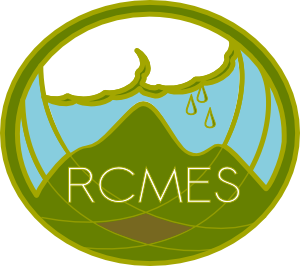
# Regional Climate Model Evaluation Toolkit



Documentation for developers

v1.0 Peter Lean April 2011

### Introduction

The Regional Climate Model Evaluation System (RCMES) has been developed at NASA JPL jointly with researchers at UCLA. This system is used to evaluate the performance of regional climate models by comparing model data against observations.

RCMES has two main components (see Figure 1); a database, known as the Regional Climate Model Evaluation Database (RCMED) and front-end tool known as the Regional Climate Model Evaluation Toolkit (RCMET).

RCMET

Evaluation Toolkit

RCMES

RCMED

Observations Database

Server-side

(hosted at JPL)

Client-side

(installed on user’s computer)

web service

***Figure 1****. Top-level architecture of RCMES system.*

RCMED is a large, scalable database of observational datasets, centrally maintained at JPL. It is designed to allow multiple datasets to be accessed via a common interface and deliver content in a common format. The internals of the database are abstracted from the user by a web service layer, which allows access to the database from any location.

Users can perform evaluation studies using the RCMET tool, which may be installed locally on the user’s machine or set up as a web portal. It has 3 main functions:

1. Access observational data from the RCMED database.
2. Load user’s model data directly from files.
3. Perform evaluation of model against observation including calculation of performance metrics and plotting results as graphs and maps.

RCMET was written using a new library of python functions written specifically for this system.

Note: It is worth considering renaming the python library ‘rcmetlib’ at some point to distinguish it from the end-to-end evaluation tool RCMET.

e.g. import rcmetlib.metrics

This document is a technical description of the RCMET code. It is intended for developers working on the RCMET code. A separate RCMET User Guide should be written for users of the system.

# RCMET Overview

## Overview of functionality

Feature list:

* Retrieval of observations from RCMED
* Loading model data (netCDF, GRIB, HDF supported)
* Re-gridding of data to a common spatial grid
* Temporal averaging/accumulation of data
* Area averaging over user-defined region
* Seasonal Cycle positing
* Statistical metric calculations
  + - Difference
    - Root Mean Square difference
    - Mean Absolute Error
    - Anomaly Correlation
    - Temporal Anomaly Correlation
* Extensible: new metrics can be added to the system by scientists
* Map plots
* Time series plots

## RCMET python package and front-end tool

RCMET consists of a library of python functions (roughly 3000 lines of code) to provide access to the RCMED observation database, read model data and perform commonly required model evaluation tasks.

The rcmes/rcmetlib package is useful in it’s own right for scientists developing in python to accomplish complex tasks in a single line of code.

In addition, the RCMET tool is a front-end interface written in python which binds the various RCMET functions together into an end-to-end model evaluation tool. Users have several options for controlling and interacting with the RCMET tool. They can enter their choices either:

1. using an interactive GUI
2. interactively from the command line
3. by setting up a configuration file.

The rcmes/rcmetlib package categorizes functions into separate files according to their role. For example, functions related to interaction with the database are stored in rcmes/db.py. Functions performing metric calculation are stored in rcmes/metric.py .

db.py - functions related to interface with the RCMED database

files.py - functions involving file IO e.g. loading model data from files

process.py - functions performing various processing tasks e.g. re-gridding

metrics.py - functions to calculate statistical metrics, model v obs

plots.py - plotting functions e.g. map and time series plots

do\_rcmes\_processing\_sub.py - the end-to-end evaluation tool

-a single function called with a long argument list of user options

rcmet\_ui.py - the command line User Interface to collect user options

and call do\_rcmes\_processing\_sub.

