PyClimate 1.1. A set of C and Python routines for the analysis of climate variability

Jon Sáenz[†], Juan Zubillaga[‡], Jesús Fernández[§]

27th July 2001

[†]jsaenz@wm.lc.ehu.es, Departamento de Física Aplicada II, Facultad de Ciencias, Universidad del País Vasco, Apdo 644, 48080–Bilbao, SPAIN
[‡]wmpzuesj@lg,ehu.es, Departamento de Física de la Materia Condensada, Facultad de Ciencias, Universidad del País Vasco, Apdo 644, 48080–Bilbao, SPAIN
[§]chus@wm.lc.ehu.es, Departamento de Física de la Materia Condensada, Facultad de Ciencias, Universidad del País Vasco, Apdo 644, 48080–Bilbao, SPAIN

1 Change Log

- Version 1.1–June, 2001.

  1. Corrected bugs.
     (a) Previous version of `svdeofs()` used a denominator of (N-1) to compute the variance, whilst the data sets were standardized using the sample variance (N) in the denominator. Thus, PCs were actually scaled by the factor sqrt((N-1)/N) and eigenvalues (variances) by a factor (N-1)/N. We are using N in all denominators in this version.
     (b) readdat.py is able to handle comments like `#Variance of XXX` (without blank between the `#` and `Variance`), which caused a crash in previous versions.
  2. To be able to run properly under Python 2.n, all exception strings have been removed and converted to exception classes. All the exceptions are instances of a root class (PyClimateException) which can be printed with a very comprehensive message via the __str__ method. The exceptions have an alternative method GetExceptionValue(), which returns the most important data relevant to each exception.
  3. The differential operators have been extended. They can work using periodic and non periodic boundary conditions in the longitudinal direction. For latitudes, the previous approach (backward and forward second order finite difference schemes) is used at the borders. In the current version, they are able to work using 3D (Z,lat,lon) and 4D (time,Z,lat,lon) fields arranged according to COARDS conventions, not just 2D fields, like in PyClimate 1.0. This allows to compute the operation for all the records in a data set (if it fits into memory) or for all the levels in a record at a time.
  4. A new module `writedat.py` provides simple functions to write arrays in a way that can be directly inverted by `readdat.py`. Both modules

(writedat.py and readdat.py) have been included into asciidat.py, althought they can still be called independently for backward compatibility.

5. A new module bpcca.py provides Canonical Correlation Analysis with an EOF prefiltering (Barnett–Preisendorfer approach).
6. Easier interface for the routines which compute EOFs and the SVD decomposition of coupled fields. The input data sets can be arbitrarily shaped, with the only precondition that the leading axis is the time dimension. The fields are automatically reshaped inside the routines before returning. Similarly, the eigenvectors and singular vectors are automatically reshaped to the original shape before returning to the caller. Some simple functions have added to these modules (svd.py and svdeofs.py).
7. A new function getcoefcorrelations() has been added to svd.py to get the correlations of paired SVD expansion coefficients.
8. A new function pcseriescorrelation() has been added to the module svdeofs.py to get the correlation patterns between each principal component and the original field.
9. New versions of the manual and the tests cover the previously described changes.
10. We have slowly started to provide __doc__ strings to SOME (very few, yet) of the routines.
11. New function create_bare_COARDS() to create the bare minimum of a COARDS compliante netCDF file in ncstruct.py.

- Version 1.0–October, 2000.

1. Improved tests. They are more accurate, reliable, informative, comprehensive and use less disk space.
2. The package compiles using distutils. This feature has been checked on FreeBSD, Linux and OSF platforms.
3. Some minor typos corrected in the documentation.
4. Added KPDF.c, a extension module to estimate univariate and multivariate kernel–based probability density functions.
5. Added a class to compute the vertical component of the curl and the divergence of an horizontal vectorial field in diffoperators.py.
6. DCDFLIB.C is currently distributed with the package.
7. KZFilter.py has been converted into a general purpose LinearFilter.py which holds the basic operations of any linear filter. There are two different subclasses currently, the Lanczos filter [5] and the previous Kolmogorov–Zurbenko filter, KZFilter.py.

2 Installation

2.1 Compilation

Version 1.0 of **pyclimate** is distributed using distutils. Thanks to Alberto García for his help in setting up the setup.py script. After downloading the file PyClimate-1.0.tar.gz, you should decompress, untar and install it. The UNIX commands

needed are:
`gunzip PyClimate-1.0.tar.gz`
`tar xf PyClimate-1.0.tar`
After these steps, go into the main directory of **pyclimate** and a typical installation command is:

- For a general installation (root passwrod required):
  `python setup.py install`
- For a private installation:
  `python setup.py install home=~`

In this last case, you will have to set manually the value of the environmenal variable `PYTHONPATH` to be able to access **pyclimate**. The script `pyclimatetest.py` in the `test` directory runs a test that covers many of the routines in the package (see subsection 2.3).

2.2 Prerequisites

Many of the functions developed in this package have been built over other freely distributed packages and libraries. So, before installing this package, the next software is needed:

- **Python** interpreter itself, freely available at:
  http://www.python.org
- The Numeric extensions to **Python**, freely available at:
  http://numpy.sourceforge.net
- Konrad Hinsen's **Scientific Python**, available at:
  http://starship.python.net/crew/∼hinsen
- The **netCDF** library, freely available at:
  http://www.unidata.ucar.edu

The previous version required **DCDFLIB.C 1.1** library, too, which is freely available at: `http://odin.mdacc.tmc.edu/anonftp/page_3.html`. The current version of the package distributes the source and documentation of this library to ease the installation of **pyclimate**. In case the original C files of some of the extensions were edited, D. Beazley's **SWIG** (http://www.swig.org) will be needed to create the **Python** interface to the C routines. The **SWIG** source files (`*.i`) are distributed in the subdirectory `swig`.

