mmDiagnosis Documentation

Release 1a

Arulalan.T, Dr.Krishna AchutaRao, Dileepkumar.R

April 13, 2015

Contents:

CHAPTER

ONE

GETTING STARTED

UV-CDAT (Ultrascale Visualization Climate Data Analysis Tools) is a powerful and complete front-end to a rich set of visual-data exploration and analysis capabilities well suited for climate-data analysis problems. For more details visit http://uv-cdat.org/

1.1 UV-CDAT Installation Guide

1.1.1 Dependencies on Linux

The following are dependencies need to be installed before installing UV-CDAT in all Linux distributions

- git
- libqt4-dev (under RedHAT the system qt isn't working, user will need to get the binaries from the website)
- libpng-dev
- libxml2-dev
- libxslt-dev
- xorg-dev
- sqlite3
- libsqlite3-dev
- libbz2-dev
- libgdbm-dev
- libxt-dev
- openssl-dev (libssl-dev)
- gfortran
- g++
- qt4-dev-tools / qtcreator
- tcl-dev
- tk-dev
- libgdbm-dev
- libdb4.8-dev
- yasm

- grace
- grads

1.1.2 Dependencies on Ubuntu (extra dependencies)

- libicu48
- libxi-dev
- libglu1-mesa-dev
- libgl1-mesa-dev
- libqt4-opengl-dev

1.1.3 Dependencies on CentOS 6.3 (extra dependencies)

- bzip2-devel
- dbus-devel
- dbus-c++-devel
- dbus-glib-devel
- gtkglext-devel
- mesalibGL
- mesalibGLU
- openGL
- gstreamer-devel
- gstreamer-plugins-base*devel
- gstreamer-plugins-bad*devel
- gstreamer-plugins-good*devel
- libcurl4
- openssl-devel

1.2 Download UV-CDAT Binary

- More detailed system requirements can be found here (https://github.com/UV-CDAT/uvcdat/wiki/System-Requirements)
- UV-CDAT latest releases binary tar ball can be found here (http://sourceforge.net/projects/cdat/files/Releases/UV-CDAT/)
- Download UV-CDAT latest binary version (2.0.0 as on Nov-2014) from here (http://sourceforge.net/projects/cdat/files/Releases/UV-CDAT/2.0.0/) with respect to your linux distribution.
- You will need admin privileges to install UV-CDAT in your system.

1.3 Installation on Linux (Pre-Built Binary)

Step 1: Download and untar the binaries matching best your OS.

$ cd / $ sudo tar xjvf UV-CDAT-[version-no]-[your OS here].tar.bz2

Step 2: For Mac and RH6 ONLY

- **Download the tarsal containing the version of QT UVCDAT has been compiled against RedHAT6** $ cd / $ sudo tar xjvf qt-RH6-64bit-4.8.0.tar.bz2
- **Mac** double click the .dmg file and follow instructions

Step 3: Set Environment Variables

- **Bash users:** $ source /usr/local/uvcdat/[version-no]/bin/setup_cdat.sh
- **t/csh users** $ source /usr/local/uvcdat/[version-no]/bin/setup_cdat.csh

Step 4: Set Alias Paths

In your system ~/.bash_aliases or ~/.bashrc add the following 4 lines.

source /usr/local/uvcdat/2.0.0/bin/setup_cdat.sh

alias uvcdat = '/usr/local/uvcdat/2.0.0/bin/python'

alias uvcdat-gui = '/usr/local/uvcdat/2.0.0/bin/uvcdat'

alias uvcdscan = '/usr/local/uvcdat/2.0.0/bin/cdscan'

Here shown version 2.0.0 for example only. User need to set proper version no whichever they installed in their system.

Step 5: Start enjoying UVCDAT

- **GUI**
  - Type $ uvcdat-gui in your linux terminal to use gui
- **Command Line Python**
  - Type $ uvcdat in your linux terminal to use python shell
  - Type $ uvcdat sample_program.py to execute any uvcdat support python programs.
- **Cdscan**
  - Type $ uvcdsan –x out.xml *.nc to scan all available nc files and make links into small xml file which can be used to load all nc files data inside uvcdat scripts from single xml file.

1.4 Installation on Linux (Build From Source)

- User can visit the following links to install from source. For AIX, UNIX kind of systems, its better start to build and install from UV-CDAT latest source itself.
- **Download UV-CDAT latest Source Code from the following link**
  - https://github.com/UV-CDAT/uvcdat/releases
- **Guide to Install from Source**
  - http://uv-cdat.org/installing.html
  - https://github.com/UV-CDAT/uvcdat/wiki/zold-Building-UVCDAT

1.5 Installation on Ubuntu

Current Stable Release 2.0 Supporting for **Ubuntu 13.x** and **14.x**

System Requirements

$ sudo apt-get install cmake cmake-curses-gui wget libqt4-dev libpng12-dev libxml2-dev libxslt1-dev xorg-dev sqlite3 libsqlite3-dev libbz2-dev libgdbm-dev libxt-dev libssl-dev gfortran g++ qt4-dev-tools tcl-dev tk-dev libgdbm-dev libdb-dev libicu-dev libxi-dev libglu1-mesa-dev libgl1-mesa-dev libqt4-opengl-dev libbz2-dev grads grace

1.5.1 Installing the Binaries (Strongly Suggested)

You must be Root

$ sudo -s

$ cd /

$ wget http://sourceforge.net/projects/cdat/files/Releases/UV-CDAT/2.0.0/UV-CDAT-2.0.0-Ubuntu-14.04-64bit.tar.gz

$ tar xvjf UV-CDAT-2.0.0-Ubuntu-14.04-64bit.tar.gz

$ source /usr/local/uvcdat/2.0.0/bin/setup_runtime.sh

Running UV-CDAT GUI

(Don't forget to source setup_runtime.sh)

$ uvcdat

Bash users add source /usr/local/uvcdat/2.0.0/bin/setup_runtime.sh to your ~/.bashrc file

Csh users add source /usr/local/uvcdat/2.0.0/bin/setup_runtime.csh to your ~/.cshrc file

Set aliases in your bashrc file as mentioned in the previous section.

https://github.com/UV-CDAT/uvcdat/wiki/Installation-on-Ubuntu

1.5.2 Installation on RedHat, Fedora & CentOS

Current Stable Release 2.0 Supporting for **RedHat6 / CentOS6**

System Requirements (Must be Root)

You must have access to the EPEL Repos help link (http://www.thegeekstuff.com/2012/06/enable-epel-repository)

$ sudo yum install cmake cmake-gui wget libpng-devel libxml2-devel libxslt-devel xorg* sqlite-devel bzip2 gdbm-devel libXt-devel openssl-devel gcc-gfortran libgfortran tcl-devel tk-devel libdbi-devel libicu-devel libXi-devel mesa-libGLU-devel mesa-libGL-devel PyQt4-devel gcc-c++ patch grace grads Installation Qt Binary :

$ cd /

$ wget http://sourceforge.net/projects/cdat/files/Releases/UV-CDAT/2.0.0/qt-CentOS-6.5-RedHat6-64bit-4.8.4.tar.bz2

$ tar xvjf qt-CentOS-6.5-RedHat6-64bit-4.8.4.tar.bz2

please add Qt to your path (example in bash) add this to your .bashrc

export Qt=/usr/local/uvcdat/Qt/4.8.4/bin/

export PATH=$PATH:$Qt

Installing the Binaries (Strongly Suggested)

You must be Root

$ sudo -s

$ cd /

$ wget http://sourceforge.net/projects/cdat/files/Releases/UV-CDAT/2.0.0/UV-CDAT-2.0.0-CentOS6.5-RedHat6-64bit.tar.gz

$ tar xvjf UV-CDAT-2.0.0-CentOS6.5-RedHat6-64bit.tar.gz

$ source /usr/local/uvcdat/2.0.0/bin/setup_runtime.sh

Set aliases in your bashrc file as mentioned in the previous section.

Bash users add source /usr/local/uvcdat/2.0.0/bin/setup_runtime.sh to your ~/.bashrc file

Csh users add source /usr/local/uvcdat/2.0.0/bin/setup_runtime.csh to your ~/.cshrc file

https://github.com/UV-CDAT/uvcdat/wiki/installation-on-RedHat—CentOS

1.6 UV-CDAT Documentation

1.6.1 Official Documentation

- CDMS (http://uv-cdat.org/documentation/cdms/cdms.html) Manual
- UV-CDAT Utilities (http://uv-cdat.org/documentation/utilities/utilities.html) Manual
- VCS (http://uv-cdat.org/documentation/vcs/vcs.html) Manual

1.6.2 Useful Tips & Tricks

- http://www.johnny-lin.com/cdat_tips/
- http://drclimate.wordpress.com/2014/01/02/a-beginners-guide-to-scripting-with-uv-cdat/
- http://tuxcoder.wordpress.com/category/python/cdat/
- http://tuxcoder.wordpress.com/category/uvcdat-2/

1.6.3 Slides

- Lesson-1 Python An Introduction (https://www.scribd.com/doc/56253490/Lesson1-Python-An-Introduction)
- Lesson-2 Numpy Arrays (https://www.scribd.com/doc/56254041/Lesson2-Numpy-Arrays)
- Lesson-3 cdutil_genutil (https://www.scribd.com/doc/56254387/Lesson3-cdutil-genutil)
- Lesson-4 VCS_XmGrace_Graphics (https://www.scribd.com/doc/56254572/Lesson4-VCS-XmGrace-CDAT-Graphics)

1.6.4 IPython With Interactive Live Execution Examples Outputs

- Introduction to NumPy Arrays (http://nbviewer.ipython.org/github/arulalant/UV-CDAT-IPython-Notebooks/blob/outputs/1.Introduction_to_NumPy_Arrays.ipynb)
- UV-CDAT-cdms (http://nbviewer.ipython.org/github/arulalant/UV-CDAT-IPython-Notebooks/blob/outputs/2.UV-CDAT-cdms.ipynb)
- UV-CDAT-cdutil_gentuil (http://nbviewer.ipython.org/github/arulalant/UV-CDAT-IPython-Notebooks/blob/outputs/3.UV-CDAT-cdutil_gentuil.ipynb)
- UV-CDAT-Graphics-vcs-xmgrace (http://nbviewer.ipython.org/github/arulalant/UV-CDAT-IPython-Notebooks/blob/outputs/4.UV-CDAT-Graphics-vcs-xmgrace.ipynb)

UV-CDAT IPython Notebooks Source https://github.com/arulalant/UV-CDAT-IPython-Notebooks

1.7 Support

1.7.1 Mailing List

http://uv-cdat.org/mailing-list.html and https://uvcdat.llnl.gov/mailing-list/

1.8 Gallery

The UV-CDAT gallery images contains all different type of plots, projection, 2D & 3D visualization can be found from this link http://uvcdat.llnl.gov/gallery.php

1.9 License

UVCDAT comes under Free Open Source GNU GENERAL PUBLIC LICENSE V3+ Read about GPL License here https://www.gnu.org/licenses/gpl.html

CHAPTER

TWO

INSTALLATION AND SETUP OF MMDIAGNOSIS FRAMEWORK

2.1 Source Code

The complete source code of this project can be found from online github repository

Repository Link : https://github.com/arulalant/mmDiagnosis

2.2 Installing the mmDiagnosis

Download the latest version of mmDiagnosis framework source code from https://github.com/arulalant/mmDiagnosis and extract the zip file.

$ cd mmDiagnosis

$ sudo /usr/local/uvcdat/2.0.0/bin/python setup.py build install

Replace your uvcdat version number in above statement.

2.3 Setup and Configuration

2.4 License

CHAPTER

THREE

DOCUMENTATION OF DIAGNOSISUTILS SOURCE CODE

The diagnosisutils package contains the Data Access Utils (page 11), Time Axis Utils (page 19), and Plot Utils (page 35) modules.

3.1 Data Access Utils

This data access utils uses the Time Axis Utils (page 19) .

The xml_data_access (page 11) module help us to access the all the grib files through single object.

Basically all the grib files are pointed into xml dom by cdscan command.

Then we can the xml files through uv-cdat.

Right now we are generating 8 xml files to access analysis grib files and 7 forecasts grib files.

In the xml_data_access (page 11) module, we are finding which xml needs to access depends upon the user inputs (Type, hour) in the functions of this module.

And once one xml object has initiated then through out that program execute session, it will remains exists and use when ever user needs it.

3.1.1 xml_data_access

class xml_data_access.**GribXmlAccess**(*XmlDir*)

xml access methods

closeXmlObjs()

closeXmlObjs() (page 11): close all the opened xml file objects by cdms2. If we called this method, it will check all the 8 xml objects are either opened or not. If that is opened by cdms2 means, it will close that file object properly. We must call this method for the safety purpose.

..note:: If we called this method, at the end of the program then it should be optimized one. If this method called at any inter mediate level means, then again it need to create the xml object.

Written By: Arulalan.T

Date : 10.09.2011

findPartners(*Type*, *date*, *hour=None*, *returnType='c'*)

findPartners() (page 11): To find the partners of the any particular day anl or any particular day and hour of the fcst. Each fcst file(day) has its truth anl file(day). i.e. today 24 hour fcst file's partner is tomorrow's truth anl file. today 48 hour fcst file's partner is the day after tomorrow's truth anl file. Keep going on the fcst vc anl files.

Same concept for anl files partner but in reverse concept. Today's truth anl file's partners are yesterdays' 24 hour fcst file, day before yesterday's 48 hour fcst file and keep going backward ...

This what we are calling as the partners of anl and fcst files. For present fcst hours partner is future anl file and for present anl partners are the past fcst hours files.

Condition :

if 'f' as passed then hour is mandatory one else 'a' as passed then hour is optional one. returnType either 'c' or 's'

Inputs :

Type = 'f' or 'a' or 'o' i.e fcst or anl or obs file date must be cdtime.comptime object or its string formate hour is like 24 multiples in case availability of the fcst files

Outputs :

If 'f' has passed this method returns a corresponding partner of the anlysis date in cdtime.comptime object If 'a' or 'o' has passed this method returns a dictionary. It contains the availability of the fcst hours as key and its corresponding fcst date in cdtime.comptime object as value of the dict.

we can get the return date as yyyymmdd string formate by passing returnType = 's'

Usage :

example 1 :

```
>>> findPartners('f','2010-5-25',24)
    2010-5-26 0:0:0.0
```

Note: The passed date in comptime in string type.

```
>>> findPartners('f',cdtime.comptime(2010,5,25),24)
    2010-5-26 0:0:0.0
```

Note: The passed date in comptime object itself.

```
>>> findPartners('a','2010-5-26')
    {24: 2010-5-25 0:0:0.0}
```

Note: Returns dictionary which contains key as hour and its corresponding date

example 2 :

```
>>> findPartners('f','2010-5-25',72)
    2010-5-28 0:0:0.0

>>> findPartners('a','2010-6-1', returnType ='s')
     {24: '20100531',
      48: '20100530',
      72: '20100529',
      96: '20100528',
```

```
    120: '20100527',
    144: '20100526',
    168: '20100525'}
```

Note: Depends upon the availability of the fcst and anl files, it should return partner date

example 3 :

```
>>> findPartners('a','20100601',144)
    2010-5-26 0:0:0.0
```

See also:

If not available for the passed hour means it should return None

Written by : Arulalan.T

Date : 03.04.2011

getData(*var*, *Type*, *date*, *hour=None*, *level='all'*, ***latlonregion*)

`getData()` **(page 13): It can extract either the data of a single date or** range of dates. It depends up on the input of the date argument. Finally it should return MV2 variable.

Condition : date is either tuple or string. level is optional. level takes default 'all'. if level passed, it must be belongs to the data variable hour is must when Type arg should be 'f' (fcst) to choose xml object. Pass either (lat,lon) or region.

Inputs :

Type - either 'a'[analysis] or 'f'[forecast] or 'o'[observation] var,level must be belongs to the data file key word arg lat,lon or region should be passed date formate must one of the followings

date formate 1: date = (startdate, enddate) here startdate and enddate must be like cdtime.comptime formate.

date formate 2: date = (startdate)

date formate 3: date = 'startdate' or date = 'date'

eg for date input : date = ('2010-5-1','2010-6-30') date = ('2010-5-30') date = '2010-5-30'

Outputs :

If user passed single date in the date argument, then it should return the data of that particular date as single MV2 variable.

If user passed start and enddate in the date argument, then it should return the data for the range of dates as single MV2 variable with time axis.

Written by: Arulalan.T

Date: 10.05.2011

getDataPartners(*var*, *Type*, *date*, *hour=None*, *level='all'*, *orginData=0*, *datePriority='o'*, ***latlonregion*)

`getDataPartners()` (page 13): It can extract either the orginDate with its partnersData or it can extract only the partnersData without its orginData for a single date or range of dates.

It depends up on the input of the orginData, datePriority,date args. Finally it should return partnersData and/or orginData as MV2 variable

Condition :

date is either tuple or string. level is optional. level takes default 'all'. if level passed, it must be belongs to the data variable hour is must when Type arg should be 'f' (fcst) to select xml object. hour is must when range of date passed, even thogut Type arg should be 'a'(anl) or 'o'(obs),to choose one fcst xml object along with hour. Pass either (lat,lon) or regionself.

Inputs:

Type - either 'a'[analysis] or 'f'[forecast] or 'o'[observation] var,level must be belongs to the data file orginData - either 0 or 1. 0 means it shouldnot return the

orginData as single MV2 var. 1 means it should return both the orginData and its partnersData as two seperate MV2 vars.

datePriority - either 'o' or 'p'. 'o' means passed date is with

respect to orginData. According to this orginData's date, it should return its partnersData.

'p' means passed date is with respect to partnersData. According to this partnersData's date, it should return its orginData.

key word arg lat,lon or region should be passed

date formate 1: date = (startdate,enddate) here startdate and enddate must be like cdtime.comptime formate.

date formate 2: date = (startdate)

date formate 3: date = 'startdate' or date = 'date'

eg for date input :

date = ('2010-5-1','2010-6-30') date = ('2010-5-30') date = '2010-5-30'

Outputs:

If user passed single date in the date argument, then it should return the data of that particular date (both orginData & partnersData) as a single MV2 variable.