## Dependencies on other python packages

The code makes extensive use of the several other python packages:

1. **numpy**; an extremely useful matrix package that allows very efficient matrix calculations (both in terms of performance efficiency and coding efficiency)
2. **SciPy**; a range of useful scientific routines (including efficient regridding libraries)
3. **PyNIO**; an IO package for reading netCDF, GRIB and HDF files through a common interface, which also uses numpy arrays.
4. **PyNGL**; a graphics package for making map plots.
5. **Matplotlib**; for graph plots.

# How to run RCMET

### Running RCMET from the command line

1. install all python package dependencies:
   1. numpy
   2. scipy
   3. PyNIO
   4. PyNGL
   5. Matplotlib
2. download the source code from the RCMES subversion repository:

svn checkout http://oodt.jpl.nasa.gov/repo/projects/wrm/trunk/rcmet/src/main/python

1. cd python
2. Run the command line user interface:

rcmet\_ui.py

1. Enter your selections following the on-screen prompt.

### Running RCMET using a configuration file

Instead of having to enter the user options one by one every time, sometimes it is preferable to run RCMET in batch mode using a configuration file. The configuration file is simply a list of answers to the questions asked by rcmet\_ui.py stored in a text file, one answer per line.

The easiest way to make a configuration file is to follow the instructions in “Running RCMET from the command line” (above) and run rcmet\_ui.py in interactive mode, but copy and paste each of your answers into into an empty text file.

Then to run rcmet\_ui.py using the configuration file:

rcmet\_ui.py < my\_rcmet\_config\_file1.txt

### Running RCMET using the GUI

[to be added – instructions on installing and running the GUI]

# Code overview

***Figure 2****. Flow chart of processing in RCMET.*

1. Collect user options (either from GUI, interactive command line or configuration file)
2. Retrieve observations from the database
3. Load model data from local files
4. Perform spatial re-gridding so that observations and model data are on a common grid
5. (optional) Perform temporal re-gridding so that observations and model data are on a common grid
6. (optional) Area-averaging over user defined masks (or over whole domain)
7. (optional) Seasonal cycle compositing
8. Metric calculation (e.g. anomaly correlation between model and obs)
9. Plot production (maps, time series, Taylor diagrams etc)

modelData, obsData are 3D masked numpy arrays (time,lat,lon). Masked arrays store both data and a mask corresponding to the data. Missing data have a mask value of True, while all other data have a mask value of False.

Times are stored as lists of python datetime objects throughout the code to allow consistent date comparison/arithmetic between model and observations.

Latitude, longitude are two dimensional numpy arrays.

## Conventions used in RCMES

Some rules that cannot be broken:

* All longitudes are from -180 to 180 (i.e. NOT 0 to 360).
* All times are stored as python datetime objects.
* All precipitation data are stored in units of mm/day.
* All temperatures are stored in units of Kelvin.

### Retrieving data from the RCMED database

A web service layer provides abstraction of the data in the RCMED database. This web service provides stability to the RCMES system as it allows the interface to RCMED to remain unchanged regardless of future changes to the underlying architecture or technology used in the database itself. Future developments to RCMED will therefore be completely transparent to users (including RCMET developers), with no code changes required to RCMET.

The RCMED web service is accessed via a URL call from python. A series of arguments allow subsets of data to be retrieved, e.g. only retrieve data within a certain latitude/longitude/time range.

The RCMET code that deals with the database retrievals is stored in rcmes/db.py. The python urllib package is used to make the url request, and the returned text data is saved to a file, before being read in line by line and parsed into appropriate data structures.

Times are parsed into python datetime structures (to take advantage of the complex date math methods provided by this class).

Data is stored in a 3d masked numpy array with dimensions, time, lon, lat.

The missing data mask of the masked array is set appropriately.

## Local caching of database retrievals

Retrievals from the database typically take on the order of 5-50 minutes (depending on the volume of data requested). This is the main bottleneck in the system and so local caching of retrieved data can help to significantly speed up repeated tasks.