2.3 Test

After installing the package, a user may be interested in knowing if it is working properly. There is a script which makes extensive use of the functions in **pyclimate** in the subdirectory `test`. This script (`pyclimatetest.py`) compares the results obtained after the installation with the ones obtained and checked during the development and extensive checking of the package. Thus, `python pyclimatetest.py` runs the tests and compares the output with the data stored in file `reference.cdf`. The output of the script is quite informative. Here is shown a excerpt of the output:

```
XG_44_44   |hgt_eofs       | : 1.00e-00 dG:-1.11022e-16
 G_45_45   |hgt_eofs       | : 1.00e+00 dG: 2.22045e-16
 G_46_46   |hgt_eofs       | : 1.00e+00 dG: 0.00000e+00
XG_47_47   |hgt_eofs       | : 1.00e-00 dG:-1.11022e-16
 RMS Error |hgt_lambdas    | : 0.00e+00
```

The first character of each file may be a blank (no difference detected) or a `X`, in which case, a difference has been detected. Most often, these differences are not important, and they are simply due to the fact that the control used is of the type `if rms!=0.0):`, which is not accurate enough. The second column shows which is the measure used to evaluate the differences. Usually, RMS errors are used, but congruence coefficients (`G_i_i`) are also used to evaluate errors in EOFs (and related quantities, like PCs, expansion coefficients for SVD analysis or heterogeneous and homogeneous correlation maps) due to the fact that both $\mathbf{x}$ and $-\mathbf{x}$ are valid solutions of the eigenvalue problem. The third column states the name of the netCDF variable which holds the reference data and the fourth column shows the value of the error measure (RMS or congruence coefficient). For the special case of congruence coefficients, a last column (`dG`) shows the difference $1 - |g_{i,i}|$ to let the user know the real difference for those lines with a `X` as the first character.
The new tests in version 1.1 have increased the size of reference.cdf to 1.6 Mb. Therefore, it is not currently distributed with `pyclimate`, and the distribution still fits in a floppy disk (or two, if you download the reference file and `gzip -9` it first). The test script can retrieve the file from the URL, and, additionally, the file can always be downloaded from the primary WEB server `http://www.pyclimate.org`. `python pyclimatetest.py -g` forces the retrieval of the reference file from its URL.

`python pyclimatetest.py` will usually be called without command line arguments, and it performs the comparisons. However, if a user detects that with her hardware or software she is getting different results (for instance, phases in EOFs) and she prefers to have her own `reference.cdf` file, she can use the command line option `-o` to overwrite the reference file. It is not advisable to do this unless the correctness of **pyclimate** has been examined at least once after the primary installation in that platform.

2.4 Examples

The previous manual included some examples on the use of the code. However, we have removed those examples from the current version of the manual to save some trees. They can be found at the directory `examples` of the distribution, and you can print them, if you don't mind using paper. Another different source of information on the use of the package can be found in the `pyclimatettest.py` file, which covers most of the functions in the package, using real–life data sets.

3 IO functions

3.1 ASCII files: `asciidat.py`

3.1.1 `readdat.py`

The main goal of file `readdat.py` is to provide a set of functions to handle IO from ASCII files. The ASCII files must be written in the way `gnuplot` files are, but not all of the `gnuplot`–compliant files are readable. Each row in the file is a row of the output data. Lines whose first token is a # are comments for these functions. No comments are allowed at the tail of a line. Empty lines are allowed, but they are discarded and not used to separate datasets in the file. Better support for these features exists in other packages for **Python**, like M. Haggerty's **Gnuplot** or the IO functions in K. Hinsen's **Scientific Python**. The interest of this file is that it allows to read files with complex numbers and other data types, a feature that is not supported by the mentioned files. Complex numbers are written as: `real,imag`, `,imag` or `real,` in the input file, with no blank spaces (nor tabs) separating the members of a complex number. Examples of use of these routines can be found in file `exiofuncs.py`

Function: def readdat(fname,typecode=None)

Reads all the data in the file **fname** and returns a bidimensional NumPy array of the same shape as the data in the original array and with a **typecode** given as `Float64` by default (`Complex64` in case there were complex elements in the file). If the parameter **typecode** is explicitly assigned, the output array is coerced to that type. If the file to be read contains only one row or column of data the returned array is one–dimensional.

Function: def readcol(fname,col=1,typecode=None)

In this case, the **col**–eth column is read and returned as a one dimensional NumPy array. The numbering of columns is not according to **Python**'s scheme, but to `gnuplot`'s one instead, that is, from 1 to N. The type coercion works as in the previous function.

Function: def readcols(fname,cols=[1],typecode=None)

Similar to `readcol`, but reads from the file **fname** and returns a n–dimensional NumPy array, where n is the length of the sequence object **cols**. The type coercion works as in the previous two functions. The columns are returned in the same order as they appear in **cols**.

3.1.2 `writedat.py`

As a `readdat.py` related tool is provided `writedat.py`. A NumPy array can be directly written in a way that can be recovered with `readdat.py`. This provides an easy way to pass arrays from one program to another with human readable intermediate files.

Function: def writedat(fname, matrix, header="", formatstring="%g ")

The name of the file to be created is passed as the first argument **fname**. If the file already exists it will be overwritten without prompting. One and two–dimensional arrays can be written and they are passed as the second argument **matrix** to the function. All other arguments are optional. A header can be specified, but the user must take care of writing a # at the beginning of each line. Finally, the output format can be controlled by passing a format string. The default format string is "%g ".

3.2 netCDF files

To access netCDF files, **pyclimate** uses the excellent netCDF package in K. Hinsen's **Scientific Python**. Two special functions have been built on top of K. Hinsen's interface to make the use of structured netCDF files easier, and they are included in the file `ncstruct.py`.

