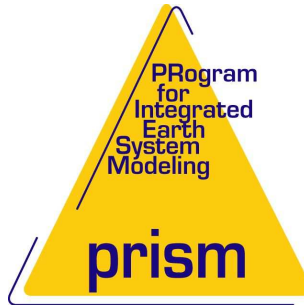


**PRISM**  
**Project for Integrated Earth System Modelling**  
**An Infrastructure Project for Climate Research in Europe**  
**funded by the European Commission**  
**under Contract EVR1-CT2001-40012**



**The COCO Processing library**

*Edited by:*  
*Rosalyn Hatcher*

PRISM-Report Series-17

1st Edition  
(last change: December 17, 2004)

## Copyright Notice

© Copyright 2003 by PRISM

All rights reserved.

No parts of this document should be either reproduced or commercially used without prior agreement by PRISM representatives.

## How to get assistance?

The individual work packages of the PRISM project can be contacted as listed below.

PRISM publications can be download from the WWW server of the PRISM project under the URL: <<http://prism.enes.org/Results/Documents/>>

## Phone Numbers and Electronic Mail Adresses

Electronic mail addresses of the individual work packages are composed as follows :

**prism\_ *work package* @prism.enes.org**

Name	Phone	<i>PRISM Work Package</i>
Rosalyn Hatcher		<i>wp4a</i>
Mick Carter		<i>wp4a</i>

# Contents

<b>1</b>	<b>The COCO Processing Library</b>	<b>1</b>
1.1	introduction . . . . .	1
1.2	Detailed documentation of COCO methods . . . . .	2
1.2.1	collapse() . . . . .	2
1.2.2	textract() . . . . .	4
1.2.3	getTimeComponents() . . . . .	5
1.3	Tutorials and Examples . . . . .	5
1.4	Links to reference materials . . . . .	6
1.5	Developer Documentation . . . . .	6
1.5.1	Download Information . . . . .	6
1.5.2	Installation, portability and local configuration . . . . .	7
1.5.3	Unit Tests . . . . .	7
1.6	Design notes and diagrams . . . . .	8
1.7	Notes for developers . . . . .	9
1.7.1	Known Problems . . . . .	9
1.7.2	Where to look to extend functionality . . . . .	9
1.7.3	Recommendations . . . . .	9



# List of Figures

1.1 COCO Architecture . . . . . 8



# List of Tables



# Chapter 1

## The COCO Processing Library

### 1.1 introduction

COCO (CDMS overloaded for CF Objects) is a data processing library. It is written in Python and is based on the Climate Data Management System (CDMS). CDMS is an object-oriented data managed system, specialized for organizing multidimensional, gridded data used in climate analysis and simulation.

The basic unit of computation in CDMS is the variable. A variable is essentially a multidimensional data array, along with its metadata describing the domain, grids and other attributes. As a data array, a variable can be sliced to obtain a portion of the data, and can be used in arithmetic computations. (For further in-depth discussion of CDMS see the CDAT homepage: <http://esg.llnl.gov/cdat/>)

See the Developer Documentation (Section 1.5) for instructions on how to install COCO. To use COCO, first start up the python interpreter and then import the 'cdms' module. This then provides access to all the cdms methods and those in the COCO layer.

```
$ python
>>> import cdms
>>>
```

In the COCO library, data and associated metadata belong together. They are read in from disk, manipulated and written out in conjunction. All input data files processed with COCO should adhere to the CF convention. If the files do not fully adhere to the convention then errors may occur, and the associated metadata may not be updated correctly. Any data written out to files will be CF compliant.

To check that any netCDF files are CF compliant, use the CF checker. This is available for download at

<http://prism.enes.org/WPs/WP4a/ProcessingLib/index.html>

or use it directly via the website

<http://titania.badc.rl.ac.uk/cgi-bin/cf-checker.pl>

COCO enhances the CDMS functionality to include manipulation of the metadata. Additionally, COCO introduces the following functionality:

**collapse(method=method, axis=axis, weights=weights, tolerance=tolerance)** Applies the chosen statistical methods to any combination of axes, with or without weights, and with or without a user specified tolerance to missing data. The methods implemented are: mean, mid-range, median, max, min, sum, standard\_deviation, variance, covariance, correlation.

**textract(\*args, \*kwargs)** Enables a time slice of data to be extracted from a single variable. For example, to extract all January's data for all years.

**getTimeComponents()** Returns a dictionary of date-time structures. For example, all the years, months, days, hours, minutes and seconds.

Full documentation for these three functions follows.

## 1.2 Detailed documentation of COCO methods

### 1.2.1 collapse()

Collapse is the function that interfaces with the statistical methods.