If user passed start and enddate in the date argument, then it should return the data (both orginData & partnersData) for the range of dates as a single MV2 variable with time axis.

Usage:

Note: if 'a'(anl) file is orginData means 'f'(fcst) files are its partnersData and vice versa.

example1:

```
>>> a,b = getDataPartners(var = 'U component of wind',Type = 'a',
       date = '2010-6-5',hour = None,level = 'all',orginData = 1,
       datePriority = 'o', lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. anl. its timeAxis date is '2010-6-5'. b is partnersData. i.e. fcst. its 24 hour fcst date w.r.t orginData is '2010-6-4'. 48 hour is '2010-6-3'.

Depends upon the availability of date of fcst files,it should return the data. In NCMRWF2010 model, it should return maximum of 7 days fcst.

If we will specify any hour in the same eg, that should return only that hour fcst file data instead of returning all the available fcst hours data.

example2:

```
>>> a,b = getDataPartners(var = 'U component of wind',Type = 'f',
        date = '2010-6-5',hour = 24,level = 'all',orginData = 1,
        datePriority = 'o', lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e.fcst 24 hour.its timeAxis date is '2010-6-5'. b is partnersData. i.e. anl. its anl date w.r.t orginData is '2010-6-6'.

example3:

```
>>> b = getDataPartners(var = 'U component of wind',Type = 'f',
       date = '2010-6-5',hour = 24,level = 'all', orginData = 0,
       datePriority = 'o', lat=(-90,90),lon=(0,359.5))
```

b is partnersData. i.e. anl. its anl date w.r.t orginData is '2010-6-6'. No orginData. Because we passed orginData as 0.

example4:

```
>>> a,b = getDataPartners(var = 'U component of wind',Type = 'f',
        date = '2010-6-5',hour = 24,level = 'all',orginData = 1,
        datePriority = 'p', lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. fcst 24 hour.its timeAxis date is '2010-6-6'. b is partnersData. i.e. anl. its anl date w.r.t orginData is '2010-6-5'. we can compare this eg4 with eg2. In this we passed datePriority as 'p'. So the passed date as set to the partnersData and orginData's date has shifted to the next day.

example5:

```
>>> a,b = getDataPartners(var = 'U component of wind',Type = 'a',
        date = ('2010-6-5','2010-6-6'),hour = 24,level = 'all',
    orginData = 1,datePriority = 'o', lat=(-90,90),lon=(0,359.5))
```

Note: Even though we passed 'a' Type, we must choose the hour option to select the fcst file, since we are passing the range of dates.

---

a is orginData. i.e. anl. its timeAxis size is 2. date are '2010-6-5' and '2010-6-6'.

b is partnersData. i.e. fcst 24 hour data. its timeAxis size is 2. date w.r.t orginData are '2010-6-4' and '2010-6-5'.

a's '2010-6-5' has partner is b's '2010-6-4'.i.e.orginData(anl) partners is partnersData's (fcst).

same concept for the remains day. a's '2010-6-6' has partner is b's '2010-6-5'.

example6:

```
>>> a,b = getDataPartners(var = 'U component of wind',Type = 'a',
         date = ('2010-6-5','2010-6-6'),hour = 24,level = 'all',
    orginData = 1,datePriority = 'p', lat=(-90,90),lon=(0,359.5))
```

Note: Even though we passed 'a' Type, we must choose the hour option to select the fcst file, since we are passing the range of dates.

a is orginData. i.e. anl. its timeAxis size is 2. date are '2010-6-6' and '2010-6-7'.

b is partnersData. i.e. fcst 24 hour data. its timeAxis size is 2. date w.r.t orginData are '2010-6-5' and '2010-6-6'.

a's '2010-6-6' has partner is b's '2010-6-5'.i.e.orginData(anl) partners is partnersData's (fcst).

same concept for the remains day. a's '2010-6-7' has partner is b's '2010-6-6'. we can compare this eg6 with eg5.In this we passed datePriority as 'p'. So the passed date as set to the partnersData and orginData's date has shifted towards the next days.

Written by: Arulalan.T

Date: 27.05.2011

getMonthAvgData (*var*, *Type*, *level*, *month*, *year*, *calendarName=None*, *hour=None*, ***latlonregion*)

getMonthAvgData() (page 16): It returns the average of the given month data (all days in the month) for the passed variable options

Condition : calendarName,level are optional. level takes default 'all' if not pass arg for it. hour is must when Type arg should be 'f' (fcst) to select xml object. Pass either (lat,lon) or region.

Inputs : Type - either 'a' or 'f' or 'o' var,level must be belongs to the data file month may be even in 3 char like 'apr' or 'April' or 'aPRiL' or like any month year must be passed as integer calendarName default None, it takes cdtime.DefaultCalendar key word arg lat,lon or region should be passed

Outputs : It should return the average of the whole month data for the given vars as MV2 variable

Usage :

example :

```
>>> getMonthAvgData(var = "Geopotential Height",Type = 'f',
            level = 'all', month = 'july',year=2010 , hour = 24,
             region = AIR ) #lat=(-90,90),lon=(0,359.5)
```

returns dataAvg of one month data as single MV2 variable

Written by: Arulalan.T

Date: 29.04.2011

getRainfallData (*date=None*, *level='all'*, ***latlonregion*)

getRainfallData() (page 16): Extract the rainfall data from the xml file which has created by the cdscan command.

Inputs [var takes from the instance member of GribAccess class]

if we passed date,level then it should return data accordingly By default level takes 'all' levels. Pass either (lat,lon) or region keyword arg

Condition [we must set the member variable called rainfallXmlPath] and (rainfallXmlVar or rainfallModel), then only we can access this method. rainfallXmlVar is the obeservation rainfall variable name to access the data. OR rainfallModel is the model name, which has set in the global variable names settings. By Using it, this method should get the observation rainfall variable name.

Written by : Arulalan.T

Date : 29.05.2011

getRainfallDataPartners(*date*, *hour=None*, *level='all'*, *orginData=1*, *datePriority='o'*, ***latlonregion*)

getRainfallDataPartners() (page 17): It returns the rainfall data & its partners data(fcst is the partner of the observation. i.e. rainfall) as MV2 vars

Condition: startdate is must. enddata is optional one. If both startdate and enddate has passed means, it should return the rainfall data and partnersData within that range.

Inputs:

orginData - either 0 or 1. 0 means it shouldnot return the

orginData as single MV2 var. 1 means it should return both the orginData and its partnersData as two seperate MV2 vars.

datePriority - either 'o' or 'p'. 'o' means passed date is with respect to orginData. According to this orginData's date, it should return its partnersData. 'p' means passed date is with respect to partnersData.

According to this partnersData's date, it should return its orginData.

key word arg lat,lon or region should be passed By default hour is None and level is 'all'.

self.rainfallXmlPath is mandatory one when you choosed orginData is 1. you must set the rainfall xml path to the rainfallXmlPath.

self.rainfallModel is the model name, which has set in the global variables settings, to get the model fcst and its obeservation variable name to access the data. It is mandatory one.

date formate 1: date = (startdate,enddate) here startdate and enddate must be like cdtime.comptime formate.

date formate 2: date = (startdate)

date formate 3: date = 'startdate' or date = 'date'

eg for date input :

date = ('2010-5-1','2010-6-30') date = ('2010-5-30') date = '2010-5-30'

By default skipdays as 1 takes place. User cant override till now.

Outputs :

If user passed single date in the date argument, then it should return the data of that particular date (both orginData & partnersData) as MV2 variable.

If user passed start and enddate in the date argument, then it should return the data (both orginData & partnersData) for the range of dates as MV2 variable with time axis.

Usage :

Note: if 'r'(observation) file is orginData means 'f'(fcst) files are its partnersData.

example1:

```
>>> a,b = getRainfallDataPartners(date = '2010-6-5',hour = None,
                  level = 'all',orginData = 1,datePriority = 'o',
                                        lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. rainfall observation. its timeAxis date is '2010-6-5'.

b is partnersData. i.e. fcst. its 24 hour fcst date w.r.t orginData is '2010-6-4'. 48 hour is '2010-6-3'.

Depends upon the availability of date of fcst files,it should return the data. In NCMRWF2010 model, it should return maximum of 7 days fcst.

If we will specify any hour in the same eg, that should return only that hour fcst file data instead of returning all the available fcst hours data.

example2:

```
>>> a,b = getRainfallDataPartners(date = '2010-6-5',hour = 24,
                level = 'all',orginData = 1,datePriority = 'o',
                                    lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. rainfall observation. its timeAxis date is '2010-6-5'. b is partnersData. i.e. fcst 24 hour. its fcst date w.r.t orginData is '2010-6-6'.

example3:

```
>>> b = getRainfallDataPartners(date = '2010-6-5',hour = 24,
                level = 'all',orginData = 0,datePriority = 'o',
                                    lat=(-90,90),lon=(0,359.5))
```

b is partnersData. i.e. fcst. its fcst date w.r.t orginData is '2010-6-6'. No orginData. Because we passed orginData as 0.

example4:

```
>>> a,b = getRainfallDataPartnerss(date = '2010-6-5',hour = 24,
                level = 'all',orginData = 1,datePriority = 'p',
                                    lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. rainfall observation. its timeAxis date is '2010-6-6'.

b is partnersData. i.e. fcst 24 hour. its fcst date w.r.t orginData is '2010-6-5'. we can compare this eg4 with eg2.

In this we passed datePriority as 'p'. So the passed date as set to the partnersData and orginData's date has shifted to the next day.

example5:

```
>>> a,b = getRainfallDataPartners(date = ('2010-6-5','2010-6-6'),
                            hour = 24,level = 'all',orginData = 1,
                datePriority = 'o', lat=(-90,90),lon=(0,359.5))
```

Note: We must choose the hour option to select the fcst file, since we are passing the range of dates.

---

a is orginData.i.e.rainfall observation.its timeAxis size is 2. date are '2010-6-5' and '2010-6-6'.

b is partnersData.i.e.fcst 24 hour data.its timeAxis size is 2. date w.r.t orginData are '2010-6-4' and '2010-6-5'.

a's '2010-6-5' has partner is b's '2010-6-4'. i.e. orginData(rainfall observation) partners is partnersData(fcst)

same concept for the remains day. a's '2010-6-6' has partner is b's '2010-6-5'.

example6:

```
>>> a,b = getRainfallDataPartners(date = ('2010-6-5','2010-6-6'),
                            hour = 24,level = 'all',orginData = 1,
                  datePriority = 'p', lat=(-90,90),lon=(0,359.5))
```

a is orginData. i.e. rainfall observation. its timeAxis size is 2. date are '2010-6-6' and '2010-6-7'.

b is partnersData. i.e. fcst 24 hour data. its timeAxis size is 2. date w.r.t orginData are '2010-6-5' and '2010-6-6'.

a's '2010-6-6' has partner is b's '2010-6-5'. i.e. orginData(rainfall observation) partners is partnersData(fcst)

same concept for the remains day. a's '2010-6-7' has partner is b's '2010-6-6'. we can compare this eg6 with eg5. In this we passed datePriority as 'p'. So the passed date as set to the partnersData and orginData's date has shifted towards the next days.

Written by: Arulalan.T

Date: 29.05.2011

getXmlPath(*Type*, *hour=None*)

getXmlPath() (page 19): To get the xml's absolute path.

Inputs [Type is either 'a' or 'o' or 'f' or 'r'] hour is mandatory when you pass 'f' or 'r'. 'a' - analysis, 'f' - forecast, 'o' - observation, 'r' - reference.

Written by : Arulalan.T

Date : 21.08.2011

listvariable(*Type*, *hour=None*)

:func:'listvariable': By passing Type and/or hour args to this method, it will return the listvariable method of the appropriate xml file.

Returns the listvariable of cdms2 open object method result

3.2 Time Axis Utils

This timeutils (page 19) module helps us to generate our own time axis, correct existing time axis bounds and generate bounds.

Here we used inbuilt methods of cdtime and cdutil module of uv-cdat.

3.2.1 timeutils

class xml_data_access.**TimeUtility**

comp2timestr(*comptime*, *returnHour='y'*)

comp2timestr() **(page 19): To convert date from cdtime.comptime into** 'yyyymmdd' formate or 'yyyymmddhh' formate as string

Condition : passing date must be comptime formate

Inputs [date in comptime.] returnHour takes 'y' or 'yes' or 'n' or 'no'. Default it takes 'y'.

Outputs [It should return the date in 'yyyymmddhh' string formate, if] returnHour passed 'y' or 'yes'. It should return the date in 'yyyymmdd' string formate, if returnHour passed 'n' or 'no'.

Usage :

example1 :

```
>>> compobj = cdtime.comptime(2010,4,29) -> 2010-4-29 0:0:0.0
>>> comp2timestr(compobj)
>>> '2010042900'
```

Note: It should return in yyyymmddhh string formate by default. Hour is 00.

example2 :

```
>>> compobj = cdtime.comptime(2010,4,29,10) -> 2010-4-29 10:0:0.0
>>> comp2timestr(compobj)
>>> '2010042910'
```

Note: It should return in yyyymmddhh string formate by default. Hour is 10.

example2 :

```
>>> compobj = cdtime.comptime(2010,4,29,10) -> 2010-4-29 10:0:0.0
>>> comp2timestr(compobj, returnHour = 'n')
>>> '20100429'
```

Note: It should return in yyyymmdd string formate only even though hour passed in the component object. Because we passed returnHour as 'n'.

Written by : Arulalan.T

Date : 29.04.2011 Updated : 21.08.2011

getSameDayData(*varName*, *fpath*, *day*, *mon*, *hr=0*, ***kwarg*)

getSameDayData [get same day data from all the available years or] needed year/s.

Inputs : varName - variable name fpath - file path day : day of the needed [of all the years of the data]. mon : month of the needed [of all the years of the data]. hr : hour of the needed KWargs : (latitude and/or longitude) or (region) and/or level

and/or year

..note:: It will extract the particular day from all the available years. Ofcourse you can control the needed year/s also by using year kwarg. Eg1: To get leap day data from all the available years,

getSameDayData(varName, path, day=29, mon=2) It will return all years the leapday data (Feb 29) along with its timeAxis and missing values.

Refer: getSeasonData

Written By : Arulalan.T

Date : 13.08.2013

getSeasonName(*startBound*, *endBound*, *year='1'*, *units='days'*)
Inputs : start bound and end bound of the season. Returns : Season name Date : 03.06.2012

getSeasonalData(*varName*, *fpath*, *sday*, *smon*, *eday*, *emon*, *hr=0*, ***kwarg*)

getSeasonalData [user defined season data extraction from the filepath.] It will extract the specified season (passed in the args) from all the available years or user can pass the needed year/s. User can extract same day for all the available years/ needed year/s. For eg : One can get all the leapday data alone from all the years.

Inputs: varName : variable name fpath : data (nc/ctl/grib/xml/cdml) file path sday : starting day of the season [of all the years of the data] smon : starting month of the season [of all the years of the data] eday : ending day of the season [of all the years of the data] emon : ending month of the season [of all the years of the data] hr : hour for both start and end date KWargs : (latitude and/or longitude) or (region) and/or level

and/or cyclic.

cyclic [If cyclic is true, it extract the cyclic year data also.]

i.e. In winter season (Nov to Apr), then it will extract the first year (Jan to Apr) and last year (Nov to Dec) also. So it will be look like cyclic year of seasonData.

If cyclic is false means, it will not extract the last

year partial months data. i.e. it will not extract last year (Nov to Dec) for winter season.

year [Its optional only. By default it is None. i.e. It will] extract all the available years seasonalData. If one interger year has passed means, then it will do extract of that particular year seasonData alone. If two years has passed in tuple, then it will extract the range of years seasonData from year[0] to year[1]. eg1 : year=2005 it will extract seasonData of 2005 alone. eg2 : year=(1971, 2013) it will extract seasonData from

1971 to 2013 years.

..note:: If end day and end month is lower than the start day & start

month, then we need to extract the both current and next year data. For eg : Winter Season (November to April). It can not be reversed for this winter season. We need to extract data from current year november month upto next year march month.

If you pass one year data and passed the above

winter season, then it will be extracted november and december months data and will be calculated variance for that alone.

If you will pass two year data then it will extract the data

from november & december of first year and january, feburary & march of next year will be extracted and calculated variance for that.

..note:: If both sday == eday and smon == emon then it will extract the this particular day from all the available years. Ofcourse you can control the needed year/s also. Eg1: To get leap day data from all the available years,

getSeasonalData(varName, path, sday=29, smon=2, emon=29, emon=2) It will return all years the leapday data (Feb 29) along with its timeAxis and missing values.

Written By : Arulalan.T

Date : 26.07.2012 Updated : 13.08.2013

getSummerData (*varName*, *fpath*, *sday=1*, *smon=5*, *eday=31*, *emon=10*, *hr=0*, ***kwarg*)

getSummerData : summer season (May to October) data

Inputs : varName - variable name fpath - file path sday : starting day of the summer season [of all the years of the data]. smon : starting month of the summer season [of all the years of the data]. eday : ending day of the summer season [of all the years of the data]. emon : ending month of the summer season [of all the years of the data]. hr : hour for both start and end date KWargs : (latitude and/or longitude) or (region) and/or level

Refer: getSeasonData

Written By : Arulalan.T

Date : 26.07.2012

getTimeAxisFullMonths (*timeAxis*, *returnType='c'*, *returnHour='y'*)

getTimeAxisFullMonths() (page 22): Get the fully available months name and its firstday & lastday from the passed timeAxis.

Condition : timeAxis must be an instance of cdms2.axis.TransientAxis

Inputs [Pass the any range of timeAxis object.] returnType is either 'c' or 's'. If 'c' means the dates are cdtime.comptime object itself. if 's' means the dates are yyyymmddhh string (by default) or yyyymmdd w.r.t returnHour. returnHour takes either 'y/yes' or 'n/no'.

Outputs [It should return a dictionary which has key as the year.] This year key has the value as dictionary type itself. The nested dictionary has month as key and value as tuple, which contains the stardate and enddate of that month. It should return month only for fully available month in the passed timeAxis. i.e. If timeAxis has some incomplete months of particular year means, it should not return that month and its dates.