Function: def nccopystruct(name,inc,dims,vars,varcontents)

This function replicates the structure of a Conventions–compliant netCDF file (see COARDS conventions `http://www.unidata.ucar.edu`). These conventions usually imply the definition of a set of variables like latitude, longitude or vertical level and some auxiliary attributes like units or the numbers assigned to the missing values or scale and offset values for packing. This function helps in the creation of structured output files from an input dataset without having to individually copy ALL of the global attributes, auxiliary variables and their attributes. In case a `history` attribute existed, it is updated with the date and the name and arguments of the calling program. The function returns a netCDF file object ready to be used. This function opens the existing netCDF file with the 'w' flag, overwriting any existing file with the same name. Parameter **name** is the name of the output datafile to be created. **inc** is the input NetCDF file whose structure is to be replicated. **dims** is a sequence of dimensions of the input datafile which are to be copied onto the output datafile. **vars** is a sequence of variables to be copied onto the output datafile without copying the contents, just the declaration, type, structure and comments. **varcontents** is a sequence of variables in the parameter **vars** whose structure **and** contents are to be copied from the input dataset. The function `exiofuncsnc.py` shows examples of use of this function.

Function: create_bare_COARDS(ncname, tdat=None, zdat=None, latdat=None, londat=None)

This function creates a bare–minimum COARDS–compliant netCDF file. **ncname** is the name of the output file that is being created. **tdat** is `None` or a tuple with first element the NumPy array of items to be held (None for a record variable) and with second element in the tuple the units attribute of the created time variable. If **tdat** is None, no time variable nor dimension will be created. **zdat** is a tuple with two elements. The first one is the NumPy array of vertical levels. The second one is the units attribute of the vertical levels. If **zdata** is `None`, no vertical dimension is created. **latdat** is a NumPy array of latitudes. The units attribute is alwasy set to "degrees_north". If this parameter is `None`, no latitudinal dimension is created. **londat** is a NumPy array of longitudes. The units attribute is set to "degrees_east". If this parameter is `None`, no zonal dimension is created.

4 Time handling routines

One of the problems that researchers in climate analysis are usually faced to is the need to position their observations accurately in time. The file `JDTime.py` provides some functions to face this problem. It does not intend to provide a complete set of date handling routines. In particular, all the computations asume UTC time–zone and no DST functions are included (M.A. Lemburg's **mxDate** provides some of these extra features). `JDTime.py` allows fast, accurate and easy date processing from **Python** without the need of extra packages. A very interesting feature is the possibility of easily using monthly intervals by means of the function `monthlystep()`, computed from the definition of a tropical year with the same value used in Unidata's `udunits.dat`, so that the time values created this way can be correctly read and interpreted by the SDF interface in plotting programs as GrADS. The date arithmetic is carried out by means of Julian Days stored as double precision (Float64) numbers. Conversion functions to date structures are provided by means of a C extension based on astronomical algorithms [6]. The file `exjdtime.py` provides some examples of the use of these functions. The algorithm does not support dates before the start of the Gregorian calendar (1582-10-15). In particular, this means that you can not properly recover dates from some of the NCEP Reanalysis netCDF files, which start at 1–1–1, a date not included in the Gregorian calendar.

Class: JDTime

Integer members: **year**, **month**, **day**, **hour**, **minute**.

Double member: **second**.

The previous members can be directly accessed from **Python**.

Function: def date2jd(jdt)

From a `JDTime` structure **jdt**, get the Julian Day number as a double.

Function: def jd2date(jd,jdt)

Given a Julian Day **jd**, get the broken date structure **jdt**.

Function: def monthlystep()

Return the number of Julian Days in a month, that is, number of tropical days per year (365.242198781) divided by twelve. This provides adequate accuracy for dates during the instrumental part of the climate record. However, this approach poses a problem because of the truncation problem that appears due to the different length of the months. When using this approximation, some months are longer than others and this means that one of the time steps may be in the first day of one month and the last step on the last day of the same month (if month is longer than 30.43 days). To avoid this truncation problem, the origin should be set to day 15th of the starting month. This way, for every record, the first two fields of the date structure (year and month) will be adequately resolved by `jd2date()`. This can be easily explained with the next example, and its output.

```
from pyclimate.JDTimeHandler import *
from pyclimate.JDTime import *

a=JDTimeHandler("hours since 1958-01-01")
b=JDTimeHandler("hours since 1958-01-15")
mstep=monthlystep()*24
print mstep," in hours = ",mstep/24.," days"
print a.getdatefields(0.0,3)
print b.getdatefields(0.0,3)
print a.getdatefields(0.0+mstep,3)," Still in January"
print b.getdatefields(0.0+mstep,3)," In February"
```

```
730.484397562  in hours =  30.4368498984  days
[1958, 1, 1]
[1958, 1, 15]
[1958, 1, 31]  Still in January
[1958, 2, 14]  In February
```

There is a second file that allows the handling of COARDS compliant time variables in netCDF files, `JDTimeHandler.py`.

Class: JDTimeHandler
Function: def __init__(self,units)
The creator parses a COARDS–type units attribute of the netCDF *time* variable (ex: *hours since 1958-01-01 0:0*) and computes the offset and the scale factor to get Julian Days from the netCDF *time* variable for each record.
Function: def getdatefields(self,tvalue,listlen=6)
This method scales the value of **tvalue** using the scaling factor corresponding to the *units* attribute and adds the offset computed from the units attribute, getting the Julian Day corresponding to the time coordinate **tvalue**. The Julian Day value is used to create a broken date structure, and its values (year,month,day,hour,minute,second) are returned as a list, with as many members as those requested by **listlen**.
Function: def gettimevalue(self,datefields,listlen=6)
This function is the inverse of the previous one. It creates a Julian Day from the values in the list **datefields** (as many values as the value of **listlen**), computes the offset from the origin of coordinates and scales the value to the units corresponding to the *units* attribute of the *time* variable used to cretae the instance of this class. Due to some truncations which appear during the floating point operations, this function is only accurate to about 12 significant figures. So, if you are creating a netCDF file from scratch, it is more accurate and fast just to add equal time intervals using a fixed time step from record to record.