There are several usage scenarios where the same data may be repeatedly requested from the database. For example, if you are evaluating 10 different models against the same observations, one at a time.

While the database does have it’s own internal cache which speed up the processing times on repeated queries, the data still needs to be transferred from the server to the client. Often this may involve the transfer of several hundred megabytes of data.

To avoid this unnecessary data transfer, a local cache of retrieved data is stored on the client’s computer. The query string used to access the web service is used as the cache filename. Every time a query is made, RCMET first checks in the cache if the query has been made before, and if so the data is loaded directly from the local cache file (within seconds) without accessing the database at all. This substantially speeds up the entire process when repeating evaluation against the same observational data and improves the user experience considerably.

The local cache files are stored as python “pickle” files, i.e. a raw binary memory dump. These are significantly faster to load in than text files.

### Loading model data

The PyNIO library developed at NCAR is used to provide read access to netCDF, GRIB, GRIB2 and HDF files using a common interface.

Latitude, longitude, data and times are all stored in the same structures as those from the database to make them directly comparable.

## Complications of dealing with model netCDF data files of undetermined internal structure

Model data are typically stored in netCDF or GRIB format. While GRIB files have a fairly rigid internal data structure, netCDF files are far more flexible and allow data to organized in an almost infinite number of ways internally. Consequently, although two models may both output data in netCDF, the way the data is organized within those files may be completely different in each. This makes the task of making a generic model file reader very complicated.

For example, two models (model A and model B) may both output precipitation data. Model A may store the data with a variable name ‘Precip’ while Model B may call it ‘precipitation’. Model A may store the geospatial grid information in two, two-dimensional data arrays named ‘latitude’ and ‘longitude’, while model B may store the same information in 1d arrays, each containing the unique latitudes and longitudes, called ‘LAT’ and ‘LON’. Similarly, the times may be stored in one model as an integer describing the number of seconds since the start of the model run, while another may store it as the number of months since an arbitrary base time. One model may call the time variable ‘Time’ and another ‘t’.

Ideally, the user should choose the variable of interest from a list of available variables (e.g. precipitation, temperature at 2m, sea level pressure etc). Then the system will automatically load that data and the latitude, longitude and time meta data associated with that data. Therefore, the system needs to automatically detect 3 variables: *latitude*, *longitude* and *time*. The system cannot anticipate in advance what these will be called and the strategy adopted here is:

1. look for matches between variable names in the file against a list of common variable names.
   1. Latitude, lats, lat, latitudes, xlat etc
   2. Time, times, t, dates etc
2. If no match is found, display a list of variable names in the file to the user and ask them which one corresponds with latitude, which corresponds with longitude and which corresponds with time.

Since all latitude and longitude data in RCMET are stored as 2d arrays. Therefore, if the model data stores 1d arrays of unique latitudes and longitudes, then 2d arrays are generated automatically for consistency throughout the code.

This method should allow most model files to be read in automatically and nearly all files to be loaded with a little help from the user.

## Collection of user choices

User options are collected using either the web-based RCMET GUI, or from the command line using rcmet\_ui.py .

## Comparing like with like

The first part of the code performs processing to make sure that like is compared with like. For example, in the vast majority of usage scenarios, the model data will be on a different spatial grid than that of the observations. Spatial re-gridding is necessary to make sure that we are comparing like with like.

Similarly, the temporal resolution of the model and observations data may be different. Time averaging can be performed to make sure that like is compared with like.

Future work:

When using satellite data, more complex processing may be required to make appropriate comparisons. For example, AIRS data is categorized into ascending and descending passes. These provide data at local morning and local evening time. Thus daily temperature data from AIRS is not the daily mean temperature, but just a snapshot from one time of day. To compare like with like in this situation, model data must be provided at sub-daily resolution and the nearest time taken (which depends on the longitude of the grid point). Currently, this functionality has not been implemented in RCMET.