Usage :

..seealso:: we can pass any range of timeAxis. Even its hourly series, it should works. But it should return month & dates for fully available dates of months in timeAxis.

example 1:

```
>>> tim = _generateTimeAxis(70, '2011-5-25')
>>> tim
   id: time
   Designated a time axis.
   units:  days since 2011-5-25
   Length: 70
   First:  0
   Last:   69
   Other axis attributes:
      calendar: gregorian
      axis: T
   Python id:  0x8aea74c
```

```
>>> getTimeAxisFullMonths(tim)
{2011: {
'JULY': (2011-7-1 0:0:0.0, 2011-7-31 0:0:0.0),
'JUNE': (2011-6-1 0:0:0.0, 2011-6-30 0:0:0.0)}
}
```

..note:: Here 2011 as the key for the primary dictionary.

And JUNE, JULY are the keys for the secondary(inner) dictionary, which contains the stardate and enddate of that month and year.

```
>>> tim.asComponentTime()[0]
2011-5-25 0:0:0.0
```

..note:: The actual firstdate of the timeAxis is 25th may 2011.

But its not complete month. So this may month is not returned in the above.

```
>>> tim.asComponentTime()[-1]
2011-8-2 0:0:0.0
```

..note:: The actual lastdate of the timeAxis is 2nd aug 2011. But its not complete month. So this aug month is not returned in the above.

example 2:

```
>>> tim1 = _generateTimeAxis(70, '2011-12-1')
>>> tim1
   id: time
   Designated a time axis.
   units:  days since 2011-12-1
   Length: 70
   First:  0
   Last:   69
   Other axis attributes:
      calendar: gregorian
      axis: T
   Python id:  0xa2c10ac


>>> getTimeAxisFullMonths(tim1, returnType = 's',
                                                returnHour = 'n')
{2011: {'DECEMBER': ('20111201', '20111231')},
 2012: {'JANUARY': ('20120101', '20120131')}}
```

..note:: In this example, we will get 2011 and 2012 are the keys of the primary dictionary. And 2011 has 'DECEMBER' month and its startdate & enddate as key and value. Same as for 2012 has 'JANUARY' month and its startdate & enddate as key and value. Here dates are in yyyymmdd string formate. Here we passed returnHour as 'n'.

Written By : Arulalan.T

Date : 24.08.2011

getTimeAxisMonths (*timeAxis*, *returnType='c'*, *returnHour='y'*)

getTimeAxisMonths() **(page 23): Get the available months name and its** firstday & lastday from the passed timeAxis.

Condition : timeAxis must be an instance of cdms2.axis.TransientAxis

Inputs [Pass the any range of timeAxis object.] returnType is either 'c' or 's'. If 'c' means the dates are cdtime.comptime object itself. if 's' means the dates are yyyymmddhh string (by default) or yyyymmdd w.r.t returnHour. returnHour takes either 'y/yes' or 'n/no'.

Outputs [It should return a dictionary which has key as the year.] This year key has the value as dictionary type itself. The nested dictionary has month as key and value as tuple, which contains the stardate and enddate of that month. It should return month only for the available months in the passed timeAxis.

Usage :

..seealso:: we can pass any range of timeAxis. Even its hourly series, it should works.

..seealso:: getTimeAxisFullMonths(), getTimeAxisPartialMonths()

example 1:

```
>>> tim = _generateTimeAxis(70, '2011-5-25')
>>> tim
   id: time
   Designated a time axis.
   units:  days since 2011-5-25
   Length: 70
   First:  0
   Last:   69
   Other axis attributes:
      calendar: gregorian
      axis: T
   Python id:  0x8aea74c


>>> getTimeAxisMonths(tim)
{2011: {
'MAY': (2011-5-25 0:0:0.0, 2011-5-31 0:0:0.0),
'JUNE': (2011-6-1 0:0:0.0, 2011-6-30 0:0:0.0),
'JULY': (2011-7-1 0:0:0.0, 2011-7-31 0:0:0.0),
'AUGUST': (2011-8-1 :0:0.0, 2011-8-2 0:0:0.0)}
}
```

..note:: Here 2011 as the key for the primary dictionary.

And MAY, JUNE, JULY and AUGUST are the keys for the secondary(inner) dictionary, which contains the stardate and enddate of that month and year.

```
>>> tim.asComponentTime()[0]
2011-5-25 0:0:0.0
```

..note:: The actual firstdate of the timeAxis is 25th may 2011.

But its not complete month. Even though it is returning that available may month startdate and its enddate.

```
>>> tim.asComponentTime()[-1]
2011-8-2 0:0:0.0
```

..note:: The actual lastdate of the timeAxis is 2nd aug 2011. But its not complete month. Even though it is returning that available may month startdate and its enddate.

..note:: It also returnning the fully available months, in this example June & July.

example 2:

```
>>> tim1 = _generateTimeAxis(70, '2011-12-1')
>>> tim1
   id: time
   Designated a time axis.
   units:  days since 2011-12-1
   Length: 70
   First:  0
   Last:   69
   Other axis attributes:
      calendar: gregorian
      axis: T
   Python id:  0xa2c10ac


>>> getTimeAxisMonths(tim1, returnType = 's',
                              returnHour = 'n')
{2011: {'DECEMBER': ('20111201', '20111231')},
 2012: {'JANUARY': ('20120101', '20120131')}}
```

..note:: In this example, we will get 2011 and 2012 are the keys of the primary dictionary. And 2011 has 'DECEMBER' month and its startdate & enddate as key and value. Same as for 2012 has 'JANUARY' month and its startdate & enddate as key and value. Here dates are in yyyymmdd string formate. Here we passed returnHour as 'n'.

Written By : Arulalan.T

Date : 06.03.2013

getTimeAxisPartialMonths(*timeAxis*, *fullMonths='auto'*, *returnType='c'*, *returnHour='y'*)

getTimeAxisPartialMonths() (page 25): Get the partially available months name and its firstday & lastday from the passed timeAxis. Its not fully available months.

Condition : timeAxis must be an instance of cdms2.axis.TransientAxis

Inputs : Pass the any range of timeAxis object.

fullMonths is which contains the year and its full months name in dictionary. i.e. fullMonths is the output of the *getTimeAxisFullMonths'(...) method for the same timeAxis. By default it takes 'auto' string. It means, it should call the 'getTimeAxisFullMonths* method. Otherwise user can pass the same kind of input.

returnType is either 'c' or 's'. If 'c' means the dates are cdtime.comptime object itself. if 's' means the dates are yyyymmddhh string (by default) or yyyymmdd w.r.t returnHour.

returnHour takes either 'y/yes' or 'n/no'.

Outputs [It should return a dictionary which has key as the year.] This year key has the value as dictionary type itself. The nested dictionary has month as key and value as tuple, which contains the stardate, enddate and no of days of that partial month. It should return month only for partially available month in the passed timeAxis. i.e. If timeAxis has some incomplete months of particular year means, those months start,enddate and its no of days should be retuned.

Usage :

..seealso:: we can pass any range of timeAxis. Even its hourly series, it should works. But it should return month, dates, total days for partially available dates of months in timeAxis.

example 1:

```
>>> tim = _generateTimeAxis(37, '2011-5-25')
# here we genearate the time axis with partial May month and
# complete June month.
>>>
>>> tim
   id: time
   Designated a time axis.
   units:  days since 2011-5-25
   Length: 37
   First:  0
   Last:   36
   Other axis attributes:
      calendar: gregorian
      axis: T
   Python id:  0xa5604ec
>>>
>>> getTimeAxisPartialMonths(tim)
{2011: {'MAY': (2011-5-25 0:0:0.0, 2011-5-31 0:0:0.0, 7)}}
```

..note:: Here 2011 as the key for the primary dictionary.

And May is the key for the secondary(inner) dictionary, which contains the stardate, enddate and total days of the May month. of that month and year.

```
>>> getTimeAxisFullMonths(tim)
{2011: {'JUNE': (2011-6-1 0:0:0.0, 2011-6-30 0:0:0.0)}}
```

..note:: Here we called the fullMonths. So the June month has

returned with its stardate and enddate.

```
>>> tim.asComponentTime()[0]
2011-5-25 0:0:0.0
```

..note:: The actual firstdate of the timeAxis is 25th may 2011.

But its not complete month. So this may month is returned in the above getTimeAxisPartialMonths() method example.

```
>>> tim.asComponentTime()[-1]
2011-6-30 0:0:0.0
```

..note:: The June month is complete month. So it should return only when we call the getTimeAxisFullMonths() method, not in getTimeAxisPartialMonths() method.

Written By : Arulalan.T

Date : 28.11.2011

getWinterData(*varName*, *fpath*, *sday=1*, *smon=11*, *eday=30*, *emon=4*, *hr=0*, ***kwarg*)

getWinterData : winter season (November to April) data

Inputs : varName - variable name fpath - file path sday : starting day of the winter season [of all the years of the data]. smon : starting month of the winter season [of all the years of the data]. eday : ending day of the winter season [of all the years of the data]. emon : ending month of the winter season [of all the years of the data]. hr : hour for both start and end date KWargs : (latitude and/or longitude) or (region) and/or level

Refer: getSeasonData

Written By : Arulalan.T

Date : 26.07.2012

has_all (*timeAxis*, *deepsearch=False*, *missingYears=0*, *missingMonths=0*, *missingHours=0*)
has_all : either the passed time axis has all the time series or not !

Returns [True it timeAxis has no missing value and no duplicates.] Otherwise returns False.

deepsearch - 1 | 0. If deepsearch enabled return if it has missing time series in between, then it should return those missing time series as component time string in list along with False.

deepsearch 0 will return the boolean value in rapid speed.

missingYears - 1 | 0. If deepsearch is 1 and missingYears is 1, then it should find only the missing years from the passed timeAxis. It will not worry about the months & hours until missingMonths & missingHours are enabled.

missingMonths - 1 | 0. If deepsearch is 1, missingYears is 1 and missingHours is 1, then it will find the missing months also in the timeAxis. It will not worry about the missing hours.

missingHours - 1 | 0. If deepsearch is 1, missingYears is 1, missingMonths is 1 and missingHours is 1, it will do just find all missing sequance time slice from the timeAxis.

User can enable deepsearch alone instead of enabling all the missingYears, missingMonths and missingHours.

..note:: By default deepsearch flag, will try to find out the missing time sequance from the user passed timeAxis itself. This function should works correctly for the timeAxis which contains same delta(diff b/w first and second timeAxisIndex) through out the timeAxis. Otherwise user can use missingYears and/or missingMonths for complex timeAxis.

..note:: return true if user passed length of 1 timeAxis or if user enabled missingYears, then if that retunes only one year, then we can just return True ! (In that case it could be daily or monthly or hourly) it may be have missing time slice also, if user passed missingYears. so user has take care of this !

Refer : _getYears()

Written By : Arulalan.T

Date : 09.10.2012

has_missing (*timeAxis*, *deepsearch=False*, *missingYears=0*, *missingMonths=0*, *missingHours=0*)

has_missing [either the passed time axis has missing time series] or not !

It just return the opposite boolean flag of the function self.has_all(timeAxis, ...)

For more doc, see the has_all.__doc__

Refer : has_all(), _getYears()

Date : 11.10.2012

monthFirstLast (*month*, *year*, *calendarName=None*, *returnType='c'*, *returnHour='n'*)

monthFirstLast() (page 27): To find and return the first date and last date of the given month of the year, with cdtime.calendar option.

Condition : passing month should be either integer of month or name of the month in string. year should be an integer or string calendar is optional. It takes default calendar

Inputs : month may be even in 3 char like 'apr' or 'April' or 'aPRiL' or like any month year must be passed as integer. returnType is either 'c' or 's'. If 'c' means it should return as cdtime.comptime object and if 's' means it should return date as yyymmddhh or yyyymmdd string formate. returnHour is either 'y' or 'n'. If yes means, and returnType is 's' means, it should return hour also.(i.e yyymmddhh), otherwise yyyymmdd only (by default).

Outputs : It should return the first date and last date of the given month & year in yyyymmdd string formate inside tuple

Usage :

example1 :

```
>>> monthFirstLast(4,2010)
>>> (2010-4-1 0:0:0.0, 2010-4-30 0:0:0.0)
```

Note: It should return as cdtime.comptime object

example2 :

```
>>> monthFirstLast('feb','2010',returnType = 's')
>>> ('20100228', '20100228')
```

Note: It should return in yyyymmdd string formate

Written by : Arulalan.T

Date : 29.04.2011

moveTime (*year*, *month*, *day*, *moveday=0*, *movehour=0*, *calendarName=None*, *returnType='c'*)

moveTime() (page 28): To move the day or/and hour in both direction and get the moved date yyyymmdd/yyyymmddhh format

Condition [passing year,month,day,moveday,movehour should be integer] type.

Inputs [moveday is an integer to move the date. If it is negative,] then we should get the previous date with interval of the no of days [i.e. moveday]

movehour is an integer to move the hours. If it is negative, then we should get the previous day hours with interval of the hours [i.e. movehour]

returnType is either 'c' or 's'. 'c' means cdtime.comptime object 's' means yyyymmdd string formate if movehour is 0 and

yyyymmddhh if movehour has passed some hour.

Outputs [It should return the comptime date object by default.] using returnType = 's', we can get yyyymmdd or yyymmddhh string formate.

Usage :

example1 :

```
>>> moveTime(2011, 04, 03, moveday=200)
2011-10-20 0:0:0.0
    ..note:: 200 days moved
>>> moveTime(2011, 04, 03, moveday = -200, returnType='s')
'2010-9-15'
    ..note:: 200 days moved in backward and return as string
             in yyyymmdd formate
```

example2 :

```
>>> moveTime(2011, 4, 3, moveday = 0, movehour = 10)
2011-4-3 10:0:0.0
    ..note:: 0 days moved.But 10 hours moved.


>>> moveTime(2011,4,3, moveday = 2, movehour = 10)
2011-4-5 10:0:0.0
    ..note:: Both days and hours are moved.
             2 days, 10 hours moved.
```

example3 :

```
>>> moveTime(2011,4,3,moveday=2,movehour=10,returnType='s')
'2011040510'
    ..note:: Here passed returnType as 's'. Also passed hour
             as 10. So it should return as yyyymmddhh fromate


>>> moveTime(2011,4,3,moveday = 2,returnType='s')
'20110405'
    ..note:: Here we didnt pass hour. So it should return
             yyyymmdd formate only.
```

example4 :

```
>>> moveTime(2011,4,3,returnType='s')
'20110403'
    ..note:: Here we didnt pass any moveday and any movehour.
             So it should return only what we passed the date,
             without any movements in days/hours.
```

example4 :

```
>>> moveTime(2011,2,28,366)
2012-2-29 0:0:0.0
    ..note:: 2012 is a leap year. By default it take
             cdtime.DefaultCalendar.


>>> moveTime(2011,2,28,366,calendarName=cdtime.NoLeapCalendar)
2012-3-1 0:0:0.0
    ..note:: Eventhough 2012 is a leap year, it doesnt give
             date like previous example, because we have
             passed cdtime.cdtime.NoLeapCalendar.
```

Written by : Arulalan.T

Date : 06.04.2011 Updated : 21.08.2011

tRange(*startdate*, *enddate*, *stepday=0*, *stephour=0*, *calendarName=None*, *returnType='c'*, *returnHour='y'*)

tRange() **(page 29): generate the dates in yyyymmdd formate or yyymmddhh** formate or cdtime.comptime object from startdate to enddate with stepday or stephour. we can set the cdtime.calendarName to generate the date(s) in between the given range.

tRange means timeRange

Condition : The startdate and enddate must be either yyyymmdd or yyyymmddhh or cdtime.comptime object or cdtime.comptime string formate. We can use either stepday or stephour at a time. Can not use both(stepday and stephour) at the same time. if enddate is higher than the startdate, then stepday/ stephour must be +ve. if enddate is lower than the startdate, then stepday/stephour must be -ve. By default stepday is 0 day and stephour is 0 hour.

Inputs : startdate, enddate stepday to skip the days. stephour to skip the hours. calendarName is one of the cdtime calendar type returnType is either 's' or 'c'. if 's' means the return date should be in string type. if 'c' means the return date should be cdtime type itself. Default returnType takes 'c' as arg. returnHour is either 'y' or 'yes' or 'n' or 'no'. If 'y/yes' means it should return the hour (yyymmddhh), if returnType is 's'. If 'n/no' means it shouldnt return hour (yyyymmdd), if returnType is 's'. Default returnHour takes 'y' as arg.

Outputs : It should return a list which contains the date(s) in between the startdate and enddate including both the startdate and enddate.

Usage :

example1 :

```
>>> tRange(20110407, 20110410, stepday = 1, returnType = 's')
['2011040700', '2011040800', '2011040900', '2011041000']
```

..note:: Here returnType is 's' and returnHour is 'yes' by

default. So it should return with hour (yyymmddhh).

```
>>> tRange(20110407, 20110410, stepday = 1, returnType = 's',
                                            returnHour = 'no')
['20110407', '20110408', '20110409', '20110410']
```

..note:: Here we passed returnHour is 'no'. So it should not return hour. (only yyyymmdd)

example2 :

```
>>> tRange(20120227, 20120301 , stepday = 1,
                   calendarName = cdtime.NoLeapCalendar,
                   returnType = 's', returnHour = 'no')
['20120227', '20120228', '20120301']
```

Note: In the example 2, 2012 is leap year, since we passed cdtime.NoLeapCalendar it generated without 29th day in feb 2012. we can use stepday as any +ve integer number. The generator returns both startdate and enddate also.