5 Distribution functions

DCDFLIB.C 1.1 is a library of C Routines for Cumulative Distribution Functions developed by Barry W. Brown, James Lovato and Kathy Russell, Department of Biomathematics, The University of Texas, M.D. Anderson Cancer Center. The library allows the determination of cumulative distribution functions, inverses, and the parameters of the following statistical distributions: Beta, Binomial, χ^2, Noncentral χ^2, F, Noncentral F, Gamma, Negative Binomial, Normal, Poisson, Student's t and Noncentral t. The file `pydcdflib.py` provides access to this library from **Python**. For each one of these distributions, a class has been created that wraps the parameters in the call to the functions of this routine and some C functions that wrap the original C calls to make them available to the **Python** interpreter. The same names have been used in the wrapping of the parameters, so that the original documentation of the **DCDFLIB.C** library is valid for each of the classes in this **Python** extension. The class is passed to a function with the same name of the original one (with `py` prepended to the name of the wrapper function). The fields of each class are updated by the wrapper functions. The code of the original C library is untouched. Examples on the use of these wrappers can be obtained from the `testDCDFLIB.py` script in the source directory of the distribution and from the enclosed `exdcdflib.py` file. There is an error when computing the parameters for the F distribution using the value `which=3`. This seems to be linked to the properties of the F distribution and, in case it were a bug, it also exists in the original **DCDFLIB.C**, as the same problem appears using the C file `testF.c`, provided in the source directory. The documentation of the original DCDFLIB.C library is distributed within the `doc` subdirectory of the distribution.

6 Simple multivariate statistical tools

The main goal of file `mvarstatools.py` is to provide some routines that are being used by others like `svdeofs.py`. However, some of them may be interesting by themselves, and they are thus documented in this section.

Function: def center(dataset)

Given an input multivariate dataset with arbitrary dimensions x_t (with the restriction that first dimension is time), this function returns the centered dataset $\tilde{x}_t = x_t - \mu$, with μ the sample average $\mu = \frac{\sum_t x_t}{N}$.

Function: def standardize(dataset)

Given an input multivariate dataset with arbitrary dimensions x_t (with the restriction that first dimension is time), this function returns the standardized dataset $\tilde{x}_t = \frac{x_t - \mu}{\sigma_x}$, with σ_x the sample standard deviation $\sigma_x = \sqrt{\frac{\sum_t (x_t - \mu)^2}{N}}$.

Function: def covariancematrix(X,Y)

Given two input datasets X and Y, with first dimension time and second dimension space, this function returns the S–mode covariance matrix $S = \frac{(X - \overline{X})^T (Y - \overline{Y})}{N}$.

Function: def correlationmatrix(X,Y)

Given two input datasets X and Y, with first dimension time and second dimension space, this function returns the S–mode correlation matrix $S = \frac{x^T x}{N}$ from the standardized x and y datasets.

Function: def detrend(dataset,tvalues,order=1)

From the **univariate** time series **dataset**, this function returns a tuple **(detrended,trendcoefs)**. The trend is computed adjusting by means of linear least squares a polinomial $x(t) = \sum_{i=0}^{M} a_i t^i$ to the data, where M is **order**. The tuple holds the detrended time series as the first member and the the trend coefficients a_i as the second element of the tuple.

Function: def congruence(pattern1,pattern2)

Given two spatial patterns **pattern1** ($\vec{e_1}$) and **pattern2** ($\vec{e_2}$), this function returns the congruence coefficient $g_{1,2} = \frac{\vec{e_1} \cdot \vec{e_2}}{|\vec{e_1}||\vec{e_2}|}$ [3, 13].

7 EOF analysis

A subroutine is provided in the file `svdeofs.py` to perform the decomposition of a dataset in its spatial eigenvectors (Empirical Orthogonal Functions, EOFs), the fraction of variance explained by each mode and its temporal expansion coefficients [11, 15] (PCs). One of the problems of this technique in climatological analysis is the singularity of covariance matrices when the number of spatial degrees of freedom is bigger than the number of temporal samples. To avoid this problem, the routine is based on the SVD decomposition of the data matrix $X = [x_{ij}]$. For this implementation, $i = 1 \ldots N$ means a temporal sample and $j = 1 \ldots M$ a grid point or observing site. Corrections derived from quadrature or area factors [2, 10] must be applied prior to this routine. There is an example on the use of these function in file `exsvdeofs.py`.

NEW: Since version 1.1, the input data set may be arbitrarily shaped, with the only restriction that the leading axis must be the time axis.

Function: def svdeofs(dataset, pcscaling=0)

Given a two dimensional dataset $N \times M$ as described above, this function returns a tuple **Z**,Λ,**E**. $\mathbf{Z} = [z_1|\ldots|z_L]$ is a $N \times L$ matrix, where each column z_k is a vector of length N and is the k–eth PC of the field. Each element λ_k in the one dimensional array Λ, is the variance accounted for by the k–eth component. Finally, the matrix $\mathbf{E} = [\mathbf{e}_1|\ldots|\mathbf{e}_L]$ has as columns each of the eigenvectors (EOFs) of the dataset. The maximum number of EOFs (L) will be $\min(M,N)$, depending on the rank of the data matrix. If the **pcscaling** is set to 0 the output eigenvalues are scaled as variances, the PCs z_k are scaled so that their variance is the variance accounted for by each component, and the eigenvectors are orthonormal, i.e. $< z_k, z_j >= \delta_{kj}\lambda_j$, $\mathbf{e}_k \cdot \mathbf{e}_j = \delta_{kj}$. If the **pcscaling** is set to 1 the output PCs have unit variance and the eigenvectors are no loger orthonormal but just orthogonal. The input dataset is centered (mean removed) inside the routine.

From version 1.1 `pyclimate` supports multidimensional **dataset** provided that the first dimension keeps being the temporal one. In that case the EOFs are returned stored as *generalized columns* (ie: they can be recovered as $E[...,eof\_number]$ instead of $E\ [\ :\ ,\ eof\_number])$ and their spatial shape is that of the original field. This approach is backwards compatible with previous versions but is more flexible for the user.

Function: def getgencol(a, ncol=0)
Returns the **ncol**-th *generalized column* of **a**, i.e. $a[...,ncol]$.

Function: def pcseriescorrelation(pcs, eofs, dataset)
Calculates the correlation between the **pcs** and the **dataset** time series at each grid point. The **eofs** must be supplied because it uses a calculus saving formula:

$$Corr(z_i, X_j) = \frac{\sigma(z_i)}{\sigma(X_j)} E_{ij}$$

where X_j is the time series at the j-th site. The correlation fields of the original time series with each PC are returned as *generalized columns*.