#### Usage

```
result = var.collapse(method=methodopts, axis=axisopts,
                      weights=weightopts, tolerance=toleranceopts)
```

#### Options

methodopts:

'mean'|'median'|'max'|'min'|'mid\_range'|'sum'|'sd'|'variance'|'covar'|'corr'

axisopts: 'x'|'y'|'z'|'t'|'(dimension\_name)'|0|1...|n

The name of the dimension or index (integer value 0..n) over which you want to compute the statistic.

weightopts:

default = None returns equally weighted statistic.

If you want to compute the weighted calculation, provide an array of weights (of the same shape as the dimensions being collapsed over or the same shape as var) here. Alternatively, specify a weighting type appropriate to the dimensions being collapsed over:

't'|'z'|'y'|'x'|'northward\_distance'|'eastward\_distance'|'area'|'simple'.

If the collapse is over more than one axis then a weighting option must be supplied for each dimension.

toleranceopts:

default value = 1 No tolerance of missing data.

Set to a fraction between 0.0 and 1.0 to specify the minimum weighted fraction of a collapsed cell which must have valid data in order for a representative value to be calculated.

#### Examples:

```
$ python
Python 2.2.2 (#1, Feb 26 2004, 11:13:44)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cdms
>>> # Read in the variable 'tap' from a file
>>> tap=cdms.open('model.nc').getVariable('tap')
>>> # Calculate an area average lat/lon mean
>>> mean=tap.collapse(method='mean',axis='yx',weights='area')
>>> mean[0].shape
(22, 9, 1, 1)
>>> print mean[0].getValue()[0,0]
[ [219.071286768 ,]]
```

```

>>> # Open output file and write out the result
>>> out=cdms.open('/home/hadro/temp/out.nc','w+')
>>> out.write(mean[0])
<Variable: tap, file: /home/hadro/temp/out.nc, shape: (22, 9, 1, 1)>
>>> out.close()

>>> # Read in one variable from a file that contains many and perform
>>> # standard deviation over the time axis
>>> tas=cdms.open('abefda.psjldjf.nc').getVariable('tas')
>>> tas
<Variable: tas, file: abefda.psjldjf.nc, shape: (1, 73, 96)>
>>> std=Collapse([tas],method='sd',axis='t')
>>> std[0].shape
(1, 73, 96)

```

If you wish to perform the same calculation on several variables of the same dimensionality, you can do so with just one call to collapse. Instead of accessing collapse as a method of the variable, import the dataoperations module and pass in the list of variables over which to calculate the statistic.

```

>>> import cdms,cdms.dataoperations
>>> # Read in several variables and perform a mean over y axis with
>>> # latitude weighting.
>>> # Note: to process several variables at once, they must all
>>> # have the same dimensionality.
>>> vars=cdms.open('abefda.psjldjf.nc').getVariables()
>>> vars
[<Variable: tasmax, file: abefda.psjldjf.nc, shape: (1, 73, 96)>,
<Variable: tasmin, file: abefda.psjldjf.nc, shape: (1, 73, 96)>,
<Variable: tas, file: abefda.psjldjf.nc, shape: (1, 73, 96)>]
>>>
>>> mn=cdms.dataoperations.Collapse(vars,method='mean',\
    axis='(latitude)',weights='latitude')
>>> len(mn)
3
>>> mn[0].shape
(1, 1, 96)
>>> mn[0].getValue()[0,0,0]
281.58047358194989
>>> mn[1].getValue()[0,0,0]
276.36157311333551
>>> mn[2].getValue()[0,0,0]
278.86281840006512
>>>

```

More about specifying weights:

```
>>> tap.collapse(method='mean',axis='yx',weights='x')
```

is invalid since it doesn't say how to weight y.

```
>>> tap.collapse(method='mean',axis='yx',weights='area')
```

is valid as weights are xy area weighted.

Currently, only a limited selection of weights have been implemented. They are:

- axis='x', weights='x' - weighting by difference betw. the bounds

- axis='y', weights='y'
- axis='z', weights='z'
- axis='t', weights='t'
- axis='yx', weights='area' - lat-lon area weighting
- axis='yx', weights='simple'
- axis='y', weights='northward\_distance' - weighted by cell-widths in m

Other Useful Notes:

- See further documentation of the collapse method in the file `cdms/dataoperations.py`. The documentation can be viewed by running:  

```
pydoc dataoperations.py
```
- Whether collapse is called using `(tap.collapse(...))` or `cdms.dataoperations.collapse([tap],...)` the result will always be a list of cdms variables. Even if the result is just one variable it will be as a list containing one variable. See first example above.

## 1.2.2 textract()

Reads in a slice of data, returning a transient variable.