---

example3 :

```
>>> tRange(startdate = 20110407, enddate = 20110410)
[]
```

Note: In this example it should not generate any dates in between the startdate and enddate, since we didnt pass either stepday or stephour.

example4 :

```
>>> tRange(startdate = cdtime.comptime(2011,04,07),
...          enddate = cdtime.comptime(2011,04,10), stepday = 1,
...                                                 returnType = 'c')
[2011-4-7 0:0:0.0, 2011-4-8 0:0:0.0, 2011-4-9 0:0:0.0,
                                    2011-4-10 0:0:0.0]
```

Note: Here the input dates are cdtime.comptime object itself.

example5 :

```
>>> tRange(startdate = cdtime.comptime(2011,04,11),
...          enddate = cdtime.comptime(2011,04,7), stepday = -1,
...          returnType = 'c')
[2011-4-11 0:0:0.0, 2011-4-10 0:0:0.0, 2011-4-9 0:0:0.0,
                     2011-4-8 0:0:0.0, 2011-4-7 0:0:0.0]
```

Note: In this example we have passed startdate is higher than then enddate, So we must have to pass the stepdays in -ve sign.

example 6:

```
>>> tRange('2011-4-7', '2011-4-10', stepday = 1)
[2011-4-7 0:0:0.0, 2011-4-8 0:0:0.0, 2011-4-9 0:0:0.0,
                                    2011-4-10 0:0:0.0]
```

..note:: Here it genearates the cdtime.comptime object by default

We passed the inputs are cdtime.comptime date string formate (yyyymmdd)only.

```
>>> tRange('2011-4-7 12:0:0.0', '2011-4-10 0:0:0.0',
...                                                 stephour = 12)
[2011-4-7 12:0:0.0, 2011-4-8 0:0:0.0, 2011-4-8 12:0:0.0,
 2011-4-9 0:0:0.0, 2011-4-9 12:0:0.0, 2011-4-10 0:0:0.0]
```

..note:: Here we passed 12:0:0.0 hours in startdate, 0:0:0.0 hours in enddate, and stephour as 12. Note here the input dates are not cdtime.comptime object. But those are cdtime.comptime string formate (yyymmddhh).

example 7:

```
>>> tRange('2011040712', '2011-4-10 10:0:0.0', stephour = 12)
[2011-4-7 12:0:0.0, 2011-4-8 0:0:0.0, 2011-4-8 12:0:0.0,
 2011-4-9 0:0:0.0, 2011-4-9 12:0:0.0, 2011-4-10 0:0:0.0]
```

..note:: Here startdate as in 'yyymmddhh' string formate and enddate as in cdtime.comptime string formate. you can play with combination of differnt inputs.

Written by : Arulalan.T

Date : 23.08.2011

timestr2comp(*date*)

timestr2comp() **(page 32): To convert date from yyyymmdd[hh] formate into** cdtime.comptime formate

Condition : passing date must be yyyymmdd formate in either int or str

Inputs: date in yyyymmdd formate or yyyymmddhh formate. i.e. hour(hh) is optional.

Outputs: It should return the date in cdtime.comptime object type

Usage:

example1:

```
>>> timestr2comp(20110423)
2011-4-23 0:0:0.0
  .. note:: It should return as cdtime.comptype. Here we didnt
            pass the hour. i.e only yyyymmdd formate
```

example2:

```
>>> timestr2comp(2011082010)
2011-8-20 10:0:0.0
  ..note:: Here it should return cdtime with hours also.
           We passed yyyymmddhh formate. i.e include hh
```

example3:

```
>>> timestr2comp(2011082023)
2011-8-20 23:0:0.0
  ..note:: we cannot pass 24 as hour here. Max 23 hours only.
```

Written by: Arulalan.T

Date: 23.04.2011 Updated : 21.08.2011

xtRange(*startdate*, *enddate*, *stepday=0*, *stephour=0*, *calendarName=None*, *returnType='c'*, *returnHour='y'*)

xtRange() **(page 32): generate the dates in yyyymmdd formate or yyymmddhh** formate or cdtime.comptime object from startdate to enddate with stepday or stephour. we can set the cdtime.calendarName to generate the date(s) in between the given range.

xtRange means xtimeRange

Condition : The startdate and enddate must be either yyyymmdd or yyyymmddhh or cdtime.comptime object or cdtime.comptime string formate. We can use either stepday or stephour at a time. Can not use both(stepday and stephour) at the same time. if enddate is higher than the startdate, then stepday/ stephour must be +ve. if enddate is lower than the startdate, then stepday/stephour must be -ve. By default stepday is 0 day and stephour is 0 hour.

Inputs : startdate, enddate stepday to skip the days. stephour to skip the hours. calendarName is one of the cdtime calendar type returnType is either 's' or 'c'. if 's' means the return date should be in string type. if 'c' means the return date should be cdtime type itself. Default returnType takes 'c' as arg. returnHour is either 'y' or 'yes' or 'n' or 'no'. If 'y/yes' means it should return the hour (yyymmddhh), if returnType is 's'. If 'n/no' means it shouldnt return hour (yyyymmdd), if returnType is 's'. Default returnHour takes 'y' as arg.

Outputs : It should return a generator not as list. Using this generator we can produce the date(s) in between the startdate and enddate including both the startdate and enddate.

Usage :

example1 :

```
>>> gen = xtRange(20110407, 20110410, stepday = 1,
...                                          returnType = 's')
>>> for i in gen:
...     print i
...
2011040700
2011040800
2011040900
2011041000
```

..note:: Here returnType is 's' and returnHour is 'yes' by

default. So it should return with hour (yyymmddhh).

```
>>> gen = xtRange(20110407, 20110410, stepday = 1,
...                        returnType = 's', returnHour = 'no')
>>> for i in gen:
...     print i
...
20110407
20110408
20110409
20110410
```

..note:: Here we passed returnHour is 'no'. So it should not return hour. (only yyyymmdd)

example2 :

```
>>> gen = xtRange(20120227, 20120301 , stepday = 1,
...                   calendarName = cdtime.NoLeapCalendar,
...                    returnType = 's', returnHour = 'no')
>>> for i in gen:
...     print i
...
20120227
20120228
20120301
```

Note: In the example 2, 2012 is leap year, since we passed cdtime.NoLeapCalendar it generated without 29th day in feb 2012. we can use stepday as any +ve integer number. The generator returns both startdate and enddate also.

---

example3 :

```
>>> gen = xtRange(startdate = 20110407, enddate = 20110410)
>>> for i in gen:
...     print i
...
>>>
```

Note: In this example it should not generate any dates in between the startdate and enddate, since we didnt pass either stepday or stephour.

example4 :

```
>>> gen = xtRange(startdate = cdtime.comptime(2011,04,07),
...          enddate = cdtime.comptime(2011,04,10), stepday = 1,
...                                                 returnType = 'c')
>>> for i in gen:
...     print i
...
2011-4-7 0:0:0.0
2011-4-8 0:0:0.0
2011-4-9 0:0:0.0
2011-4-10 0:0:0.0
```

Note: Here the input dates are cdtime.comptime object itself.

example5 :

```
>>> gen = xtRange(startdate = cdtime.comptime(2011,04,11),
...          enddate = cdtime.comptime(2011,04,7), stepday = -1,
...          returnType = 'c')
>>> for i in gen:
...     print i
...
2011-4-10 0:0:0.0
2011-4-9 0:0:0.0
2011-4-8 0:0:0.0
2011-4-7 0:0:0.0
```

Note: In this example we have passed startdate is higher than then enddate, So we must have to pass the stepdays in -ve sign.

example 6:

```
>>> gen = xtRange('2011-4-7', '2011-4-10', stepday = 1)
>>> for i in gen:
...     print i
...
2011-4-7 0:0:0.0
2011-4-8 0:0:0.0
2011-4-9 0:0:0.0
2011-4-10 0:0:0.0
```

..note:: Here it genearates the cdtime.comptime object by default

We passed the inputs are cdtime.comptime date string formate (yyyymmdd)only.

```
>>> gen = xtRange('2011-4-7 12:0:0.0', '2011-4-10 0:0:0.0',
                                                    stephour = 12)
>>> for i in gen:
...     print i
...
```

```
2011-4-7 12:0:0.0
2011-4-8 0:0:0.0
2011-4-8 12:0:0.0
2011-4-9 0:0:0.0
2011-4-9 12:0:0.0
2011-4-10 0:0:0.0
```

..note:: Here we passed 12:0:0.0 hours in startdate, 0:0:0.0 hours in enddate, and stephour as 12. Note here the input dates are not cdtime.comptime object. But those are cdtime.comptime string formate (yyymmddhh).

example 7:

```
>>> gen = xtRange('2011040712', '2011-4-10 10:0:0.0',
...                                                stephour = 12)
>>> for i in gen:
...     print i
...
2011-4-7 12:0:0.0
2011-4-8 0:0:0.0
2011-4-8 12:0:0.0
2011-4-9 0:0:0.0
2011-4-9 12:0:0.0
2011-4-10 0:0:0.0
```

..note:: Here startdate as in 'yyymmddhh' string formate and enddate as in cdtime.comptime string formate. you can play with combination of differnt inputs.

Written by : Arulalan.T

Date : 07.04.2011

Updated : 23.08.2011

3.3 Plot Utils

The plot (page 35) module has the properties to plot the vcs vector with some default template look out.

User can control the reference point, scale of arrow marks of the vector plot.

3.3.1 plot

class plot.**reference_std_dev**

This class implements the type: standard deviation of a reference variable. It is just a float value with one method that will be used in the computation of a test variable RMS.

compute_RMS_function(*s*, *R*)

Compute and return the centered-pattern RMS of a test variable from its standard-deviation and correlation.

Input: self: reference-variable standard deviation s: test variable standard deviation R: test-variable correlation with reference variable

plot.**vectorPlot**(*u*, *v*, *name*, *path=None*, *reference=20.0*, *scale=1*, *interval=1*, *svg=1*, *png=0*, *latlabel='lat5'*, *lonlabel='lon5'*, *style='portrait'*)

vectorPlot() (page 35): Plotting the vector with some default preferences.

Input [u - u variable] v - v variable name - name to plot on the top of the vcs path - path to save as the image file. reference - vector reference. Default it takes 20.0 (i.e 2 degree) scale - scaling of the arrow mark in vector plot interval - slicing the data to reduce the density (noise)

in the vector plot, with respect to the interval. Default it takes 1. (i.e. doesnot affect the u & v)

svg - to save image as svg png - to save image as png

Condition [u must be 'u variable' and v must be 'v variable'.] name must pass to set the name on the vector vcs path is not passed means, it takes current workig directory reference must be float. interval not be 0.

Usage : using this function, user can plot the vector.

user can control the reference point of the vector, and scale length of the arrow marks in plot.

Also can control the u and v data shape by interval.

filename should be generated from the 'name' passed by the user, just replacing the space into underscore '_'.

if svg and png passed 1, the image will be saved with these extensions in the filename.

Written By : Arulalan.T

Date : 26.07.2011

3.4 More

More utilities will be added and optimized in near future.

CHAPTER

FOUR

DOCUMENTATION OF DIAGNOSIS SOURCE CODE

The diagnosis package contains the following modules.

4.1 Monthly Progress

The monthly progress of diagnosis (page 37) are listed below.

These monthly progress will be automated.

4.1.1 Climatology

climatology_utils.**dailyClimatology**(*varName*, *infile*, *outfile*, *leapday=False*, ***kwarg*)

dailyClimatology [It will create the daily climatolgy and stored] in the outfile.

Inputs: varName : variable name to extract from the input file infile : Input file absolute path outfile : outfile absolute path (will be created in write mode) leapday : False | True

If it is True, then it will create 366 days climatolgy (include 29th feb) If it is False, then it will create 365 days climatolgy

KWargs:

ovar [out varName. If it is passed then the climatology variable] name will be set as ovar. Otherwise the input varName will be set to it.

squeeze : 1 (it will squeeze single dimension in the climatolgy)

todo : need to set year 1 for 366 days climatology.

Written By : Arulalan.T Date : 13.08.2013

climatology_utils.**monthlyClimatology**(*varName*, *infile*, *outfile*, *memory='low'*, ***kwarg*)

monthlyClimatology [It will create the monthly climatolgy.] Its timeaxis dimension length is 12.

memory ['low'/'high'.] If it is low, then it compute climatology in optimized manner by extracting full time-series data of particular latitude, longitude & level points by loop throughing each latitude, longitude & level axis. It needs low RAM memory.

If it is 'high', then it load the whole data from the input file and compute climatology. It needs high RAM memory.

KWargs:

ovar [out varName. If it is passed then the climatology variable] name will be set as ovar. Otherwise the input varName will be set to it.

squeeze : 1 (it will squeeze single dimension in the climatolgy)

todo : need to give option to create 366 days climatology.

Written By : Arulalan.T Date : 13.08.2013

4.1.2 Month Mean

The word *Mean* means average of the data. The average will be taken over the month time axis is called month mean.

The below script *compute_month_mean.py* should explain more how we are implementing monthly mean and generating the nc files.

compute_month_mean.**genMonthMeanDirs**(*modelname*, *modelpath*, *modelhour*)

genMonthMeanDirs() [It should generate the directory structure] whenever it needs. It reads the timeAxis information of the model data xml file(which is updating it by cdscan), and once the full months is completed, then it should check either that month directory is empty or not.

case 1: If that directory is empty means, it should call the function called *genMonthMeanFiles*, to calculate the mean analysis and anomaly for that month and should store the processed files in side that directory.

case 2: If that directory is non empty means, **have to update***

Inputs [modelname is the model data name, which will become part of the] directory structure. modelpath is the absolute path of data where the model xml files are located. climatolgyyear is the year of climatolgy data. climregridpath is the absolute path of the climatolgy regridded path w.r.t to this model data resolution (both horizontal and vertical) climpfilename is the climatolgy Partial File Name to combine the this passed name with (at the end) of the climatolgy var name to open the climatolgy files.

Outputs [It should create the directory structure in the processfilesPath] and create the processed nc files.

Written By : Arulalan.T

Date : 01.12.2011

compute_month_mean.**genMonthMeanFiles**(*meanMonthPath*, *monthdate*, *year*, *typehour*, ***model*)

genMonthMeanFiles() [It should calculate monthly mean analysis &] monthly mean forecast hours value for the month (of year). Finally stores it as nc files in corresponding directory path which are passed in this function args.

Inputs [meanMonthPath is the absolute path where the processed month mean] analysis & fcst hour nc files are going to store. monthdate (which contains monthname, startdate & enddate) and year are the inputs to extract the monthly data. typehour is tuple which has the type key character and fcst hour to create sub directories inside mean directory.

KWargs: modelName, modelXmlPath, modelXmlObj

modelName is the model data name which will become part of the process nc files name. modelPath is the absolute path of data where the model xml files are located. modelXmlObj is an instance of the GribXmlAccess class instance. If we are passing modelXmlObj means, it will be optimized one when we calls this same function for same model for different months.

We can pass either modelXmlPath or modelXmlObj KWarg is enough.

Outputs [It should create monthly mean analysis and monthly mean forecast] hours for all the available variables in the vars.txt file & store it as nc file formate in the proper directories structure (modelname, process name, year, month and then [Analysis or hours] hierarchy).

Written By : Arulalan.T

Date : 08.09.2011 Updated : 06.12.2011

4.1.3 Month Anomaly

Anomaly means the difference between the model analysis and climatology.

Monthly Anomaly : Take the difference between the model analysis data of the particular month and the climatology data of the corresponding month.

Anomaly = Analysis - Climatology

The below script *compute_month_anomaly.py* should explain more how we are implementing monthly anomaly and generating the nc files.

compute_month_anomaly.**genMonthAnomalyDirs**(*modelname*, *modelpath*, *climregridpath*, *climpfilename*, *climatologyyear*)

genMonthAnomalyDirs() [It should generate the directory structure] whenever it needs. It reads the timeAxis information of the model data xml file(which is updating it by cdscan), and once the full months is completed, then it should check either that month directory is empty or not.

case 1: If that directory is empty means, it should call the function called *genMonthAnomalyFiles*, to calculate the mean analysis and anomaly for that month and should store the processed files in side that directory.

case 2: If that directory is non empty means, **have to update***

Inputs [modelname is the model data name, which will become part of the] directory structure. modelpath is the absolute path of data where the model xml files are located. climregridpath is the absolute path of the climatolgy regridded path w.r.t to this model data resolution (both horizontal and vertical) climpfilename is the climatolgy Partial File Name to combine the this passed name with (at the end) of the climatolgy var name to open the climatolgy files. climatolgyyear is the year of climatolgy data.

Outputs [It should create the directory structure in the processfilesPath] and create the processed nc files.

Written By : Arulalan.T

Date : 01.12.2011

compute_month_anomaly.**genMonthAnomalyFiles**(*meanAnomalyPath*, *meanAnalysisPath*, *climRegridPath*, *climPFileName*, *climatologyYear*, *monthdate*, *year*, ***model*)

genMonthAnomalyFiles() [It should calculate monthly mean anomaly] from the monthly mean analysis and monthly mean climatolgy, for the month (of year) and process it. Finally stores it as nc files in corresponding directory path which are passed in this function args.

Inputs [meanAnomalyPath is the absolute path where the processed mean] anomaly nc files are going to store. meanAnalysisPath is the absolute path where the processed mean analysis nc files were already stored. climRegridPath is the absolute path where the regridded monthly mean climatologies (w.r.t the model vertical resolution) nc files were already stored. climPFileName is the partial nc filename of the climatolgy. climatologyYear is the year of the climatolgy to access it. monthdate (which contains monthname, startdate & enddate) and year are the inputs to extract the monthly data.

KWargs: modelName, modelXmlPath, modelXmlObj

modelName is the model data name which will become part of the process nc files name. modelPath is the absolute path of data where the model xml files are located. modelXmlObj is an instance of the GribXmlAccess class instance. If we are passing modelXmlObj means, it will be optimized one when we calls this same function for same model for different months.

We can pass either modelXmlPath or modelXmlObj KWarg is enough.

Outputs [It should create mean anomaly for the particular variables which] are all set the clim_var option in the vars.txt file. Finally store it as nc file formate in the proper directories structure (modelname, process name, year and then month hierarchy).

Written By : Arulalan.T

Date : 08.09.2011 Updated : 07.12.2011

4.1.4 Month Fcst Sys Error

Forecast Systematic Error means the difference between the model forecast hour data and model analysis.

This also called as *Fcst Sys Err*.

Month Fcst Sys Error : Take the difference between the model forecast hour data of the particular month and the model analysis data of the same month.