Function: def getvariancefraction(lambdas)
From the array Λ returned by the previous function (**lambdas**), returns the fraction of variance $f_i = \lambda_i / \sum_{k=1}^{L} \lambda_k$ associated to each of the eigenvalues.

Function: def bartletttest(lambdas,temporalsamples)
Test the hypothesis that the last $p-k$ eigenvalues are equal by means of the statistic [8]:

$$\chi^2 = -\nu \sum_{j=k+1}^{p} \ln(\lambda_j) + \nu(p-k) \ln\left[\frac{\sum_{i=k+1}^{p} \lambda_i}{p-k}\right] \tag{1}$$

It is distributed as a χ^2 with $\frac{1}{2}(p-k+1)(p-k+2)$ degrees of freedom. ν is the degrees of freedom of the covariance matrix. **lambdas** (λ_i) are the eigenvalues returned by `svdeofs` (array Λ) and **temporalsamples** is the number of temporal samples in the dataset. The function returns a tuple of arrays (each of them one item shorter than Λ), (χ_i,p_{χ_i}). The first element of the tuple is an array with the values of χ^2 obtained from eqn (1), while the second element holds their probabilities. This test is usually too liberal and tends to overfactor [8].

Function: def northtest(lambdas,temporalsamples)
Given as input the eigenvalues **lambdas** (Λ from `svdeofs`) and the number of samples in the dataset (**temporalsamples**), tests the degeneracy of eigenvalues according to $\delta\lambda_i \approx \frac{\sqrt{2}\lambda_i}{\sqrt{N}}$. It returns the array of values $\Delta\Lambda = \{\delta\lambda_i\}$. In case that for a pair of eigenvalues $\lambda_i - \delta\lambda_i \approx \lambda_{i+1} + \delta\lambda_{i+1}$, both members of the pair should be mantained or removed simultaneously in the factorization [10].

Function: def mctesteofs(dataset,eofs,subsamples,length)
This function tests, by means of a Monte Carlo test [3], the stability of the EOFs to

temporal subsampling of the original dataset. Given the original **dataset** and (possibly, some of) the EOFs of the whole dataset as returned by svdeofs() (**eofs**), this function builds several **subsamples** chosing the records in the original dataset at random. The subsamples are of length **length**, which must be smaller than the length of the original dataset. The function returns a Numeric array of shape `(subsamples, eofs.shape[-1]))` which holds the congruence coefficients of the master i(whole sample) and perturbed (subsample) eigenvectors for each subsample and eigenvector.

8 SVD of coupled datasets

The SVD decomposition of the covariance matrix of two datasets is one of the simplest, yet powerful, method to analyze the linear relationship between two coupled geophysical datasets [1, 4, 9, 15]. Some functions that implement most of the computations usually needed for this analysis are provided in the file `svd.py`. One example on the use of these functions is in the file `testsvd.py`.

Function: def svd(xfield,yfield)

Given two fields X (**xfield**) and Y (**yfield**), defined as in section 7 (first dimension of the array is the temporal one), this function returns the SVD decomposition of their covariance matrix $C_{xy} = \frac{X^T Y}{N}$. It returns a tuple of three arrays U, Σ and V. Each column $\mathbf{u}_i$ holds the i–eth singular vector of the left (X) field. Same convention is followed by the ordering of elements in V. Σ is a one–dimensional array which holds the singular values σ_i. The singular vectors are orthonormal ($\mathbf{u}_i \cdot \mathbf{u}_j = \mathbf{v}_i \cdot \mathbf{v}_j = \delta_{ij}$) and can be used to linearly project each of the fields to define the expansion coefficients $p_i(t) = x(t) \cdot \mathbf{u}_i$ and $q_i(t) = y(t) \cdot \mathbf{v}_i$ whose covariance is the same as the singular value associated to that mode $< p_i, q_i >= \sigma_i$. The datasets are centered (sample mean is removed) inside the function prior to the SVD computation. **NEW:** Since version 1.1, the data sets may have arbitrary dimensions, but the leading one must be time.

As in the case of EOF decomposition the fields can also be entered as more–than–two–dimensional arrays and the returned matrices will be accordingly reshaped to avoid the user to care about the internal use of two–dimensional matrices. The SVD patterns are returned as *generalized columns* in the sense described in the EOF section.

Function: def getcoefs(data,svectors)

Given a dataset **data** (X or Y) and the corresponding set of singular vectors (U or V) returned by `svd()`, this function returns the expansion coefficients (p_i or q_i) up to the order stated by the evaluation of `svectors.shape[-1]))`. This, in fact, means truncating the representation of the field at this number. This same approach is used in the next two routines. The output Numeric arrays hold the coefficients in each column. That is, the expansion coefficients associated to the `i`–eth singular vector are: `coefs[:,i]`.

Function: def getcoefcorrelations(scoefs, zcoefs)

Given the left and right SVD expansion coefficients returns an one–dimensional array containing the paired correlation of the corresponding coeffients.

Function: def homogeneousmaps(data,svectors)

From the input dataset **data** and the array **svectors**, returns an array with as many homogeneous correlation maps ($< X, \mathbf{u}_i >$ or $< Y, \mathbf{v}_i >$) as ending subarrays (singular vectors) are in **svectors**. Each subarray `themaps[...,imap]` is the `imap`–eth map.

Function: def heterogeneousmaps(xdata,ycoefs)

From the input dataset **xdata** and the array of expansion coefficients **ycoefs**, it returns an array with as many heterogeneous correlation maps ($< Y, p_k >$ or $< X, q_k >$) as columns (expansion coefficients) are in **ycoefs**. The ordering of the maps is as described in previous routine.

Function: def SCF(sigmas)

This function returns the squared covariance fraction associated to each singular value, $\mathrm{scf}_k = \frac{\sigma_k^2}{\sum^K \sigma_k^2}$ from the array Σ returned by `svd()`.

