.cos(np.deg2rad(lats))
# wgts = np.sqrt(coslat)[..., np.newaxis]
# solver = EofSolver(myData, weights=wgts)

# Retrieve the first two EOFs.
eofs = solver.eofs(neofs=2)

# Retrieve the eigenvalues
eigenVals = solver.eigenvalues()
```
What is CDAT?

- CDAT: Climate Data Analysis Tools
- Software "glued" under the Python framework
- CDAT packages use:
  - cdms2 - Climate Data Management System (file I/O, variables, types, metadata, grids)
  - cdutil - Climate Data Specific Utilities (spatial and temporal averages, custom seasons, climatologies)
  - vcs - Visualization and Control System (manages graphical window: picture template, graphical methods, data)
```python
f = cdms2.open('slp.mon.mean.nc')
slp = f('slp', time=('1979', '2003'),
    latitude=(85, 20), longitude=(-70, 40))
f.close()

# Put time point at the beginning instead of middle of month
cdutil.setTimeBoundsMonthly(slp)

# Extract Dec-Jan-Feb seasons
djfslp = cdutil.DJF(slp)

coslat = np.cos(np.deg2rad(slp.getLatitude()[:][:]))
wgts = np.sqrt(coslat)[..., np.newaxis]

slpsolver = Eof(djfslp, weights=wgts)
eofs = slpsolver.eofs(neofs=3)
eigenvalueVec = slpsolver.eigenvalues()
```
CDAT: Looking at the Results

```python
print eigenvalueVec()
print 100*eigenvalueVec()[0:3]/fsum(eigenvalueVec())

# Initialise a VCS canvas for plotting
p = vcs.init()

# Plot the first EOF
p.plot(eofs[0],'default','isofill')
```
CDAT: Seasonal SLP Plot
How Do We Do EOF?

We use:

- netCDF4: to read the dataset
- Numpy: to manipulate arrays
- eof2: for EOF calculations
- Matplotlib/Basemap: for plotting
Numpy: Reading the Data

```python
ncin = Dataset('slp.mon.mean.nc', 'r')

lons = ncin.variables['lon']
lats = ncin.variables['lat']
time = ncin.variables['time']

slp = ncin.variables['slp']
```
# get the "unit" of variable time
```
timeUnit = time.getncattr('units')
```

# extract the numbers in time coordinate of the
# time range of interest (January 1979 to January 2003)
```
dateNum1 = date2num(datetime(1979,1,1,0,0),
                      units=timeUnit)
dateNum2 = date2num(datetime(2003,1,1,0,0),
                      units=timeUnit)
```

# obtain the time coordinate indices of time range of interest
```
dateIndex1 = np.where(time[:] == dateNum1)[0][0]
dateIndex2 = np.where(time[:] == dateNum2)[0][0]
```

# generate a date index array for time coordinate
# indices in time range of interest
```
dateIndex = np.arange(dateIndex1, dateIndex2+1)
```
Numpy: Spatial Subsetting

```python
# Realign SLP so that longitude goes from -180 to 180
slpShift = np.zeros((slp.shape[0], slp.shape[1],
                     slp.shape[2]), dtype=np.float32)
for i in range(slp.shape[0]):
    slpShift[i,:,:], lonShift =
    shiftgrid(180., slp[i,:,:], lon[:], start=False)

# Subsetting and Resersing latitudes
latIndex = np.nonzero((lat[:::-1]>= 25 ) &
                      (lat[:::-1]<= 80 ))[0]

# Subsetting longitudes
lonIndex = np.nonzero((lonShift >= -70 ) &
                       (lonShift <=40 ))[0]

# extract the desired data subset for analysis
slpSubset=slpShift[:,:::-1,:] # dateIndex[0]:dateIndex[-1]+1
                                # latIndex[0]:latIndex[-1]+1,lonIndex]
```
Numpy: Seasonal Climatology

```python
# Compute Dec-Jan-Feb seasonal climatology
slpDJF = np.zeros((25, latIndex.shape[0], lonIndex.shape[0]), dtype=np.float32)
slpDJF[0, :, :] = (slpSubset[0, :, :] * 31. + slpSubset[1, :, :] * 28.) / (31. + 28.)  # first season

for i in range(1, 24):
    if np.mod(i, 4) == 1:  # leap year
        slpDJF[i, :, :] = ((slpSubset[12.*i-1., :, :] + slpSubset[12.*i, :, :]) * 31. + slpSubset[12.*i+1., :, :]*29.) / (31.+31.+29.)
    else:  # non leap year
        slpDJF[i, :, :] = ((slpSubset[12.*i-1., :, :] + slpSubset[12.*i, :, :]) * 31. + slpSubset[12.*i+1., :, :]*28.) / (31.+31.+28.)

slpDJF[24, :, :] = (slpSubset[-2, :, :]+slpSubset[-1, :, :])*31./62.
```
Numpy: EOF Solver

# Create an EOF solver to do the EOF analysis.
# Square-root of cosine of latitude weights are applied before the computation of EOFs.

coslat = np.cos(np.deg2rad(lat[::-1][latIndex]))
wgts = np.sqrt(coslat)[..., np.newaxis]
solver = EofSolver(slpDJF, weights=wgts)

# extract eigenvalues
eigenValues = solver.eigenvalues()

# compute % contribution of each EOF
percentContrib = eigenValues*100./np.sum(eigenValues)

# compute the first three leading EOFs (EOFs 0, 1 and 2)
eofs = solver.eofs(neofs=3)
Numpy: Seasonal SLP Plot

First EOF, explains 53.8549% of seasonal DJF SLP variation

Second EOF, explains 16.1476% of seasonal DJF SLP variation

Third EOF, explains 11.4884% of seasonal DJF SLP variation
Numpy: Reading the Data Global Data

```python
ncin = Dataset('slp.mon.mean.nc', 'r')

lons = ncin.variables['lon'][:]
lats = ncin.variables['lat'][:]
time = ncin.variables['time'][:]

slp = ncin.variables['slp'][:]

ncin.close()
```
Numpy: EOF Solver

# Create an EOF solver to do the EOF analysis.
# Square-root of cosine of latitude weights are
# applied before the computation of EOFs.

coslat = np.cos(np.deg2rad(lats))
wgts = np.sqrt(coslat)[..., np.newaxis]
solver = EofSolver(slp, weights=wgts)

eof1 = solver.eofsAsCorrelation(neofs=1)
pc1 = solver pcs(npcs=1, pcscaling=1)
Numpy: SLP EOF1 Expressed as Correlation

EOF1 expressed as correlation

Correlation coefficient

-0.8, -0.4, 0.0, 0.4, 0.8
Numpy: SLP PC1 Time Series

PC1 Time Series

Time Range: 01 January 1948 --- 31 December 2012

Normalized Units

-2.0
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0
References I