Fcst Sys Err = Model Fcst Hour Data - Model Analysis

The below script *compute_month_fcst_sys_error.py* should explain more how we are implementing monthly fcst sys err and generating the nc files.

compute_month_fcst_sys_error.**genMonthFcstSysErrDirs**(*modelname*, *modelpath*, *modelhour*)

genMonthFcstSysErrDirs() [It should generate the directory structure] whenever it needs. It reads the timeAxis information of the model data xml file(which is updating it by cdscan), and once the full months is completed, then it should check either that month directory is empty or not.

case 1: If that directory is empty means, it should call the function called *genMonthFcstSysErrFiles*, to calculate the mean analysis and fcstsyserr for that month and should store the processed files in side that directory.

case 2: If that directory is non empty means, **have to update***

Inputs [modelname is the model data name, which will become part of the] directory structure. modelpath is the absolute path of data where the model xml files are located. modelhour is the list of model data hours, which will become part of the directory structure.

Outputs [It should create the directory structure in the processfilesPath] and create the processed nc files.

Written By : Arulalan.T

Date : 08.12.2011

compute_month_fcst_sys_error.**genMonthFcstSysErrFiles**(*meanFcstSysErrPath*, *meanPath*, *monthdate*, *year*, *modelhour*, ***model*)

genMonthFcstSysErrFiles() [It should calculate mean analysis &] fcstsyserr for the passed month (of year) and process it. Finally stores it as nc files in corresponding directory path which are passed in this function args.

Inputs [meanFcstSysErrPath is the absolute path where the processed mean] fcstsyserr nc files are going to store. meanPath is the absolute path (partial path) where the processed monthly mean analysis and fcst hour nc files were stored already. monthdate (which contains monthname, startdate & enddate) and year are the inputs to extract the monthly data. modelhour is the list of model data hours, which will become part of the directory structure.

KWargs: modelName, modelXmlPath, modelXmlObj

modelName is the model data name which will become part of the process nc files name. modelPath is the absolute path of data where the model xml files are located. modelXmlObj is an instance of the GribXmlAccess class instance. If we are passing modelXmlObj means, it will be optimized one when we calls this same function for same model for different months.

We can pass either modelXmlPath or modelXmlObj KWarg is enough.

Outputs [It should create mean forecast systematic error for all the] available variables in the vars.txt file & store it as nc file formate in the proper directories structure (modelname, process name, year, month and then hours hierarchy).

Written By : Arulalan.T

Date : 08.09.2011 Updated : 08.12.2011

4.2 Seasonly Progress

The seasonly progress of diagnosis (page 37) are listed below.

- Season Mean (page 41)
- Collect Season Fcst Rainfall (page 43)
- Region Statistical Score (page 43)
- Season Statistical Score Spatial Distribution (page 44)

These season progress will be automated with respect to the given season as input in the configure.txt.

The word *season* means the consecutive months.

For eg: JJAS contains June, July, August, September.

4.2.1 Season Mean

The word *Mean* means average of the data. The average will be taken over the season time axis is called season mean.

The below script *compute_season_mean.py* should explain more how we are implementing seasonly mean and generating the nc files.

compute_season_mean.**genMeanAnlFcstErrDirs**(*modelname*, *modelpath*, *modelhour*)

genMeanAnlFcstErrDirs() [It should create the directory structure] whenever it needs. It reads the timeAxis information of the model data xml file(which is updating it by cdscan), and once the full seasonal months are completed, then it should check either that season directory is empty or not.

case 1: If that directory is empty means, it should call the function called *genSeasonMeanFiles*, to calculate the mean analysis and fcstsyserr for that season and should store the processed files in side that directory.

case 2: If that directory is non empty means, **have to update***

Inputs [modelname is the model data name, which will become part of the] directory structure. modelpath is the absolute path of data where the model xml files are located. modelhour is the list of model data hours, which will become part of the directory structure.

Outputs [It should create the directory structure in the processfilesPath] and create the processed nc files.

Written By : Arulalan.T

Date : 08.12.2011

compute_season_mean.**genSeasonMeanFiles**(*meanSeasonPath*, *meanMonthPath*, *seasonName*, *seasonMonthDate*, *year*, *Type*, ***model*)

genSeasonMeanFiles() [It should calculate the seasonly mean for] either analysis or forecast systematic error. It can be choosed by the Type argment. Finally stores it as nc files in corresponding directory path which are passed in this function args.

Inputs [meanSeasonPath is the absolute path where the processed season] mean analysis or forecast systematic error nc files are going to store. Inside the fcst hours directories will be created in this path, if needed.

meanMonthPath is the absolute path where the processed monthly mean analysis or monthly mean fcstsyserr nc files were stored, already. seasonName is the name of the season. seasonMonthDate(list of months which contains monthname, startdate & enddate) for the season. year is the part of the directory structure. Type is either 'a' for analysis or 'f' for fcstsyserr.

KWargs: modelName, modelXmlPath, modelXmlObj

modelName is the model data name which will become part of the process nc files name. modelHour is the model hours as list, which will become part of the directory structure. modelPath is the absolute path of data where the model xml files are located. modelXmlObj is an instance of the GribXmlAccess class instance. If we are passing modelXmlObj means, it will be optimized one when we calls this same function for same model for different months.

We can pass either modelXmlPath or modelXmlObj KWarg is enough.

Process [This function should compute the seasonly mean for analysis and] fcstsyserr by just opening the monthly mena analysis/fcstsyserr nc files (according to the season's months) and multiply the monthly mean into its weights value. So that monthly mean data should become monthly full data (not mean). Then add it together for the season of months. Finally takes the average by just divide the whole season data by sum of monthly mean weights.

So it should simplify our life, just extracting data which

timeAxis length is 4, for 4 months in season. (eg JJAS).

Outputs [It should create seasonly mean analysis and forecast systematic] error for all the available variables in the vars.txt file, and store it as nc file in the proper directory structure (modelname, process name, year, season, and/or hours hierarchy).

Written By : Arulalan.T

Date : 08.12.2011

4.2.2 Collect Season Fcst Rainfall

This script *collect_season_fcst_rainfall.py* should collect the whole season forecast hourly rainfall with respect to the hours of season. Finally it should create the nc files for every fcst hours that contains the fcst rainfall with needed time axis to compute the further process.

Written by: Dileepkumar R JRF- IIT DELHI

Date: 23.06.2011

Updated By : Arulalan.T Date: 14.09.2011 Date: 19.10.2011

collect_season_fcst_rainfall.**createSeaonFcstRainfallData**(*modelname*, *modelpath*, *modelhour*, *rainfallPath*, *rainfallXmlName=None*)

createSeaonFcstRainfallData(): It should create model hours forecast rainfall data nc files, in side the 'StatiScore' directory of processfilesPath in hierarchy structure. The fcst rainfall timeAxis are in partners timeAxis w.r.t observation rainfall and fcst hours.

4.2.3 Region Statistical Score

The below script *compute_region_statistical_score.py* should compute the various statistical scores regional wise and make the nc files.

Here regions are 'Central India', 'Peninsular India', etc. That is region name/variable defines the latitude and longitude range.

Example: Let 'ts'(Threat Score) is a statistical score, we are calculate this for different regions.

Written by: Dileepkumar R JRF- IIT DELHI

Date: 02.08.2011;

Updated By : Arulalan.T Date : 16.09.2011

compute_region_statistical_score.**genStatisticalScore**(*modelname*, *modelhour*, *seasonName*, *year*, *procStatiSeason*, *procRegion*, *plotCSV*, *rainfallPath*, *rainfallXmlName=None*)

genStatisticalScore() [It should compute the statistical scores] like " Threat Score, Equitable Threat Score, Accuracy(Hit Rate), Bias Score, Probability Of Detection, False Alarm Rate, Odds Ratio, Probability Of False Detection, Kuipers Skill Score, Log Odd Ratio, Heidke Skill Score, Odd Ratio Skill Score, & Extreame Dependency Score" by accessing the observation and forecast data.

It should compute the statistical scores for different regions.

Finally it should store the scores variable in both csv and nc files in appropriate directory hierarchy structure.

..note:: We are replacing the -ve values with zeros of both the observation and fcst data to make correct statistical scores.

Inputs [modelname, modelhour, seasonName, year are helps to generate the] path. procStatiSeason is the partial path of process statistical score season path. procRegion is an absolute path to store the nc files plotCSV is an absolute path to store the csv files. rainfallPath is the path of the observation rainfall. rainfallXmlName is the name of the xml file name, it is an optional one. By default it takes 'rainfall_regrided.xml'

Outputs [It should store the computed statistical scores for all the] regions and store it as both ncfile and csv files in the appropriate directory hierarchy structure.

compute_region_statistical_score.**genStatisticalScoreDirs**(*modelname*, *modelhour*, *rainfallPath*, *rainfallXmlName=None*)

Func *genStatisticalScoreDirs* : It should generate the appropriate directory hierarchy structure for 'StatiScore' in both the processfiles path and plotgraph path. In plotgraph path, it should create 'CSV' directory to store the 'statistical scores' in csv file formate.

This function should call the *genStatisticalScore* function to compute and statistical score.

Inputs [modelname and modelhour are the part of the directory hierarchy] structure. rainfallPath is the path of the observation rainfall. rainfallXmlName is the name of the xml file name.

4.2.4 Season Statistical Score Spatial Distribution

The below script *compute_season_stati_score_spatial_distribution.py* should compute the various statistical scores w.r.t each & every lat, lon location of particular region.

Example: Let 'TS'(Threat Score) is a statistical score, we are calculate this spatially.

Written by: Dileepkumar R JRF- IIT DELHI

Date: 02.09.2011;

Updated By : Arulalan.T Date : 17.09.2011 Date : 06.10.2011

compute_season_stati_score_spatial_distribution.**genStatisticalScorePath**(*modelname*, *modelhour*, *rainfallPath*, *rainfallXmlName=None*)

`genStatisticalScorePath()`: It should make the existing path of process files statistical score. Also if that is correct path means, it should calls the function *genStatisticalScoreSpatialDistribution* to compute the statistical score in spatially distributed way.

Inputs [modelname and modelhour are the part of the directory hierarchy] structure.

compute_season_stati_score_spatial_distribution.**genStatisticalScoreSpatialDistribution**(*mode*
mod-
el-
hour,
sea-
son,
year,
sta-
tiSea
son-
Path,
rain-
fall-
Path,
rain-
fal-
lXml-
Nam
lat=
lon=

Func *genStatisticalScoreSpatialDistribution* : It should compute the statistical scores like " Threat Score, Equitable Threat Score, Accuracy(Hit Rate), Bias Score, Probability Of Detection, Odds Ratio, False Alarm Rate, Probability Of False Detection, Kuipers Skill Score, Log Odd Ratio, Heidke Skill Score, Odd Ratio Skill Score, & Extreame Dependency Score" in spatially distributed way (i.e compute scores in each and every lat & lon points) by accessing the observation and forecast data.

Inputs [modelname, modelhour, season, year are helps to generate the] path. statiSeasonPath is the partial path of process statistical score season path. rainfallPath is the path of the observation rainfall. rainfallXml-Name is the name of the xml file name, it is an optional one. By default it takes 'rainfall_regrided.xml'. lat, lon takes tuple args. If we passed it, then the model lat, lon should be shrinked according to the passed lat,lon. Some times it may helpful to do statistical score in spatially distributed in particular region among the global lat,lon.

Outputs [It should store the computed statistical scores in spatially] distributed way for all the modelhour(s) as nc files in the appropriate directory hierarchy structure.

4.3 Diagnosis Plots

The diagnosis plots of diagnosis (page 37) are listed below.

- Winds Plots (page 45)
- Iso Plots (page 48)
- Statistical Score Bar Plots (page 50)
- Statistical Score Spatial Distribution Plots (page 51)

4.3.1 Winds Plots

Generate the vector plots using U and V component of the wind data. The below script *generate_winds_plots.py* should generates the wind plots and save it as either png or jpg or svg in the suitable directory for month and season wise.

generate_winds_plots.**editVectorPlot**(*modelname*, *processtype*, *year*, *monthseason*, *hour=None*, *level=None*, *region=None*, *reference=20*, *scale=1*, *interval=4*, *svg=0*, *png=1*, *latlabel='lat5'*, *lonlabel='lon5'*, *outpath=None*)

editVectorPlot() (page 45) [To edit/reproduce any particular vector plots by] passing modelname, processtype, year, month/season, hour, level(s), region, reference point of vector, scale of vector arrow markers, interval of the U & V datasets, svg & png options.

Inputs [modelname is the part of the directory structure.] processtype is any one of the processes.for eg : 'Mean Analysis', 'Mean Fcst' or 'Anomaly' or 'FcstSysErr', etc., year is year in string type. monthseason is either month name or season name. It should find out either it is month or season and make the correct path. hour is the hour string which is the part of the directory structure only for 'FcstSysErr' and 'Mean Fcst' process type.

level is either single level, or list of levels or 'all'. 'all' means, it takes all the availableLevels from the variables. level value must be int, float only. Not be string, other than 'all' keyword. region is the region variable which should cut particular region shape from the global data. By default it is None, i.e. takes global data region itself.

reference is the reference points to be plotted in the vcs vector plot. It must be float only. scale is the length of the arrow markers in the vcs vector plot. interval is the integer value, to split the U and V datasets, to make clear view of the vector plot. By default it takes 4.

svg is the flag. If flag is set, then the vector plot should be saved as svg formate. By default it is 0.

png is the flag. If flag is set, then the vector plot should be saved as png formate. By default it is 1.

outpath is the absolute path, where the generated plots should be stored. By default, it is None, that is it should save in the appropirate plotsgraphs directory, which is generated by this function.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**genMonthAnomalyDirs**(*modelName*, *availableMonths*)

genMonthAnomalyDirs() (page 46): It should generate the directory hierarichy structure of month anomaly in the plotsgraphspath. And calls the function genVectorPlots to make vector plots and save it inside the appropirate directory, by reading the u, v nc files of the appropirate process month anomaly files path.

Inputs [modelName is the one of the directories name.] availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file, and generate the vector plots to those levels only.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**genMonthMeanDirs**(*modelName*, *availableMonths*)

genMonthMeanDirs() (page 46): It should generate the directory hierarichy structure of month mean in the plotsgraphspath. And calls the function genVectorPlots to make vector plots and save it inside the appropirate directory, by reading the u, v nc files of the appropirate process month mean files path.

Inputs [modelName is the one of the directories name.] availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file, and generate the vector plots to those levels only.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**genSeasonFcstSysErrDirs**(*modelName*, *modelHour*, *availableMonths*)

genSeasonFcstSysErrDirs() (page 47): It should generate the directory hierarichy structure of season fcstsyserr in the plotsgraphspath. And calls the function genVectorPlots to make vector plots and save it inside the appropirate directory, by reading the xml file of the appropirate process season fcstsyserr files path.

Inputs [modelName is the one of the directories name.] modelHour is the one of the directories name. availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file, and generate the vector plots to those levels only.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**genSeasonMeanDirs**(*modelName*, *availableMonths*)

genSeasonMeanDirs() (page 47): It should generate the directory hierarichy structure of season mean in the plotsgraphspath. And calls the function genVectorPlots to make vector plots and save it inside the appropirate directory, by reading the xml file of the appropirate process season mean files path.

Inputs [modelName is the one of the directories name.] availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file, and generate the vector plots to those levels only.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**genVectorPlots**(*uvar*, *vvar*, *upath=None*, *vpath=None*, *xmlpath=None*, *outpath=None*, *month=None*, *date=None*, *level=None*, *region=None*, *reference=20.0*, *scale=1*, *interval=4*, *svg=0*, *png=1*, *latlabel='lat5'*, *lonlabel='lon5'*, *style='portrait'*)

genVectorPlots() (page 47): It should generate the vector plots in vcs background and save it as png(by default) inside the outpath, with some default vector properties like reference, scale and interval.

Inputs [uvar is the 'u' variable name] vvar is the 'v' variable name upath is the 'u' nc file absolute path. vpath is the 'v' nc file absolute path. xmlpath is the xml file absolute path which must contains the u and v vars. outpath is the absolute path, where the generated plots should be stored. By default, it is None. It means, it should save in the current working directory itself. level is either single level, or list of levels or 'all'. 'all' means, it takes all the availableLevels from the variables. level value must be int, float only. Not be string, other than 'all' keyword. region is the region variable which should cut particular region shape from the global data. By default it is None, i.e. takes global data region itself.

reference is the reference points to be plotted in the vcs vector plot. It must be float only. scale is the length of the arrow markers in the vcs vector plot. interval is the integer value, to split the U and V datasets, to make clear view of the vector plot. By default it takes 4.

svg is the flag. If flag is set, then the vector plot should be saved as svg formate. By default it is 0.

png is the flag. If flag is set, then the vector plot should be saved as png formate. By default it is 1.

Condition [If we passed xmlpath, then we no need to pass upath and vpath] args. uvar and vvar must be available in the passed filepath.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

generate_winds_plots.**getProcessPath**(*modelname*, *processtype*, *year*, *monthseason*, *hour=None*)

getProcessPath() (page 48): By passing fewer args and get the correct and absolute path of the process files, which generated by automated or manual for the purpose of this diagnosis.

Inputs [modelname is the part of the directory structure.] processtype is any one of the processes.for eg : 'Mean Analysis', or 'Mean Fcst' or 'Anomaly' or 'FcstSysErr', etc., year is year in string type. monthseason is either month name or season name. It should find out either it is month or season and make the correct path. hour is the hour string which is the part of the directory structure only for 'FcstSysErr' and 'Mean Fcst' process type.

Outputs [Return the absolute path of the process files, only if that] directory is exists. Other wise it raise error.

To Do [Need to decide about, either it should raise error, or it should] return None, if wrong args passed or process directory doesnot exists.

Written By : Arulalan.T

Date : 11.09.2011 Updated: 11.12.2011

4.3.2 Iso Plots

Generate the iso plots for the iso variables which are all set in the 'vars.txt' file.

The below script *generate_iso_plots* should generates the following kind of plots.

- iso line
- iso fill
- iso fill line

Finally save the generated iso plots as either png or jpg or svg in the suitable directory for month and season wise.

generate_iso_plots.**genIsoFillLinePlots**(*var*, *key*, *isoLevels*, *xmlpath=None*, *path=None*, *outpath=None*, *month=None*, *date=None*, *level='all'*, *region=None*, *svg=0*, *png=1*)

genIsoFillLinePlots() (page 48): It should generate the isoFillLine plots in vcs background and save it as png(by default) inside the outpath, with isoLevels (passed by user) and isoColors (default).