## Temporal ‘re-gridding’

The standard deviation of daily mean values over a year is different from the standard deviation of monthly mean values over a year. Therefore, to make sure you are comparing like with like, some form of temporal ‘re-gridding’ is sometimes necessary. For example, if the observation data is daily and the model data is monthly, then the observation data must first be averaged into monthly values.

This is not necessary for all metrics (e.g. the bias will come out the same regardless of whether daily mean or monthly mean data are provided).

## Missing data strategy

Numpy has a special class of arrays called Masked Arrays which contain both data and a separate mask array (filled with True or False). These allow very efficient calculations on data containing missing data, e.g. mydata.mean() will calculate the average of the data values while ignoring values which are flagged in the mask as being missing data.



***Figure 3.*** *Schematic diagram to illustrate how missing data is categorized when averaging from daily to monthly data.*

However, when re-gridding data spatially or temporally, a lot of thought must be made on the most appropriate way to deal with missing data.

## Method for missing data classification during temporal re-gridding and calculating metrics over a time period

For example, consider the example of comparing monthly model data against daily observational data from the database. The observation data must first be averaged in time from daily into monthly data. Satellite data often have missing data, which varies spatially with time due to the time varying satellite overpasses. For a given location, there may be 20 days data out of a possible 30 days in the month. The average monthly value could be calculated from the existing 20 data points, but at what point should the monthly value for that location be considered missing data? RCMET defines a tolerance level for missing data when calculating time aggregate statistics. It uses a fractional threshold to decide if a data point is classified as missing data or not. For example, if the threshold in the code is set to 0.5, then if greater than 50% of the daily data points are missing data, then the monthly value is deemed to be a missing data point.

Furthermore, when calculating a 10-year mean bias from the previously calculated monthly data, a second missing data categorization is required. If greater than 50% of months in the 10 year evaluation period are missing data, then the bias at that location is also deemed to be missing data.

Step 1: conversion of daily data into monthly data (to match the model data). If more than 50% of the daily data in a given month are missing data (black) then the resulting monthly value will be flagged as missing data.

Step 2: If more than 50% of the monthly data are flagged as missing then the period aggregate metric will also be flagged as missing.

## Method for missing data classification during spatial re-gridding

When re-gridding data spatially, care has to be taken along the boundaries of missing data regions. RCMET takes a conservative approach and if any of the data in the new grid cell was missing data on the original grid then the whole of the new grid cell is deemed to be missing data.

For example, in the case of re-gridding data from an original model grid at 0.1x0.1deg resolution to a observation grid at 2.5x2.5deg resolution. Each grid cell on the new grid corresponds to 625 grid cells on the old grid. If just one of those original grid cells was flagged as missing data then the entire new grid cell will be deemed to be missing data too.

This conservative method avoids any possibility of unrepresentative data providing misleading evaluation results.

## Area-weighted averaging

RCMET provides functionality to calculate area-weighted means over a user defined region.

The weighted mean of the values is calculated with weights determined by the area of each grid cells.

## Plotting routines

The type of plot generated by RCMET is determined automatically given the dimensions of the metric data. For example, if the metric was to calculate RMS difference at every grid point over a 30-year period, then the metric data array will have two dimensions. Consequently, RCMET will generate a map plot.

If the user selected to calculate the anomaly correlation over the same 30-year period, then the output data would be one dimensional (one value per time). In this situation RCMET would produce a time series plot.

i.e. the type of plot generated depends on the dimensions of the metricData array.

metricData array = 1D > time series plot

metricData array = 2D > map plot

metricData array = 3D > sequence of map plots

## Guidelines for adding new metric calculation routines to RCMET

RCMET is intended to be extensible, i.e. researchers should be able to add new statistical metrics to the system.

The example below provides a step-by-step guide on how to add a new metric into the system.

Here are some guidelines and things to consider before writing your metric code.

Figure 2 shows where the metric calculation occurs in the RCMET workflow.