Function: def CSCF(sigmas)

This function returns the cumulative squared covariance fraction associated to a set of singular values, $\mathrm{cscf}_k = \sum_{i=0}^{k} \frac{\sigma_i^2}{\sum^K \sigma_k^2}$, from the array Σ returned by `svd()`.

Function: def makemctest(Um,Vm,ldata,rdata,itimes,ielems)

This function performs a Monte Carlo test on the stability of the left singular vectors **Um** and the right singular vectors **Vm** obtained from `svd()` or truncated to the leading ones (`Um.shape[-1]==Vm.shape[-1]`). It computes **itimes** simultaneous subsamples (**ielems** long each) of each datafield **ldata**, **rdata**, and evaluates the congruence coefficient of the master singular vectors to those derived from the subsamples. It returns a tuple of Numerical Python arrays `(ucoefs,vcoefs)` which hold the congruence coefficient for each sample and each U/V master singular vector (`ucoefs.shape=(itimes,Um.shape[-1])`). **ielems** must be smaller than `len(ldata)`.

9 CCA (Canonical Correlation Analysis)

Canonical correlation analysis can be performed by `bpcca.py`. This technique gets the pattern decomposition of two fields such that the correlation between the temporal coefficients of paired patterns is maximum. The Barnett and Preisendorfer [1] approach has been implemented. In this approach an EOF prefiltering is performed and the CCA is carried out in EOF–coordinates. The user never deals with these coordinates but can customize some features of the EOF decomposition.

The results are scaled for the canonical expansion coefficients to have unit variance. If the columns of A and B are the canonical expansion coefficients and P and Q are the canonical patterns, the fields S and Z can be partially (due to the EOF prefiltering) recovered as:

$$\widetilde{S} = AP^T$$
$$\widetilde{Z} = BQ^T$$

The internal calculus are carried out by means of singuar value decomposition of the principal component cross covariance matrix.

Class: bpcca

Function: def __init__(sfield, zfield, retainedeofs=None)

Given two two–dimensional datasets this constructor performs the CCA decomposition of the fields. **sfield** and **zfield** are the left and right fields respectively. The user can specify the number of EOFs to be retained in each field as a tuple **retainedeofs**. If no **retainedeofs** is specified 70% rule is used: EOFs are retained until a 70% of the variance is explained.

Function: def leftpatterns()

This method returns an array which columns are the left (those associated to the **sfield**) canonical patterns.

Function: def rightpatterns()
Idem for the right patterns.
Function: def leftexpcoeffs()
This method returns an array which columns are the left (those associated to the **sfield**) canonical canonical expansin coefficients.
Function: def rightexpcoeffs()
Idem for the right coeffiients.
Function: def correlation()
Gives the ordered canonical correlations of the expansion coefficients as an array.
Function: def getvariancefractions()
Returns a tuple containing the variance fraction explained by each canonical pattern. The first element of the tuple is the array of fractions corresponding to the left field. The second one is that of the right field.

10 Multivariate linear filters: `LinearFilter.py`

Filters are used in climate analysis to atenuate components of irrelevant frequencies from a broadband signal and to select the frequencies of interest for the problem under consideration. The simplest of these filters are the linear filters and two of them are provided: the Kolmogorov-Zurbenko filter and the Lanczos filter. In the current version, the filter does not handle missing values.
Both of them filter the data by weighting the input with coefficients a_k that depend on the filter:

$$y_t = \sum_{k=-n}^{n} a_k x_{k+t}$$

This common part of the computation is carried out by subclassing the **LinearFilter** class.
Class: LinearFilter
Method: LinearFilter.getfiltered(ifield)
Given an input field, the function returns the filtered data corresponding to each input record. This function must be called in an iterative way. The filter has been implemented with performance in mind, so that, in order to avoid several iterations through a possibly long dataset, it iterates just once, performing a multivariate filtering to each record. This makes that during the initialization phase, the internal running buffer must be initialized (filled) until valid values can be obtained. During the initialization phase of the filter, this function returns `None`. After the internal buffer is ready, the function returns valid filtered fields until the user arrives at the final record of the iteration.
Method: LinearFilter.reset()
This method resets the linear filter so that it is able to start computing with a data set of a different shape avoiding the creation of a new instance of the class.

10.1 KZ filter: `KZFilter.py`

The Kolmogorov–Zurbenko digital filter is a very simple but powerful digital filter. Its features are documented elsewhere [7, 12].
Class: KZFilter
Method: KZFilter.__init__(self,points,iterations,lowpass=1)
Default (and only) constructor. **points** is an odd integer representing the number of

points in the moving average, while **iterations** is an integer representing the number of times that the average is iterated to get a better damping of the secondary maximums. When the flag **lowpass** is false, the filter instance works as a high–pass filter.
Method: KZFilter.getcoefs(self)
This function returns the coefficients that are being used in the filter.
Method: KZFilter.cutofffrequency()
Returns the approximate cutoff frequency of the filter.

10.2 Lanczos filter: `LanczosFilter.py`

The Lanczos filter is another very common filter and its mathematical properties and the algorithms for the compilation of the coefficients can be found for instance at [5].
Class: LanczosFilter
Method: LanczosFilter.__init__(self,filtertype='bp', fc1=0.0, fc2=0.5, n=None)
Default constructor. **filtertype** specifies if the filter is low ('lp'), high ('hp') or band ('bp') pass. The cut-off frequencies for the bandpass case are set in **fc1** and **fc2**. If the **filtertype** is set to 'lp' or 'hp' both cut-off frequencies must be equal. Band–pass filter will be assumed if the frequencies are different, regardless you have specified other type of filter!
The number of different coefficients to be calculated is set with the parameter **n**. Even though it has a default value that is appropiate in the band–pass case, it is strongly recommended not to use the default value in any other case (high–pass or low–pass) [5].
Method: LanczosFilter.getcoefs(self)
As in section 10.1 this function returns the coefficients that are being used in the filter.