### Usage:

```
result=var.textract(*args, year=None, month=None, day=None)
```

### Options:

Optional Input:

'args' is an argument list of conditional strings. If this string is supplied, textract will append to it, starting with 'and'. Otherwise, the resulting search expr is purely the result of the keywords. If more than one string is supplied they will be concatenated together with 'and'.

Input Keywords:

The optional keyword arguments 'year', 'month', 'day' may be used to specify specific search criteria. For example to extract specific years, etc. If any of the above keywords is a list, this is by default converted into a list of OR'ed conditions.

### Examples:

```
sp.textract(year=1990,month=[11,12])
```

extracts data satisfying the search expression:

```
year = 1990 and (month = 11 or month = 12)
```

```
sp.textract('year < 1990', month = 12)
```

extracts data satisfying the search expression:

```
year < 1990 and month = 12
```

```
sp.textract(year=[1980,1990],month=[1,2,3])
```

extracts data satisfying the search expression:

```
(year = 1980 or year = 1990) and (month = 1 or month = 2 or month = 3)
```

```
sp.textract('year >= 1980', 'year < 1990')
```



```
>>> o = cdms.open('out.nc','w')
>>> o.write(tap)
<Variable: tap, file: /home/hadro/temp/out.nc, shape: (22, 9, 73, 96)>
>>> o.close()
```

To determine the shape of the data array:

```
>>> tap.shape
(22, 9, 73, 96)
```

Applying a mask from one file to a variable in another file and calculating an annual mean.

```
import cdms,MA
# Open file read
f=cdms.open('ustrw_e63_all.nc')
variab=f('ustrw')
data=MA.array(variab.getValue())
# Open and read in variable containing mask
maskVar=cdms.open('slm_e63_all.nc','r').getVariable('slm')
mask=MA.array(maskVar.getValue())
# Create new masked data array
new=MA.masked_array(data,mask=mask)
# Put the new data back into the variable
variab.putValue(new)print variab.getValue()[1,0,0]
# Annual mean
import cdms
annual_mn=variab.collapse(method='mean',axis='t')
```

## 1.4 Links to reference materials

NetCDF(CF) website

<http://www.cgd.ucar.edu/cms/eaton/cf-metadata/>

CDAT homepage

<http://esg.llnl.gov/cdat/>

Documentation:

<http://esg.llnl.gov/cdat/documentation.html>

## 1.5 Developer Documentation

### 1.5.1 Download Information

All source code for the processing library developed under the PRISM project is available from the PRISM cvs server [bedano.cscs.ch](http://bedano.cscs.ch).

Once logged onto the PRISM cvs server using ssh, you can download COCO by running the following commands:

```
$ export CVS_RSH = /path/to/ssh
$ export CVSROOT = :ext:cvs@bedano.cscs.ch:/users/cvs
$ cvs checkout PRISM_Data
```

This will download the UDUNITS package and all the COCO code. Please follow the steps below to install.

## 1.5.2 Installation, portability and local configuration

In order to use the CF Data Processing Library the following are prerequisites:

1. CDAT-4.0
2. Either the python distribution packaged with CDAT or Python2.3 or later if CDAT installed using independent distributing of python.
3. UDUNITS with the python interface The Unidata units library, `udunits`, supports conversion of unit specifications between formatted and binary forms, arithmetic manipulation of unit specifications, and conversion of values between compatible scales of measurement. This package, complete with the python interface, is available from the PRISM cvs server.

Installing the CF Data Processing Library:

Once the CF Data Processing code has been downloaded; carry out the following steps:

1. Go to directory `cf_data_processing`
2. Replace the `cdms` python (`.py`) files (`site-packages/cdms/*.py`) with those in directory `cf_data_processing/cdms`
3. Copy the CF Standard Name table `cf_data_processing/cdms/standard_name.xml` to `site-packages/cdms` directory.
4. Copy directory `oo` and file `oo.pth` to `site-packages`.
5. Replace the `genutil` python (`.py`) files (`site-packages/genutil/*.py`) with those in directory `cf_data_processing/genutil`.

## 1.5.3 Unit Tests

Once the package has been installed it is recommended that the unit tests are run. The unit tests are in the directory `cf_data_processing/tests/functional`.

To run these tests issue the following command from within the directory:

```
$ python dataopsunittests.py
```

A typical output will be of the form:

```
.....
..checkCollapseTimeLevel
...checkCollapseLevel
.....
-----
Ran 40 tests in 68.366s

If there are any errors they will be indicated thus:

$ python dataopsunittests.py
.....F.....checkCollapseTimeLevel
...checkCollapseLevel
.checkCollapseLongitude
.....checkTExtract
.
=====
FAIL: checkCollapseTime (__main__.AveragerTestCase)
-----

Traceback (most recent call last):
  File "dataopsunittests.py", line 184, in checkCollapseTime
    assert axis.units == 'days since 1975'
```

```
AssertionError
```

```
-----
Ran 40 tests in 63.769s
```

```
FAILED (failures=1)
```

The output informs the user that the `checkCollapseTime` test has failed at line 184 and that the value of `axis.units` is not, for some reason 'days since 1975'.