It occurs after the spatial and temporal re-gridding, i.e. by the time the data reaches the metric code, both observations and model data should be on the same spatial grids and valid at the same times. The data arrays should be of equal size.

Input:

The metric code should work with 3D, 2D and 1D input data arrays of observation and model. i.e. data may vary with time and space (3D), or just a single time (2D), or even a simple time series of area-averaged values (1D).

Appropriate consideration should be to deal with missing data.

Output:

The output from a metric function should be a masked numpy array and can have different dimensions to the input data. For example, a metric that calculated the different in domain mean values would have 3D input arrays, but produce a one dimensional output array (one value per time). This is perfectly acceptable and the downstream plotting routines will adjust detect a 1d array and automatically produce a time series plot as the output.

# Example adding new metric calculation into RCMET code

e.g. Sven the scientist wants to add a simple new RCMET metric to calculate the square of the difference at every grid point. These are the steps he takes:

1. He downloads the latest code from the repository into a local working directory:

svn checkout <http://oodt.jpl.nasa.gov/repo/projects/wrm/trunk/rcmet/src/main/python/lib>

1. …navigates to the rcmes code directory:

cd python/rcmes

1. …and opens the file that contains the metric code in his favorite text editor:

gedit metrics.py

1. Then he can add his new metric code as a new function at the bottom of the file, including plenty of comments to make the code understandable to other developers:

e.g.

################################################

# Difference Squared

################################################

def calc\_difference\_squared(t1,t2):

'''

# Calculate mean difference between two fields

# over time for each grid point

# Input:

# t1 – masked numpy array representing mdoel data

# (indexed in order: [time, longitude,latitude])

# t2 – masked numpy array representing obs data

# (indexed in order: [time, longitude,latitude])

#

# Output:

# diffsq – masked numpy array of metric data.

#

# Sven T. Scientist April 2011

'''

print 'Calculating difference'

# Use array operators to calculate difference

# at each grid point and then square the results

diffsq = (t1 - t2)\*\*2

return diffsq

NB. In this simple example, the handling of missing data is dealt with adequately by the masked array objects, however if more complex metrics were to be calculated then masking of missing data may have to be defined explicitly. Please test your code thoroughly to make sure that it behaves as expected.

1. Edit do\_rcmes\_processing\_sub.py (Part 7) to make the call to the metric if selected by the user:

e.g.

if metricOption=='diffsq': #NB. This value must match that in the config file

metricData = rcmes.metrics.calc\_difference\_squared(modelData,obsData)

metricTitle = 'Difference Squared' # used in plot titles

1. Now that the metric is written, the metric configuration file must be adjusted so that the user interface can see that the new metric exists.

TODO: currently, the list of available metrics is hardwired into the GUI and rcmet\_ui.py. In future, it would be desirable to generate the list of available metrics from a configuration file. This allows new metric code to be added without making any changes to the user interface code.

For example,

* gedit metric\_config.txt &

Add the following line to the bottom:

Difference squared, diffsq, calc\_difference\_squared

-where ‘Difference squared’ is the text that will appear in the user interface to describe the metric

-where ‘diffsq’ is the name used in the user interface code,

i.e. if user selects this then metricOption = ‘diffsq’

-and ‘calc\_difference\_squared’ is the name of the function in the rcmes.metrics file that you have just created.

1. Test that it works as expected with a range of different model and observational data.
2. Email the rcmes-dev@jpl.nasa.gov list to give all developers a summary of the latest additions.
3. Commit the new code back to the repository:

svn commit

–a text editor will pop up for you to insert an informative message to the commit log,

e.g.

“Added new metric to calculate square of differences at every grid point. This code has been tested successfully using CORDEX and WRF model data and AIRS, TRMM and MODIS observational data.”

# Future work

Sub-setting model data to use data from nearest time to satellite overpass for certain satellite datasets.

Improved user-customization options for plots. E.g. give user more control over line thickness, color scheme, titles etc.

Output metric data to netCDF file