Inputs [var is the variable name.] key to identify, it is which variable to make plot name. isoLevels is the levels to plot isoFillLine and set the legend levels in vcs. xmlpath is the xml file absolute path. path is the nc file absolute path. pass any one (path or xmlpath)

outpath is the absolute path, where the generated plots should be stored. By default, it is None. It means, it should save in the current working directory itself. level is either single level, or list of levels or 'all'. 'all' means, it takes all the availableLevels from the variables. level value must be int, float only. Not be string, other than 'all' keyword. region is the region variable which should cut particular region shape from the global data. By default it is None, i.e. takes global data region itself.

svg is the flag. If flag is set, then the vector plot should be saved as svg formate. By default it is 0.

png is the flag. If flag is set, then the vector plot should be saved as png formate. By default it is 1.

Written By : Arulalan.T

Date : 21.09.2011 Updated: 12.12.2011

generate_iso_plots.**genIsoLinePlots**(*var*, *key*, *xmlpath=None*, *path=None*, *outpath=None*, *month=None*, *date=None*, *level='all'*, *region=None*, *svg=0*, *png=1*)

genIsoLinePlots() (page 49): It should generate the isoLine plots in vcs background and save it as png(by default) inside the outpath, with isoLevels (find out by this method) and isoColors (default).

Inputs [var is the variable name.]

key to identify, it is which variable to make plot name. xmlpath is the xml file absolute path. path is the nc file absolute path. pass any one (path or xmlpath)

outpath is the absolute path, where the generated plots should be stored. By default, it is None. It means, it should save in the current working directory itself. level is either single level, or list of levels or 'all'. 'all' means, it takes all the availableLevels from the variables. level value must be int, float only. Not be string, other than 'all' keyword. region is the region variable which should cut particular region shape from the global data. By default it is None, i.e. takes global data region itself.

svg is the flag. If flag is set, then the vector plot should be saved as svg formate. By default it is 0.

png is the flag. If flag is set, then the vector plot should be saved as png formate. By default it is 1.

..note:: This function should find out the isoLevels to set in the isoline plot and its legend in vcs. IsoLevels is the range of min and max of all the levels data min and max.

Written By : Arulalan.T

Date : 21.09.2011 Updated: 12.12.2011

generate_iso_plots.**genSeasonFcstSysErrDirs**(*modelName*, *modelHour*, *availableMonths*, *plotLevel*)

genSeasonFcstSysErrDirs() (page 49): It should generate the directory hierarichy structure of season fcstsyserr in the plotsgraphspath. And calls the function genIsoFillLinePlots to make 'isofillline' plots and save it inside the appropirate directory, by reading the xml file of the appropirate process season fcstsyserr files path.

It should plot for all the vars in the 'isovars' which has set in the global 'vars.txt' file.

To plot isoFillLinePlot, this function should find out the isoLevels for all the variables of all the levels and all the hours.

isoLevels [It is a range of levels which is from the min (of data of] all the hours and levels), to the max (of data of all the hours and levels) to set the levels in the vcs plot and legend.

Inputs [modelName is the one of the directories name.] modelHour is the one of the directories name. availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file, and generate the 'IsoFillLine' plots to those levels only.

Written By : Arulalan.T

Date : 20.09.2011 Updated: 12.12.2011

generate_iso_plots.**genSeasonMeanDirs**(*modelName*, *availableMonths*, *plotLevel*)

`genSeasonMeanDirs()` **(page 49): It should generate the directory hierarichy** structure of season mean in the plotsgraphspath. And calls the function genIsoLinePlots to make 'isoline' plots and save it inside the appropirate directory, by reading the xml file of the appropirate process season mean files path.

Inputs [modelName is the one of the directories name.] availableMonths is the dictionary which is generated by fully available months from the timeAxis.

..note:: It should takes the levels which is set in the global config file and generate the 'isoline' plots to those levels only.

Written By : Arulalan.T

Date : 20.09.2011 Updated: 12.12.2011

4.3.3 Statistical Score Bar Plots

The script *generate_statistical_score_bars.py* should generate the bar plots w.r.t the statistical scores for different region & fcst hours. Finally save the generated bar plots as either png or jpg or svg in the suitable directory for season wise. Date : 04.08.2011

Updated on : 28.09.2011

generate_statistical_score_bars.**genBarDiagrams**(*var*, *path*, *hours*, *outpath=None*, *bargap=0.28*, *barwidth=0.8*, *yticdiff=0.25*)

`genBarDiagrams()`: It should generate the least directory hierarichy structure of season statiscore in the plotsgraphspath by score name. It will plots score values in xmgrace as bar diagram and save it inside the appropirate directory, by reading the nc file of the appropirate process season Region statiscore files path.

It should plot for all the vars of that statiscore nc files.

Inputs [var is the variable name. If var is 'all' means, then it should] plot the bar diagram for all the available variables in the passed path nc or xml file.

path is an absolute nc or xml file path.

outpath is the path to store the images. If it is None means, it should create the least (plotname)directory in the current directory path itself and save it.

bargap is the value of the gap ratio in between each bars of each threshold in xaxis of score bar diagram.

barwidth is the width of the each bar in xaxis of the score bar diagram.

yticdiff is the difference of the tic levels in y axis of the bar diagram.

Written By : Dileep Kumar.R, Arulalan.T

Updated on : 28.09.2011

generate_statistical_score_bars.**genSeasonStatiScoreDirs**(*modelName*, *modelHour*, *availableMonths*)

`genSeasonStatiScoreDirs()`: It should generate the directory hierarichy structure of season statiscore in the plotsgraphspath. And calls the function genIsoFillPlots to make 'isofill' plots and save it inside the appropirate directory, by reading the nc file of the appropirate process season hour statiscore files path.

It should plot for all the vars of that statiscore spatial distributed nc files.

Inputs [modelName is the one of the directories name.] modelHour is the one of the directories name. availableMonths is the dictionary which is generated by fully available months from the timeAxis.

Written By : Arulalan.T

Date : 27.09.2011

4.3.4 Statistical Score Spatial Distribution Plots

The script *generate_stati_score_spatial_distribution_plots.py* should generate the iso fill plots w.r.t the statistical scores for different fcst hours. Finally save the generated iso fill plots as either png or jpg or svg in the suitable directory for season wise.

generate_stati_score_spatial_distribution_plots.**genIsoFillPlots**(*var*, *path*, *outpath=None*, *region=None*, *svg=0*, *png=1*)

genIsoFillPlots() (page 51): It should generate the directory least hierarichy structure of season statiscore in the plotsgraphspath,by the plotname.

It should plot for all the vars of that statiscore spatial distributed nc files.

Inputs [var is the variable name. If var is 'all' means, then it should]

plot the isofill for all the available variables in the passed path nc or xml file.

path is an absolute nc or xml file path.

outpath is the path to store the images. If it is None means, it should create the least (plotname)directory in the current directory path itself and save it.

region to extract the region from the var data.

if svg is 1, then plot should be saved as svg. if png is 1, then plot should be saved as png.

..note:: isoLevels and isoColors are set inbuilt (some default) range of levels and colors with respect to the variable name of statistical scores.

Written By : Dileep Kumar.R, Arulalan.T

Date : 26.09.2011

generate_stati_score_spatial_distribution_plots.**genSeasonStatiScoreDirs**(*modelName*, *modelHour*, *availableMonths*)

genSeasonStatiScoreDirs() (page 51): It should generate the directory hierarichy structure of season statiscore in the plotsgraphspath. And calls the function genIsoFillPlots to make 'isofill' plots and save it inside the appropirate directory, by reading the nc file of the appropirate process season hour statiscore files path.

It should plot for all the vars of that statiscore spatial distributed nc files.

Inputs [modelName is the one of the directories name.] modelHour is the one of the directories name. availableMonths is the dictionary which is generated by fully available months from the timeAxis.

Written By : Arulalan.T

Date : 26.09.2011

4.4 Statistical Scores

4.4.1 Contigency Table & Related Statistical Scores

The module *ctgfunction.py* should helps to calculate the contigency table and its related statistical scores.

For eg : Threat Score, Bias Score, Proabability of detection and more.

`ctgfunction.`**`accuracy`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`accuracy()` **(page 52):Hit Rate ,the most direct and intuitive measure of the**

accuracy of categorical forecasts is hit rate. The average correspond- ence between individual nforecasts and the events they predict. Scalar measures of accuracy are meant to summarize,in a single number, the overall quality of a set of forecasts. Can be mislead, since it is heavily influenced by the most common category, usually "no event" in the case of rare weather.

Accuracy= (a+d)/(a+b+c+d); 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs: Range: 0 to 1

Perfect Score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No:236-240)

Links : http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`bias_score`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`bias_score()` **(page 52): Bias score(frequency bias)-Measures the correspondence** between the average forecast and the average observed value of the predictand. This is different from accuracy, which measures the average correspondence between individual pairs of forecasts and observations.

Bias= (a+b)/(a+c); 'a' -hits, 'b'-false alarm, & 'c'-misses

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs: Range: 0 to infinity

Perfect score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 241)

Links : http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`contingency_table_2x2`**(*obs*, *fcst*, *th*)

`contingency_table_2x2()` (page 53):Creates the 2x2 contigency table useful for forecast verification. From 2x2 condigency table we can find thse statistical scores such as Hit Rate(HR), Bias(BS), Threat Score(TS), Odds Ratio(ODR)...etc

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

Outputs: A list of values [a, b, c, d]

where a = No of values such that both observed and predicted > threshold

b = No of values such that observed is < threshold and predicted > threshold

c = No of values such that observed is > threshold and predicted < threshold

d = No of values such that both observed and predicted < threshold

Usage:

example: From the contingency table the following statistics can be calculated. >>> HR = (a+d)/(a+b+c+d) >>> ETS = a/(a+b+c) >>> BS = (a+b)/(a+c) >>> ODR = (a*d)/(b*c)

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS

Links: http://www.cawcr.gov.au/projects/verification/#Atger_2001

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`eds`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

:func:'eds':Extreme dependency score, converges to 2n-1 as event frequency approaches 0, where n is a parameter describing how fast the hit rate converges to zero for rarer events. EDS is independent of bias, so should be presented together with the frequency bias.

EDS={2Log[(a+c)/(a+b+c+d)]/Log(a/(a+b+c+d))}-1; 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: -1 to 1, 0 indicate no skill.

Perfect Score: 1

Reference: [1]Stephenson D.B., B. Casati, C.A.T. Ferro and C.A. Wilson, 2008: The extreme dependency score: a non-vanishing measure for forecasts of rare events.

Meteorol. Appl., 15, 41-50.

Link: http://www.cawcr.gov.au/projects/verification.

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`ets`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`ets()` **(page 54):Equitable threat score (Gilbert skill score), the number of** forecasts of the event correct by chance, 'a_random',is determined by assuming that the forecasts are totally independent of the observations, and forecast will match the observation only by chance. This is one form of an unskilled forecast, which can be generated by just guessing what will happen.

ETS=(a-a_random)/(a+c+b-a_random) a_random=[(a+c)(a+b)]/(a+b+c+d); 'a' -hits, 'b'-false alarm,

'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: -1/3 to 1, 0 indicate no skill.

Perfect Score: 1

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`far`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`far()` **(page 54):False alarm ratio(FAR)-Proportion of forecast events that fail** to materialize.

FAR= b/(a+b); 'a' -hits, & 'b'-false alarm

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: 0 to 1

Perfect Score: 0

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 240-241)

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

ctgfunction.**hss**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

hss() (page 55):Heidke skill score (Cohen's k), the reference accuracy measure in the Heidke score is the hit rate that would be achieved by random forecasts, subject to the constraint that the marginal distributions of forecasts and observations characterizing the contingency table for the random forecasts, P(Yi) and p(oj), are the same as the marginal distributions in the actual verification data set.

HSS= 2.(a d - bc)/[(a + c)(c + d) + (a + b)(b + d)];

'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: -infinity to 1, 0 indicate no skill.

Perfect Score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 248-249)

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

ctgfunction.**kss**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

kss() (page 55):Hanssen and Kuipers discriminant(Kuipers Skill Score), the contribution made to the Kuipers score by a correct "no" or "yes" forecast increases as the event is more or less likely, respectively.A drawback of this score is that it tends to converge to the POD for rare events, because the value of "d" becomes very large.

HK= (ad-bc)/[(a + c)(b + d)]; 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range:-1 to 1 Perfect Score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 249-250)

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`logodr`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`logodr()` (page 56): Log Odds ratio, LOR is the logaritham of odds ratio. When the sample is small/moderate it is better to use Log Odds Ratio. It is a good tool for finding associations between variables.

LOR=Log(Odds Ratio); Odds Ratio=ad/bc; 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: -infinity to infinity, 0 indicate no skill.

Perfect Score: infinity

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.`**`odr`**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

`odr()` (page 56):Odds ratio, the odds ratio is the ratio of the odds of an event occurring in one group to the odds of it occurring in another group.cThe term is also used to refer to sample-based estimates of this ratio.Do not use if any of the cells in the contingency table are equal to 0. The logarithm of the odds ratio is often used instead of the original value.Used widely in medicine but not yet in meteorology.

OD= ad/bc; 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: 0 to infinity, 1 indicate no skill

Perfect Score: infinity

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

ctgfunction.**orss**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

orss() **(page 57): Odds ratio skill score (Yule's Q), this score was proposed** long ago as a 'measure of association' by the statistician G. U. Yule (Yule 1900) and is referred to as Yule's Q. It is based entirely on the joint conditional probabilities, and so is not influenced in any way by the marginal totals.

ORSS= (ad-bc)/(ad+bc); 'a' -hits, 'b'-false alarm, 'c'-misses, & 'd'- correct negatives

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: -1 to 1, 0 indicates no skill

Perfect Score: 1

Reference: Stephenson, D.B., 2000: Use of the "odds ratio" for diagnosing forecast skill. Wea. Forecasting, 15, 221-232.

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

ctgfunction.**pod**(*obs=None*, *fcst=None*, *th=None*, ***ctg*)

pod() **(page 57):Probability of detection(POD), simply the fraction of those** occasions when the forecast event occurred on which it was also forecast.

POD= a/(a+c); 'a' -hits, & 'c'-misses

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs: Range: 0 to 1

Perfect Score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 240)

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

`ctgfunction.pofd(obs=None, fcst=None, th=None, **ctg)`

podf():Probability of false detection (false alarm rate), measures the fraction of false alarms given the event did not occur.

POFD=b/(d+b); 'b'-false alarm & 'd'- correct negatives

Inputs: obs - the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs:

Range: 0 to 1

Perfect Score: 0

Links: http://www.cawcr.gov.au/projects/verification/

`ctgfunction.ts(obs=None, fcst=None, th=None, **ctg)`

:func:'ts': Threat Score (Critical Success Index), a frequently used alternative to the hit rate, particularly when the event to be forecast (as the "yes" event) occurs substantially less frequently than the non-occurrence ("no").

TS=a/(a+b+c); 'a' -hits, 'b'-false alarm, &'c'-misses

Inputs: obs- the observed values has to be a numpy array(or whatever

you decide)

fcst - the forecast values th - the threshold value for which the contingency table needs

to be created (floating point value please!!)

By default obs, fcst, th are None. Instead of passing obs, fcst, and th values, you can pass 'ctg_table' kwarg as 2x2 matrix value.

Outputs: Range: 0 to 1

Perfect Score: 1

Reference: "Statistical Methods in the Atmospheric Sciences", Daniel S Wilks, ACADEMIC PRESS(Page No: 240)

Links: http://www.cawcr.gov.au/projects/verification/

Written by: Dileepkumar R, JRF, IIT Delhi

Date: 24/02/2011

4.5 More

More utilities will be added and optimized in near future.

CHAPTER

FIVE

DOCUMENTATION OF MJO SOURCE CODE

5.1 MJO-LEVEL1 Diagnosis Utils

5.1.1 eof

eof_diag.**genEofVars**(*infiles*, *outfile*, *eobjf=True*, *latitude=(-30*, *30*, *'cob')*, *NEOF=4*, *season='all'*, *year=None*, ***kwarg*)

5.1.2 Power Spectrum Utils

psutils.**areaAvg**(*varName*, *fpath*, ***kwarg*)

Returns the area averaged data by accessing the var data from the fpath itself by extracting needed portion of data only by the following key word arguments.

KWargs : (latitude and/or longitude) or (region) and/or level

Written by : Arulalan.T

Date : 08.01.2013

psutils.**powerSpectrum**(*varName*, *fpath*, *sday*, *smon*, *eday*, *emon*, *hr=0*, *nodays=None*, ***kwarg*)

varName - variable fpath or data :

This anomaly data should not be spatial one. It should be spatially averaged, daily data. It may contains many years data with proper time axis.

nodays [No of days. though its averaged data, pass the no of days of the] data of each year which has averaged. For 180 days summer/winter season averaged of multi-year data, you have to pass nodays=180.

Written By : Arulalan.T

Date : 31.10.2012

psutils.**waveNumber**(*varName*, *fpath*, *sday*, *smon*, *eday*, *emon*, *hr=0*, ***kwargs*)

varName - variable fpath or data :

This anomaly data should be meridionally averaged one. It should be daily data. It may contains many years data with proper time axis.

KWArgs :- nodays : No of days. though its averaged data, pass the no of days of the

data of each year which has averaged. For 180 days summer/winter season averaged of multi-year data, you have to pass nodays=180.

window : do cosine window if True

iglimit [ignore newlon_limit days. By default it takes 10 days.] i.e. If extracted season data is less than the (actual seasonal days - iglimit) it will throw an error.

Date : 28.11.2012, 02.12.2012

psutils.**zonalAvg**(*varName*, *fpath*, ***kwarg*)

Returns the zonal averaged data by accessing the var data from the fpath itself by extracting needed portion of data only by the following key word arguments.

KWargs : (latitude and/or longitude) or (region) and/or level

Returns : It vanishes the latitude in the input data and returns it.

Written by : Arulalan.T

Date : 08.01.2013

5.1.3 Variance Utils

variance_utils.**anomaly**(*data*, *climatology*, *climyear=1*, ***kwarg*)

Calculating anomaly. Anomaly = model data - climatology

In this function, it should find out either data has leap year day data or not. Also find out either climatology data has leap year day data or not.