11 Differential operators: `diffoperators.py`

Some routines to compute differential operators on the sphere specially suited for climate analysis are provided. By now, they only process regular latitude–longitude grids. The operators make the computations by means of centered differences, except at the borders, where they use backward or forward differences. Periodic boundary conditions are supported in the zonal axis. All the operators assume that the input fields are organized according to the COARDS scheme ([latindex,lonindex]), with other dimensions (time, vertical level) prepending the latitude and longitude axis of the arrays.
Class: HGRADIENT
Method: HGRADIENT.__init__(self,lats,lons,asdegrees=1,PBlon=0)
This is the constructor for the horizontal gradient calculator. **lats** is a NumPy array which holds the latitudes corresponding to the grid points, **lons** holds the longitudes of the grid points and if the optional parameter **asdegrees** is set to true, it means that the input angular variables are expressed as degrees, not radians. **PBlon** is true if periodic boundary conditions are to be applied in the longitudinal axis. In this case, second order centered differences are used in the whole zonal domain.
Method: HGRADIENT.hgradient(self,hfield,R=6.37e6)
Given as input the horizontal field **hfield** (Φ), get as a result a tuple which holds the zonal and meridional components of the gradient on the sphere ($\frac{1}{R\cos\varphi}\frac{\partial\Phi}{\partial\lambda}$,$\frac{1}{R}\frac{\partial\Phi}{\partial\varphi}$). **R** is assigned by default the radius of the Earth.
Class: HDIVERGENCE
Method: HDIVERGENCE.__init__(self,lats,lons,asdegrees=1,PBlon=0)
Default constructor, with parameters defined as for the gradient operator.

Method: HDIVERGENCE.hdivergence(u,v,R=6.37e6)
From the zonal (**u**) and meridional (**v**) components of a vectorial field, find the horizontal divergence $\frac{1}{R\cos\varphi}\left[\frac{\partial u}{\partial\lambda}+\frac{\partial(v\cos\varphi)}{\partial\varphi}\right]$. The function returns a bidimensional array with the divergence field.
Class: VCURL
Method: VCURL.__init__(self,lats,lons,asdegrees=1,PBlon=0)
Default constructor, with parameters defined as for the gradient operator.
Method: VCURL.vcurl(u,v,R=6.37e6)
From the zonal (**u**) and meridional (**v**) components of a vectorial field, find the vertical component of the curl operator on the sphere, $\frac{1}{R\cos\varphi}\left[\frac{\partial v}{\partial\lambda}-\frac{\partial(u\cos\varphi)}{\partial\varphi}\right]$. The function returns a scalar field with the vertical component of the curl of the vectorial field.

12 Multivariate and Univariate Kernel based Probability Density Function Estimation: `KPDF.c`

Histograms are useful and simple estimators for the Probability Density Function (PDF) associated with physical or modelling experiments. However, the estimation of PDFs by means of histograms presents some drawbacks like the instability to the selection of the bins (bin number and bin origin) or the lack of derivability. Thus, it has been proposed [14] the use of a kernel based approach to the estimation of the PDF. This module provides functions to perform univariate and multivariate PDF estimation. The relevant equations for this module are (2) for univariate density estimation and (3) for multivariate series.

$$f(x)=\frac{1}{nh}\sum_{1=1}^{n}K\left(\frac{x-X_i}{h}\right), \tag{2}$$

$$f(\mathbf{x})=\frac{\det S^{-\frac{1}{2}}}{nh^d}\sum_{1=1}^{n}K\left(h^{-2}(\mathbf{x}-\mathbf{X}_i)^T S^{-1}(\mathbf{x}-\mathbf{X}_i)\right), \tag{3}$$

with n the number of points, h a scalar which holds the bandwidth used in the estimation, X_i and $\mathbf{X}_i$ the experimental points, x and $\mathbf{x}$ the points where the PDF is being estimated, K is the function used as the kernel and S is the scaling matrix which can optionally be used in the multivariate estimators, usually, the covariance matrix.
Function: def UPDFEpanechnikov(edata,gdata,h)
Returns a NumPy array with the PDF associated to the experimental data **edata** evaluated at points **gdata** (NumPy array, too) using a bandwidth **h**. For this and next functions, **edata** and **gdata** must be NumPy arrays and the function returns a NumPy array of typecode Float64. The kernel used in the estimation is a Epanechnikov one. It raises `TypeError` if **edata** or **gdata** is not a 1-D NumPy array It raises `MemoryError` if it is unable to allocate the returned array or some temporal arrays which are needed as working space.
Function: def UPDFBiweight(edata,gdata,h)
Same as the previous function, but use a biweight kernel, instead (S86, Table 3.1). It raises `TypeError` if **edata** or **gdata** is not a 1-D NumPy array It raises `MemoryError` if it is unable to allocate the returned array or some temporal arrays which are internally needed as working space.
Function: def UPDFTriangular(edata,gdata,h)
Same as the previous function, but using a triangular kernel function (see S86, Table

3.1) It raises `TypeError` if **edata** or **gdata** is not a 1-D NumPy array It raises `MemoryError` if it is unable to allocate the returned array or some temporal arrays which are internally needed as working space.

Function: def UPDFOptimumBandwidth(edata)

Get an approximation to the optimum bandwidth by means of equation 3.28 (S86). Unless there is a better suggestion, it can be a good starting point. **edata** is, as previously, a NumPy array (internally works as Float64). It raises `ValueError` if **edata** is not a source sequence.

Function: def MPDFEpanechnikov(edata,gdata,h,Sm1=None,K=0.0)

Given a multidimensional (dimension $p \in [2,3]$ currently) experimental data set **edata** (NumPy array), evaluate the PDF at each p-dimensional point in the NumPy array **gdata**, using a radially symmetric bandwidth h. The optional parameter **Sm1** is a $p \times p$ matrix (NumPy array, too) which can be used to scale each of the directions in the **edata** and **gdata** arrays. Usually, the inverse of the covariance matrix of **edata** is a good selection for this parameter. It can also be `None`, in which case, no scaling is applied to the directions in the p–dimensional space. **K** is a renormalizing constant which must be computed according to the scaling defined by **Sm1**. If the inverse of the covariance matrix is selected as **Sm1** and the square root of the determinant of the covariance matrix is input as **K**, then, a Fukunaga estimator is being used in the estimation (S86, Eq. 4.7). This function uses a multidimensional Epanechnikov kernel (S86, Eq. 4.4) for the estimation. It raises `ValueError` if **edata** or **gdata** are not NumPy arrays. It raises `TypeError` if **h** is not a number. It raises `ValueError` if the dimensionality of the data sets **edata** and **gdata** is not correct because they exceed the implemented capability, they are not "compatible" (for instance, one is 2-D and the other one 3-D), or the scaling matrix is not of the adequate range ($p \times p$).