The tests are split up into separate test suites, one suite for each functionality. For example, one suite tests `Averager`, another suite tests covariance functions.

It is easy to configure the test suite to run selected test suites. At the end of the `dataopsunittests.py` file is code of the format

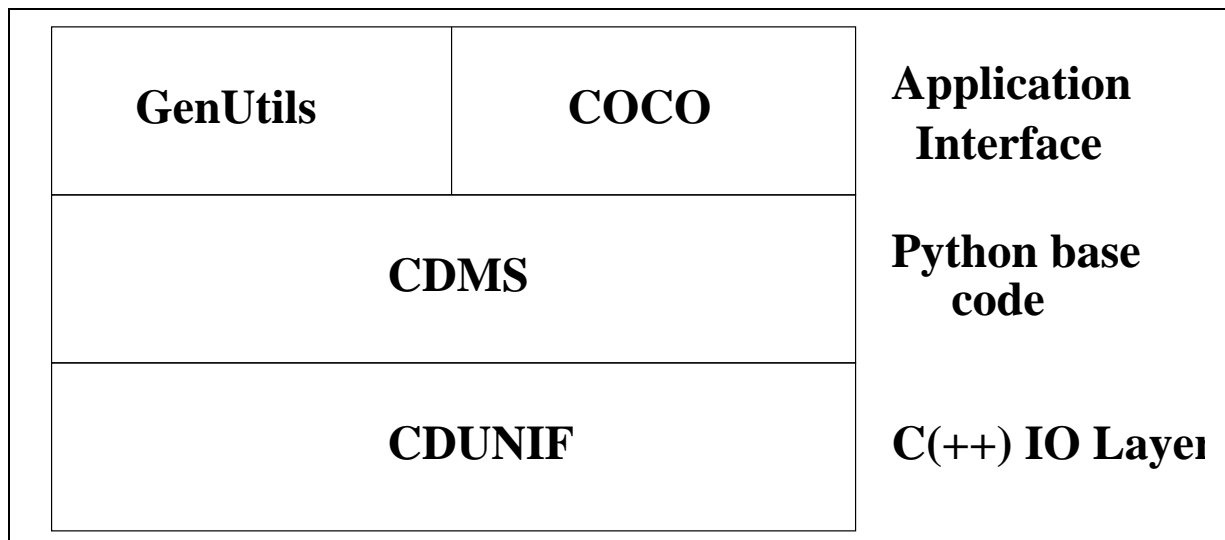
```
TestSuite1 = unittest.makeSuite(AveragerTestCase, 'check')
TestSuite2 = unittest.makeSuite(VarianceTestCase, 'check')
TestSuite3 = unittest.makeSuite(StdDeviatorTestCase, 'check')
TestSuite4 = unittest.makeSuite(MinimizerTestCase, 'check')

alltests = unittest.TestSuite((TestSuite1,
                               TestSuite2,
##                               TestSuite3,
##                               TestSuite4,
                               ))
```

The test suites included in the final `alltests` list are those that will be run. In the example above only `TestSuite1` (`Averager`) and `TestSuite2` (`Variance`) will be run. In short, tests can be added, removed or commented out as required.

## 1.6 Design notes and diagrams

Figure 1.1 shows the architecture of COCO.



**Figure 1.1:** COCO Architecture

Outline specification for a CF Processing Library:

<http://prism.enes.org/WPs/WP4a/ProcessingLib/requirements.html>

## 1.7 Notes for developers

### 1.7.1 Known Problems

There is a problem with plotting variables using `vcdat` if the `netCDF` files contain auxiliary coordinates. It results in a runtime error stating that the “Grid lat/lon domains do not match the variable domain”. A workaround has been implemented that means that auxiliary coordinates are currently ignored.

### 1.7.2 Where to look to extend functionality

All of the user interface methods for the statistical functionality are contained in a python file called `cdms/dataoperations.py`. Addition of further statistical functionality including weights calculations should be made to this file.

### 1.7.3 Recommendations

At the current time there are a lot of additions to the standard CDAT release. It is intended that these changes will eventually be incorporated into the mainstream development of CDAT. Until then it is recommended to stay with the CDAT version 4.0b3. The intricacies and interaction between CDAT and the COCO layer are complex resulting in changes to CDAT having knock-on effect with COCO.