Depends upon these case, it should remove the leap day data from either model data or climatology data, when shapes are mis-match (i.e. both leap and non leap year data has passed as arguments) to compute anomaly.

KWargs : When data and climatology shapes are mis-match, then cregrid : if True, then climatology data will be regridded w.r.t

model/obs/data and then anomaly will be calculated.

dregrid [if True, then model/obs/data will be regridded w.r.t] climatology data and then anomaly will be calculated.

..note:: We can not enable both cregrid and dregrid at the same time.

Written By : Arulalan.T

Date : 11.06.2012 Updated : 24.06.2013

variance_utils.**calculateSeasonalVariance**(*varName*, *fpath*, *sday*, *smon*, *eday*, *emon*, *hr=0*, ***kwarg*)

calculateSeasonalVariance [calculate Variance for anomaly seasonal data] in temporal way. It should extract the data only for the seasonal days of the year. Even it has more years, then it should extract only the seasonal days data of all the years, and then calculate the variance for the seasonal days of the all the years of the data.

Inputs: varName : anomaly variable name fpath : anomaly (nc) file path sday : starting day of the season [of all the years of the data]. smon : starting month of the season [of all the years of the data]. eday : ending day of the season [of all the years of the data]. emon : ending month of the season [of all the years of the data]. hr : hour for both start and end date

KWargs:

year [Its optional only. By default it is None. i.e. It will] extract all the available years seasonalData. If one interger year has passed means, then it will do extract of that particular year seasonData alone. If two years has passed in tuple, then it will extract the range of years seasonData from year[0] to year[1]. eg1 : year=2005 it will extract seasonData of 2005 alone. eg2 : year=(1971, 2013) it will extract seasonData from

1971 to 2013 years.

..note:: If end day and end month is lower than the start day and start month, then we need to extract the both current and next year data. For eg : Winter Season (November to April). It can not be reversed for this winter season. We need to extract data from current year november month upto next year march month.

If you pass one year data and passed the above

winter season, then it will be extracted november and december months data and will be calculated variance for that alone.

If you will pass two year data then it will extract the data

from november & december of first year and january, feburary & march of next year will be extracted and calculated variance for that.

Written By : Arulalan.T

Date : 26.07.2012

variance_utils.**calculateVariance**(*varName*, *fpath*, *speed=True*, ***kwarg*)

calculate Variance for anomaly data in temporal way.All Seasonal Variance.

if speed is True, then it should calculate the variance for the whole data.

If data size is too large which cant handle by normal system, then we should off the speed arg. speed = False. So it will take the partial data (in latitude and/or level wise) with its full time axis. By this way, we can calculate the variance for large size of data.

It should loop through the each and every latitude grid point, (but not by longitude) get the full time series with full longitude wise data and compute the statistics.variance and store it.

KWargs:

year [Its optional only. By default it is None. i.e. It will] extract all the available years data to compute variance over timeAxis. If one interger year has passed means, then it will do extract of that particular year data alone. If two years has passed in tuple, then it will extract the range of years data from year[0] to year[1]. eg1 : year=2005 it will extract seasonData of 2005 alone. eg2 : year=(1971, 2013) it will extract seasonData from

1971 to 2013 years.

Written By : Arulalan.T

Date : 26.07.2012

variance_utils.**computeAnomaly**(*mvar*, *modelPath*, *cvar*, *climPath*, *climyear*, *outPath*, *convertTI2N=False*, *hour=None*, *sign=1*, ***kwarg*)

computeAnomaly [compute the anomaly by opening the model and climatology] data from the files. Finally writes the output anomaly into outpath file.

Input : mvar - model variable name modelPath - model nc file absolute path cvar - climatology variable name climPath - climatology nc file absolute path climyear - Year in climatology file outPath - anomaly nc file (output) absolute path convertTI2N - It takes either True or False. If it is True, then the

model data will be converted from Time Integrated to Normal form. i.e. Units will be converted from Wsm^-2 to Wm^2.

hour - hour of the model data (will be used in convertTI2N function) sign - change sign of the model data (will be used in convertTI2N fn)

kWargs :- long_name - long name for the output anomaly comments - comments for the output anomaly cregrid - climatology data will be regridded w.r.t model data dregrid - model data will be regridded w.r.t climatology data

by default both cregrid & dregrid are False

Refer : anomaly, convertTimeIntegratedToNormal

Written By : Arulalan.T

Date : 11.06.2012

variance_utils.**lfilter**(*data*, *weights*, *cyclic=True*, ***kwarg*)

Lanczos Filtered Todo : Write doc of the work flow KWarg:

returntlen [return time axis length. By default it takes None.]

It can take positive or negative integer as value. If it is +ve no, then it will do filter only to the first no days rather than doing filter to the whole available days of the data. If it is -ve no, then it will do filter only to the last no days. Timeaxis also will set properly w.r.t return data. if cyclic is enabled or disabled, then returntlen index will change according to cyclic flag.

Lets assume weights length is 101. Eg1: Lets consider data length is 365.

If cyclic is False, then filteredData length is 365-(100*2)=165 when returntlen is None. If returntlen = 10, so filteredData length is 10 and its index is range(100, 110). If returntlen = -10, so filteredData length is 10 and its index is range(255, 265).

Eg2: If cyclic is True, then filteredData length is 365 when returntlen is None. If returntlen = 10, so filteredData length is 10 and its index is range(0, 10). If returntlen = -10, so filteredData length is 10 and its index is range(355, 365).

So returntlen option is useful when we need to filter only to the certain days. But regardless of returntlen option, we need to pass the full data to apply Lanczos Filter to either returntlen days or full days. This option will save a lot of time when returntlen is very less compare to the total length of the data.

Written By : Arulalan.T

Date : 15.09.2012 Updated : 22.09.2013

variance_utils.**plotVariance**(*data*, *outfile*, *season*, *title=''*, *pdf=1*, *png=0*, ***kwarg*)

Written By : Alok Singh, Arulalan.T

Date : 28.05.2012 Updated : 08.07.2013

variance_utils.**summerVariance**(*varName*, *fpath*, *sday=1*, *smon=5*, *eday=31*, *emon=10*, *hr=0*, ***kwarg*)

summerVariance : summer season (May to October) variance

Inputs : varName - anomaly variable name fpath - anomaly file path sday : starting day of the summer season [of all the years of the data]. smon : starting month of the summer season [of all the years of the data]. eday : ending day of the summer season [of all the years of the data]. emon : ending month of the summer season [of all the years of the data]. hr : hour for both start and end date

KWargs :

year [It could be single year or range of years in tuple.] By default it is None. So it will calculate all available years summerVariance

Refer: calculateSeasonalVariance

Written By : Arulalan.T

Date : 26.07.2012

variance_utils.**winterVariance**(*varName*, *fpath*, *sday=1*, *smon=11*, *eday=30*, *emon=4*, *hr=0*, ***kwarg*)

winterVariance : winter season (November to April) variance

Inputs : varName - anomaly variable name fpath - anomaly file path sday : starting day of the winter season [of all the years of the data]. smon : starting month of the winter season [of all the years of the data]. eday : ending day of the winter season [of all the years of the data]. emon : ending month of the winter season [of all the years of the data]. hr : hour for both start and end date

KWargs :

year [It could be single year or range of years in tuple.] By default it is None. So it will calculate all available years winterVariance

Refer: calculateSeasonalVariance

Written By : Arulalan.T

Date : 26.07.2012

5.2 MJO-LEVEL2 Diagnosis Utils

5.2.1 combined eof utils

ceof_diag.**genCeofVars**(*infiles*, *outfile*, *eobjf=True*, *lat=(-15*, *15*, *'cob')*, *NEOF=4*, *season='all'*, ***kwarg*)

genCeofVars : generate the combined eof variables as listed below.

Input :-

infiles [List of tuples contains variable generic name, actual varible] name (say model varName), data path. So ceof input files list contains the tuples which contains the above three information.

For eg : ceof(olr, u200, u850) infiles = [('olr', 'olrv', 'olr.ctl'),

('u200', 'u200v', 'u200.ctl'), ('u850', 'u850v', 'u850.ctl')]

outfile [output nc file path. All the supported output varibles of ceof] will be stored in this nc file. This nc file will be opened in append mode.

eobjf [eof obj file store. If True, the eofobj of ceof will be stored] as binary ('*.pkl') file in the outfile path directory, using pickle module.

lat [latitude to extract the data of the input file.] By default it takes as (-15, 15, 'cob').

NEOF : no of eof or no of mode. By default it takes 4.

season [It could be 'all', 'sum', 'win'. Not a list.] all : through out the year data from all the available years sum : only summer data will be extracted from all the available years win : only winter data will be extracted from all the available years

Output [The follwing output varibles from 1 to 5 will be stored/append]

into outfile along with individual varName, season (where ever needed).

1. Std of each zonal normalized varibles

2. Percentage explained by ceof input variables
3. variance accounted for ceof of each input variables
4. eof variable of each input variables
5. PC time series, Normalized PC time Series (for 'all' season only)
6. Store the eofobj (into .pkl file) [optional]

Written By : Dileep.K, Arulalan.T

Date :

Updated : 11.05.2013

ceof_diag.**genProjectedPcts**(*infiles*, *outfile*, *eofobj*, *lat=(-15*, *15*, *'cob')*, *NEOF=4*, *season='mjjas'*, ***kwarg*)

KWargs:

ogrid [if grid has passed then data will be regridded before] normalize it w.r.t passed grid resoltution. By default it takes None.

dtype [dtype should be either 'anl', or fcst hours.] if both season and dtype has passed then, according to dtype season time will be calculated using xmlobj.findPartners() method. By default it takes 'Analysis'. i.e. period from 01-05-yyyy to 30-9-yyyy for the 'mjjas' season. If dtype will be '24' means then period will be from 30-04-yyyy to 29-09-yyyy. Likewise user can pass fcst hour.

Date : 08.08.2013

ceof_diag.**getZonalNormStd**(*data*)

data - pass filtered data Return : normalized data and std of the meridionally averaged data.

Once we averaged over latitude, then it will become zonal data.

ceof_diag.**makeGenCeofVars**(*rawOrAnomaly='Anomaly'*, *filteredOrNot='Filtered'*, *seasons=['all', 'sum', 'win'], **kwarg*)

Written By : Arulalan.T

Date : 22.07.2013

ceof_diag.**makeProjectedPcts**(*rawOrAnomaly='Anomaly'*, *filteredOrNot='Filtered'*, *seasons=['mjjas'], obsname='MJO', **kwarg*)

KWarg:

exclude [exclude hours. If some hours list has passed then those] model hours will be omitted. eg : 01 hour. Note : it will omit those exclude model hours directory. So for the remaining model anl, fcst hours only calculated the projected pcts.

Written By : Arulalan.T

Date : 08.08.2013

5.2.2 phase3d utils

trig.**getHalfQuadrantOfCircle**(*x*, *y*)

getHalfQuadrantOfCircle [return the half quadrant of the circle for the] given x, y points. Usually we split the circle into 4 equal quadrants, but in this function we split circle into 8 equal parts. Thats why we call it as half quadrants. i.e we split one quadrant into two equal parts (two equal half quadrants).

Written By : Arulalan.T

Date : 02.04.2013

License : GPL V3

phase3d.**miso_phase3d**(*xdata, ydata, sxyphase=None, dmin=None, dmax=None, colors=['violet', 'green', 'orange', 'red'], tcomment='Centre India & North India', bcomment='Southern tp & IO', lcomment='Peninsular & Centre India', rcomment='FootHill & SIO', title='Indian Monsoon Intraseasonal Oscillation Index', stitle1='', stitle2='', ctitle='Weak MISO', pposition1=8, pdirection='clock', plocation='out', mintick=4, **kwarg*)

miso_phase3d [Monsoon Intraseasonal Oscillation Phase 2 Dimensional] Diagram of 3D Var (pcs1, pcs2 & time)

Inputs : xdata : Normalized PC1 or relevant data ydata : Normalized PC2 or relevant data sxyphase : phase number of start x,y points of npc1, npc2. By default

it takes None. For the simplicity to the user, passed the default/correct argument to the pposition1 as 8.

pposition1 [phase number of the phase position1 of the graph. For the] MJO it must be 8. User can over write the sxyphase and pposition1 args while calling this function, if needed.

pdirection : By default it takes 'clock' to this MJO. plocation : Phase name/string drawn inside/outside of the graph. By

default it takes 'in'.

colors [list contains 6 colors to indicate 6 months dataset. User] can overwrite it either by single or many color.

Return : It should return the 'x' of the xmgrace object which has plotted the data with this MJO args. User can either save it into ps, pdf, etc., or modify/update further.

Refer : phase3d() for the detailed documents of all the arguments.

Plot Properties Reference ["An Indian monsoon intraseasonal oscillations] (MISO) index for real time monitoring and forecast verification", E. Suhas , J. M. Neena , B. N. Goswami, Clim Dyn, DOI 10.1007/s00382-012-1462-5

Author : Arulalan.T

Date : 20.02.2013

License : GPL V3

phase3d.**mjo_phase3d**(*npc1, npc2, sxyphase=None, dmin=0, dmax=0, colors=['magenta', 'blue', 'violet', 'green', 'orange', 'red'], tcomment='Western Pacific', bcomment='Indian Ocean', lcomment='Western Hemisphere & Africa', rcomment='Maritime Continent', title='MJO Phase diagram of PC-1 and PC-2', stitle1='', stitle2='', ctitle='Weak MJO', pposition1=5, pdirection='anticlock', plocation='in', mintick=0, **kwarg*)

mjo_phase3d [Madden-Julian Oscillation Phase 2 Dimensional Diagram of] 3D Var (pcs1, pcs2 & time)

Inputs : npc1 : Normalized PC1 (xdata) npc2 : Normalized PC2 (ydata) sxyphase : phase number of start x,y points of npc1, npc2. By default

it takes None. For the simplicity to the user, passed the default/correct argument to the pposition1 as 5.

pposition1 [phase number of the phase position1 of the graph. For the] MJO it must be 5. User can over write the sxyphase and pposition1 args while calling this function, if needed.

pdirection : By default it takes 'anticlock' to this MJO. plocation : Phase name/string drawn inside/outside of the graph. By

default it takes 'in'.

colors [list contains 6 colors to indicate 6 months dataset. User] can overwrite it either by single or many color.

Return : It should return the 'x' of the xmgrace object which has plotted the data with this MJO args. User can either save it into ps, pdf, etc., or modify/update further.

Refer : phase3d() for the detailed documents of all the arguments.

Plot Properties Reference : http://climate.snu.ac.kr/mjo_diagnostics/index.htm

Author : Arulalan.T

Date : 20.02.2013

License : GPL V3

phase3d.**phase3d**(*xdata, ydata, sxyphase, dmin=None, dmax=None, colors=['red'], tcomment='', bcomment='', lcomment='', rcomment='', title='Phase Diagram', stitle1='', stitle2='', ctitle='Weak', **kwarg*)

phase3d [This will allow user to plot the 2 dimensional line plot in] circular path among 8 phases of 3 dimensional datasets. i.e. pcs1, pcs2 and time are 3 dimensional datasets. It as projected into 2 dimensional line plot. But it actually reprsent 3 dimensional dataset. So we named it as phase3d. Also named as phase space diagram.

Inputs:

xdata [single dimensional xaxis dataset.] eg : normalized pc1 time series

ydata [single dimensional yaxis dataset.] eg : normalized pc2 time series Both xdata & ydata should be continous time series dataset.

xdata.id and ydata.id will be set as xaxis & yaxis label.

dmin [data min - xaxis and yaxis minimum scale/label value.] It must be -ve.

dmax [data max - xaxis and yaxis maximum scale/label value.] It must be +ve. By default both dmin, dmax takes as None. If it is None, then this function will automatically find out the min & max from the xdata & ydata. Finally set dmin = -dmax. So that we will get squared region.

colors [phase line colors. It should be eithre string or list of] colors. If it is string or single color list, the it will apply that color through out the data line in the graph. If user passed it as list of colors (more than one colors) then it will apply those colors to the available months data set. So make sure that your colorlist and available months length should be same.

tcomment : Top Comment bcomment : Bottom Comment lcomment : Left Comment rcomment : Right Comment title : Title of the diagram stitle1 : Sub Title 1 -

If 'year' has passed then it will draw string as start year and end year from the xdata.

stitle2 [Sub Title 2 -] If 'month' has passed then it will draw string as season of available months (from startmonth to endmonth).

If 'date' has passed then it will draw string as season of available dates (from startdate to enddate).

User can overwrite the stitle1 & stitle2 strings.

ctitle : Centred Title or text inside the circle.

sxyphase [start x, y phase - start x,y points of xdata, ydata phase] number. So user can pass the phase number of the starting points of the whole data (xdata, ydata). With depends upon pdirection it will be draw the phase name over the 8 half quadrants of the graph. User no need to pass phase numbers for the whole datasets (xdata, ydata). Just pass the first or start day of season phase alone. User can use pposition1 argument also instead of sxyphase argument. No default argument takes place here. So user must pass argument to this sxyphase arg. If user need to use pposition1 arg, then they must pass None to this sxyphase argument.

KWargs :-

pposition1 [phase position 1 - Lets understand the distribution of] 8 phases. Lets consider from 0 to 45 degree region as phase position 1 (pposition1).From 45 to 90 degree region as phase position 2(pposition2) and so on. So if we walk in anti-clock wise direction we will endup with pposition8 in the region of 315 to 360 degree.

User can pass argument to pposition1 as integer from

1 to 8. So whatever user passed in the pposition1 phase number, that phase no string will be draw within in range from 0 to 45 degree.

eg [pposition1=5 . i.e. phase name 'PHASE 5' will be] drawn in the 0 to 45 degree region of the graph.

By default it takes None argument.