Function: def MPDFGaussian(edata,gdata,h, Sm1=None, K=0.0)

Same as MPDFEpnechnikov, but using a Gaussian multivariate kernel (S86, Eq. 4.3). It raises `ValueError` if **edata** or **gdata** are not NumPy arrays. It raises `TypeError` if **h** is not a number. It raises `ValueError` if the dimensionality of the data sets **edata** and **gdata** is not correct because they exceed the implemented capability, they are not "compatible" (for instance, one is 2-D and the other one 3-D), or the scaling matrix is not of the adequate range ($p \times p$).

Function: def MPDFOptimumBandwidth(edata)

Provides a first guess to the optimum bandwidth (S86, Eq. 4.14). It raises `ValueError` if **edata** is not a source sequence or if its dimensions exceed current capabilities.

Function: def MPDF2DGrid2Array(xdata,ydata,fastestdim=0)

This function returns an array of shape `(len(xdata)*len(ydata),2)` which holds in a linear fashion the grid defined by arrays **xdata** and **ydata**. The optional parameter **fastestdim** allows the selection of the dimension which grows faster (outer one by default) or inner one if it is non zero.

This functionality can be obtained by means of other standard Python techniques, like using complex calls to `replicate()`, `concatenate()` and `map()`, or the new feature of *List comprehension* in Python 2.0. However, this function is much more efficient. **Function: def MPDF3DGrid2Array(xdata,ydata,zdata,fastestdim=0)**

This function s similar to `MPDF2DGrd2Array` in the sense that it returns an array shaped like `(len(xdata)*len(ydata)*len(zdata),3)` which holds in a linear fashion the grid defined by arrays **xdata**, **ydata** and **zdata**. The optional parameter **fastestdim** allows the selection of the dimension which grows faster (outer one by default) or inner one if it is non zero.

13 Other links

There are other packages that can be combined with **Python**, **pyclimate**, and the other packages that have been mentioned previously and which allow to easily build powerful data analysis tools and programs. To name a few, for graphics, for instance, there exists the posibility of using **DISLIN** (http://www.linmpi.mpg.de/dislin/), free for some platforms. There is another set of statistical functions by G. Strangman (http://www.nmr.mgh.harvard.edu /Neural_Systems_Group/gary/python.html). A complete repository of **Python** programs can be found at the official WEB server (http://www.python.org) or Starship, (http://starship.python.net) the server for the community of **Python** contributors and developers.

Special mention deserve two other **Python** packages designed to work with climate data. One of them is CDAT, developed by the Lawrence Livermore National Laboratory (`http://sourceforge.net/projects/cdat`), and the other one is mtaCDF, developed at the Institüt für Meereskunde, Kiel (`http://www.ifm.uni-kiel.de-/to/sfb460/b3/Products/mtaCDF.html`).

References

[1] C. S. Bretherton, C. Smith, and J. M. Wallace. An intercomparison of methods for finding coupled patterns in climate data. *J. Climate*, 5:541–560, 1992.

[2] C. E. Buell. The number of significant proper functions of two–dimensional fields. *Journal of Applied Meteorology*, 17(6):717–722, 6 1978.

[3] X. Cheng, G. Nitsche, and J. M. Wallace. Robustness of low–frequency circulation patterns derived from EOF and rotated EOF analyses. *J. Climate*, 8:1709–1713, 1995.

[4] S. Cherry. Singular value decomposition analysis and canonical correlation analysis. *J. Climate*, 9:2003–2009, 9 1996.

[5] C. E. Duchon. Lanczos filtering in one and two dimensions. *J. Appl. Met.*, 18:1016–1022, 1979.

[6] P. Duffet-Smith. *Practical Astronomy with your Calculator*. Cambridge University Press, 3 edition, 1990.

[7] R. E. Eskridge, J. Y. Ku, S. T. Rao, P. S. Porter, and I. G. Zurbenko. Separating different scales of motion in time series of meteorological variables. *Bulletin of the American Meteorological Society*, 78(7):1473–1483, July 1997.

[8] J.E. Jackson. *A User's Guide to Principal Components.* Wiley Series in Probability and Statistics. Applied Probability and Statistics. John Wiley & Sons., 1991.

[9] M. Newman and P.D. Sardeshmukh. A caveat concerning singular value decomposition. *J. Climate*, 8:352–360, 1995.

[10] G.R. North, T.L. Bell, R.F. Cahalan, and F.J. Moeng. Sampling errors in the estimation of empirical orthogonal functi ons. *Monthly Weather Review*, 110:699–706, July 1982.

[11] R. Preisendorfer and C.D. Mobley. *Principal Component Analysis in Meteorology and Oceanography*. Developments in Atmospheric Science. Elsevier, Amsterdam, 1988.

[12] S. T. Rao, I. G. Zurbenko, R. Neagu, P. S. Porter, J. Y. Ku, and R. F. Henry. Space and time scales in ambient ozone data. *Bulletin of the American Meteorological Society*, 78(10):2153–2166, October 1997.

[13] M. B. Richman and P. J. Lamb. Climatic pattern analysis of three– and seven–day summer rainfall in the Central United States: Some methodological considerations and a regionalization. *J. Climate*, 24:1325–1343, 1985.

[14] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, Londres, 1986.

[15] H. von Storch and A. Navarra, editors. *Analysis of Climate Variability. Applications of Statistical Techniques*. Springer, Berlin, 1995.