..note:: User can not use both sxyphase and pposition1 args. Can be used either sxyphase or pposition1 only.

pdirection [phase direction - It can be either 'clock' or 'anticlock']

It will determine what is the phase number in the pposition2, pposition3, ..., pposition8 w.r.t the input of pposition1 argument or sxyphase and clock/anticlock direction. eg 1: pposition1 = 5, pdirection = 'anticlock'

pposition1=5, pposition2=6, pposition3=7, pposition4=8, pposition5=1, pposition6=2, pposition7=3, pposition8=4.

i.e. w.r.t pposition1 as 5 and pdirection as 'anticlock'

the phase names of the rest phase positions are

determined as increment trend of pnames in ppositions.

eg 2: pposition1 = 5, pdirection = 'clock'

pposition1=5, pposition2=4, pposition3=3, pposition4=2, pposition5=1, pposition6=8, pposition7=7, pposition8=6.

i.e. w.r.t pposition1 as 5 and pdirection as 'clock'

the phase names of the rest phase positions are

determined as decrement trend of pnames in ppositions.

same examples we can play with sxyphase also.

plocation [It takes input as 'in/inside/out/outside'. If it is] 'in' or 'inside', then the phase name/number will be drawn inside the squared graph itself. If it is 'out' or 'outside', then the phase name/number will be drawn outside the squared graph.

mintick [minor ticks count. By default it takes as 0. user can pass 4] also.

timeorder [It takes a list of nos which representing the month order] from 1 to 12. It could be any order. If user has passed any order of nos in this, then line plots will be plotted in the passed months order. By default it takes None. In that case the data months wil be sorted in the correct order, then that will be plotted. eg : timeorder=[11, 12, 1, 2, 3, 4] . This will be useful to plot winter season when we have only one year data. So user has to enable both cyclic=True and timeorder as the above list. Then it will be plotted as NDJFMA. Otherwise it will plot as JFMAND & make looks like mess-up. For cyclic option refer timeutils.getSeasonData().

Methods Used : _sortMonths, getTimeAxisMonths, getHalfQuadrantOfCircle

Return : It should return the 'x' of the xmgrace object which has plotted the data with this MJO args. User can either save it into ps, pdf, etc., or modify/update further.

Author : Arulalan.T

Date : 20.02.2013

License : GPL V3

5.2.3 Wheeler Kiladis Diagram Utils

wk_utils.**genWKVars**(*data*, *outpath*, *outfile*, *segment=96*, *overlap=60*, ***kwarg*)

data - model anomaly

output - It produces the following 6 variables & will be written

into outfile.

1. power
2. power_S # symmetric power
3. power_A # anti-symmetric power
4. background
5. power_S_bg # symmetric power / background
6. power_A_bg # anti-symmetric power / background

outfile - user has to pass the outfile name with .nc file extension. This outfile name will be updated along with no of days in segment and no of days overlap. Finally it will return the new nc file name with its outpath.

Return - outfile path (outfile name has modified here)

Written By : Arulalan.T

Updated : 24.06.2013

wk_utils.**plotWKBackground**(*infile*, *outpath*, *outfile='background'*, *lmin=-1*, *lmax=2*, *png=0*, *pdf=1*)

figure2 : plotting background

wk_utils.**plotWKPowers**(*infile*, *outpath*, *outfile='Powers'*, *png=0*, *pdf=1*)

figure1 : plotting power_S & power_A

wk_utils.**plotWK_Sym_ASym**(*infile*, *outpath*, *outfile='Sym_ASym'*, *lmin=1.0*, *lmax=2.1*, *ptitle='Wheeler Kiladis Diagram'*, *comment_1=''*, *comment_2='Variable (Data) :*, *Period : JJAS'*, *png=0*, *pdf=1*)

figure3 : plotting power_S_bg & power_A_bg

lmin : legend min user can pass even +ve value or -ve value lmax : legend max must be +ve value

ptitile : Plot title

comment_1 [If comment_1 is '' or None, then comment will be extracted] from power_S_bg variable (while generating this variable using funciton genWKVars, the comment has set to all the variables as like '96-day segment, 60-day overlapping'. The parameters days will be set w.r.t input of that funciton).

comment_2 [By default it is 'Variable (Data)][, Period][JJAS'.] User can change it as follows eg : 'Variable (Data) : OLR (NCMRWF), Period : JJAS 2010'

Both comment_1 & comment_2 can be changed by user.

CHAPTER

SIX

DOCUMENTATION OF MISO SOURCE CODE

6.1 MISO Diagnosis Utils

6.1.1 harmonic climatology utils

harmonic_util.**harmonic**(*data*, *k=3*, *time_type='daily'*, *phase_shift=15*)

Inputs [] data : climatology data k : Integer no to compute K th harmonic. By default it takes 3. time_type : daily | monthly | full (time type of input climatology)

'daily' -> it returns 365 days harmonic, 'monthly' -> it returns 12 month harmonic, 'full' -> it retuns harmonic for full length of input data.

phase_shift [Used to subtract 'phase_shift' days lag to adjust] phase_angle w.r.t daily or monthly. By default it takes 15 days lag to adjust phase_angle w.r.t daily data. User can pass None disable this option.

Returns : Returns "sum mean of mean and first K th harmonic" of input climatology data.

Concept :

Earth science data consists of a strong seasonality component as indicated by the cycles of repeated patterns in climate variables such as air pressure, temperature and precipitation. The seasonality forms the strongest signals in this data and in order to find other patterns, the seasonality is removed by subtracting the monthly mean values of the raw data for each month. However since the raw data like air temperature, pressure, etc. are constantly being generated with the help of satellite observations, the climate scientists usually use a moving reference base interval of some years of raw data to calculate the mean in order to generate the anomaly time series and study the changes with respect to that.

Fourier series analysis decomposes a signal into an infinite series of harmonic components. Each of these components is comprised initially of a sine wave and a cosine wave of equal integer frequency. These two waves are then combined into a single cosine wave, which has characteristic amplitude (size of the wave) and phase angle (offset of the wave). Convergence has been established for bounded piecewise continuous functions on a closed interval, with special conditions at points of discontinuity. Its convergence has been established for other conditions as well, but these are not relevant to the analysis at hand.

Reference: Daniel S Wilks, 'Statistical Methods in the Atmospheric Sciences' second Edition, page no(372-378).

Written By : Arulalan.T

Date : 16.05.2014

CHAPTER

SEVEN

DOCUMENTATION OF OTHER UTILS SOURCE CODE

7.1 binary2ascii convertor

This function used to convert any kind of climate model binary files such as netcdf, grib1, grib2, pp, ctl, etc., to ascii / csv file.

binary2ascii.**binary2ascii**(*var*, *fpath*, *opath=None*, *dlat=None*, *dlon=None*, *freq='daily'*, *missing_value='default'*, *speedup='True'*)

binary2ascii [Convert the binary file such as nc, ctl, pp into ascii csv] files. It should create individual files for each years. Csv file contains the month, day, lat & lon information along with its corresponding data.

It has optimised code to extract data and write into file by using numpy.tofile() function. Its just extract the particular/each lat grid, extract all the longitude values in single dimension array and write into file object at a time. So it is more optimised.

Inputs : var - variable name

fpath - binary file input absolute path

opath - output directory path. Inside this folder, it should create csv files with variable name along with year. If user didnt pass any value for this, then it should create variable name as folder name for the output in the current working directory path.

dlat - need data lat shape in ascii. eg (0, 40) dlon - need data lon shape in ascii. eg (60, 100)

freq - it takes either 'daily' or 'monthly'. It is just to fastup the time dimension loop by skipping 365 days in daily and 12 months in monthly to access the another/ next year dataset.

missing_value - if missing_value passed by user, then that value should be set while writing into csv file. By default it takes 'default' value, i.e. it will take fill_value from the binary file information itself.

speedup - This binary2ascii.py works fine only for all 12 months or 365 days data. If some months are missing in b/w means, it will fail to work. So in that case, you switch off this speedup option.

todo - to get the available years, we need to use timeutils.py module. in that case, the above speedup option no need.

Written By : Arulalan.T

Date : 22.08.2012

7.2 numpy utils

This numutils (page 76) module helps us to generate our own time axis, correct existing time axis bounds and generate bounds.

Here we used inbuilt methods of cdtime and cdutil module of uv-cdat.

7.2.1 numutils

numutils.**nextmax**(*x*, *val=None*)

nextmax(): Returns the max value next to the top max value of the numpy x. If val doesnot passed by user, it returns the second most max value.

Inputs [x, numpy array] val, any value. If value passed, then it should return the max value of the next to the passed value.

Usage :

```
>>> x = numpy.array([[10, 1], [100, 1000]])
>>> x
array([[  10,    1],
       [ 100, 1000]])
>>> nextmax(x)
100
    we didnt pass any val, so it should return 2nd max value

>>> nextmax(x,99)
10
    we passed 99 as the val. So it should return the next max value
    to the 99 is 10.

>>> nextmax(x,1000)
100
    we passed 1000 as the val. So it should return the next max value
    to the 1000 is 100.

>>> nextmax(x,1)
```

we passed 1 as the val. i.e. the least number in the x (or least number which is not even present in the x). So there is no next max number to 1. It should return None.

we can find out 3rd most max value by just calling this function two times. >>> n = nextmax(x) >>> nextmax(x, n) >>> 10

10 is the 3rd most number in x.

Written By : Arulalan.T

Date : 27.09.2011

numutils.**nextmin**(*x*, *val=None*)

nextmin(): Returns the min value next to the least min value of the numpy x. If val doesnot passed by user, it returns the second lease min value.

Inputs [x, numpy array] val, any value. If value passed, then it should return the min value of the next to the passed value.

Usage :

```
>>> x = numpy.array([[10, 1], [100, 1000]])
>>> x
array([[  10,    1],
       [ 100, 1000]])
>>> nextmin(x)
10
    we didnt pass any val, so it should return 2nd min value

>>> nextmin(x, 11)
100
```

we passed 11 as the val. So it should return the next min value to the 11 is 100.

```
>>> nextmin(x, 101)
1000
```

we passed 101 as the val. So it should return the next min value to the 101 is 1000.

```
>>> nextmin(x, 1000)
```

we passed 1000 as the val. i.e. the most number in the x (or most number which is not even present in the x). So there is no next min number to 1000. It should return None.

we can find out 3rd least min value by just calling this function two times. >>> n = nextmin(x) >>> nextmin(x, n) >>> 100

100 is the 3rd least number in x.

Written By : Arulalan.T

Date : 27.09.2011

numutils.**permanent**(*data*)

permanent: Square Matrix permanent

It would be numpy data or list data.

Matrix permanent is just same as determinant of the matrix but change -ve sign into +ve sign through out its calculation of determinant.

eg 1:

```
>>> a = numpy.ones(9).reshape((3,3))
>>> z = permanent(a)
>>> print z
>>> 6.0
```

eg 2:

```
>>> a = numpy.ones(16).reshape((4,4))
>>> z = permanent(a)
>>> print z
>>> 24.0
```

Written By : Arulalan.T

Date : 01.08.2012

numutils.**remove_nxm**(*data*, *n*, *m*)

remove_nxm : Remove n-th row and m-th column from the matrix/numpy data. zero is the starting index for the row and column. To remove first row & column, we need to pass 0 as args.

eg:

```
>>> a = numpy.arange(20).reshape((4,5))
>>> print a
>>> [[ 0  1  2  3  4]
     [ 5  6  7  8  9]
     [10 11 12 13 14]
     [15 16 17 18 19]]
>>> b = remove_nxm(a, 2, 2)
>>> print b
>>> [[ 0  1  3  4]
     [ 5  6  8  9]
     [15 16 18 19]]
>>>
  ..note:: removed 2-nd row and 2-column from the matrix a.

>>> c = remove_nxm(a, 0, 4)
>>> print c
>>> [[ 5,  6,  7,  8],
     [10, 11, 12, 13],
     [15, 16, 17, 18]]
>>>
 ..note:: removed 0-th row and 4-th column from the matrix a.
```

Written By : Arulalan.T

Date : 01.08.2012

7.3 Non-rectilinear Utils

nonrect_utils.**get1LatLonFromNonRectiLinearGrid**(*grid*, *lat*, *lon*, *diff=0.5*)

func [get1LatLonFromNonRectiLinearGrid] It is locating the input lat & lon in the non-rectilinear grid data and returning its corresponding first dimension index (i) & second dimension index (j) (of the grid which is very close to the input lat & lon values).

Inputs :

grid [Its the cdms2 dataset grid value. Use x.getGrid() to pass] the dataset grid value, where x is cdms2 dataslab.

lat : latitude which you looking for. lon : longitude which you looking for. diff : By default it takes 0.5. It is the purpose of masking the

outer region other than (lat-diff, lat+diff) and (lon-diff, lon+diff).

Logic : Here we are getting the lat_vertices and lon_vertices data as well as lat_slab and lon_slab from the grid. i.e. Using grid.getBounds(), grid.getLatitude() & grid.getLongitude() functions.

Do the mask operation on the lat_vertices where ever outer than the (lat-diff, lat+diff).

Do the mask operation on the lon_vertices where ever outer than the (lon-diff, lon+diff).

Multiply the resultant masked boolean array of lat & lon gives us the near about 10 grids location which are all with in the range of (lat-diff, lat+diff) and (lon-diff, lon+diff) both matched together.

So from this 10 grids locations, using distance b/w two points in the curved line equation, we can identify the minimum distance from the input lat & lon.

Finally we can loate the minimum distance grid cell's first dimension index (i) and its second dimension index (j).

Here index i belongs to longitude and index j belongs to latitude.

Return : Return the first dimension index (i) & second dimension index (j) value where we located the nearest grid cell of the input lat, lon passed by the user.

Reference : function 'getArcDistance()'

Example :

eg 1:

```
>>> f = cdms2.open("zos_Omon_ACCESS1-0_rcp45_r1i1p1.xml")
>>> x = f('zos', time=slice(1), squeeze=1)
>>> lat, lon = 10, 300
>>> latidx, lonidx = get1LatLonFromNonRectiLinearGrid(x.getGrid(), lat, lon)
>>> val = x[latidx][lonidx]
```

..note:: Here val is the data value of that lat, lon. Mind that index (i/lonidx) is first dimension and index (j/latidx) is second dimension of the data. Though to access the data here, we need pass latidx as 1st and lonidx as 2nd. Just work out this example, you will understand.

eg 2:

```
>>> latidx, lonidx = get1LatLonFromNonRectiLinearGrid(x.getGrid(), lat, lon)
>>> # extract the time series data points of 10N, 60S position alone
>>> val = f(var, i=(lonidx), j=(latidx))
>>> val.shape
(365,)
```

..note:: Mind that i belongs to longitude & j belongs to latitude.

eg 3: # efficient manner. >>> f = cdms.open("zos_Omon_ACCESS1-0_rcp45_r1i1p1.xml") # getting variable access alone, not the whole data. >>> x = f['zos'] >>> latidx, lonidx = get1LatLonFromNonRectiLinearGrid(x.getGrid(), lat, lon)

..note:: Since we can not directly use latitude, longitude values in the non-rectilinear grid data, we are using 1st dimension (j) for latitude and 2nd dimension (i) for longitude as corresponding indecies to get the data.

Written By : Arulalan.T

Date : 19.09.2012

nonrect_utils.**getArcDistance**(*x1*, *y1*, *x2*, *y2*, *radius=6371*)

func : get the Arc distance

Using equation of distance between two points on arc line.

We can use this function to find out the distances between some list of latitudes & longitudes positions to some other single/list of lat, lon position of the earth. For this we need to pass the radius of the earth.

Inputs : x1, x2 - single latitude / list of latitudes y1, y2 - single longitude/ list of longitudes

But x1 & y1 should be same shape. Also x2 & y2 should be same shape.

radius - radius of the circle. By default it takes the earth's radius in kilometer.

Written By : Arulalan.T

Date : 23.09.2012

7.4 Weather Utils

Author : Arulalan.T

Date: 17.05.2012

`weatherutils.`**`getHighs`**(*data*, *value=1015*)

getHighs [get the centred highs values along with its latitude, longitude] of the passed data. Centred highs means the surrounded lat, lon values are should be less than the centre high value.

arguments: data : cdms2 variable with latitude, longitude axis information value : the centred highs are greate than or equal to this value.

default val is 1015.

return [returning the list containing tuples which are containing lat,]

lon and centred high values. If there is no centred highs less or equal to the passed value arg, then return an empty list.

eg: [(35.5, 85.5, 1027.11), (37.5, 73.5, 1024.29), (31.0, 83.0, 1019.28), (40.0, 91.5, 1015.39)]

Author : Arulalan.T

Date : 17.05.2012

`weatherutils.`**`getLows`**(*data*, *value=1000*)

getLows [get the centred lows values along with its latitude, longitude] of the passed data. Centred lows means the surrounded lat, lon values are should be higher than the centre low value.

arguments: data : cdms2 variable with latitude, longitude axis information value : the centred lows are less than or equal to this value.

default val is 1000.

return [returning the list containing tuples which are containing lat,]

lon and centred low values. If there is no centred lows less or equal to the passed value arg, then return an empty list.

eg: [(30.0, 74.5, 993.28003), (27.0, 80.0, 994.07001), (21.0, 90.0, 998.26001), (34.0, 100.0, 999.53998), (25.0, 95.5, 1000.3)]

Author : Arulalan.T

Date : 17.05.2012

`weatherutils.`**`is_surrounding_greater`**(*data*, *lat*, *lon*)

is_surrounding_greater [Find either the surrounded lat, lon position] values are greater to the centre value on passed lat, lon of the data.

i.e. check the surrounded lat, lon position values against to the passed lat, lon position value. If all the surrounded position values are greater than the center value (passed lat, lon position value) then return this center value.

In any case while checking the surrounded lat, lon position value is lower than the center value, then immediately come out the function by returning False.

If any case while checking the surrounded lat, lon position value is equal to the center value, then again check that point's (position's) surrounded lat, lon position value and do comparison. It has some default limit.

Author : Arulalan.T

Date : 17.05.2012

`weatherutils.`**`is_surrounding_less`**(*data*, *lat*, *lon*)

is_surrounding_less [Find either the surrounded lat, lon position] values are lesser to the centre value on passed lat, lon of the data.

i.e. check the surrounded lat, lon position values against to the passed lat, lon position value. If all the surrounded position values are lesser than the center value (passed lat, lon position value) then return this center value.

In any case while checking the surrounded lat, lon position value is higher than the center value, then immediately come out the function by returning False.

If any case while checking the surrounded lat, lon position value is equal to the center value, then again check that point's (position's) surrounded lat, lon position value and do comparison. It has some default limit.

Author : Arulalan.T

Date : 17.05.2012

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

b

c

e

g

h

n

p

t

v

w

x