tscale Documentation

Release 1.0

jfiddes

Oct 13, 2020

Contents

TopoSCALE is a downscaling tool that uses the well-resolved description of the atmospheric column provided by atmospheric models, together with high-resolution digital elevation models (DEMs), to downscale coarse-grid climate variables to a fine-scale subgrid. The main aim of this approach is to provide high-resolution driving data for a land-surface model (LSM). Full desciption available in this publication: http://www.geosci-model-dev.net/7/387/2014/

CHAPTER 1

Dependencies

Python specific dependencies are bundled in the virtual env distributed with the code repository and defined in the requirements.txt file . Additional system requirments:

- linux (not tested on other platforms)
- python 2.7 (to be updated to py3)
- **gdal:** apt-get install gdal
- pip:

```
apt-get install pip
```

- Climate Data Operators (CDO) used to concat monthly netcdfs downloaded from ECMWF:

```
apt-get install cdo
```

- ECMWF CDS API set up:

```
https://confluence.ecmwf.int/display/CKB/
↪How+to+migrate+from+ECMWF+Web+API+to+CDS+API
```

CHAPTER 2

Quickstart

Get the code using git:

```
git clone https://github.com/joelfiddes/tscaleV2.git
```

or direct download:

```
https://github.com/joelfiddes/tscaleV2/archive/master.zip
```

Activate virtual environment:

```
cd ./tscale
source env/bin/activate
```

CHAPTER 3

Modes of operation

POINT: Generates downscaled timeseries for a specific point defined by long/lat

TSUB: Generates downsclaed timeseried for TopoSUB cluster centroids

GRID: Generates a 2D grid of surface fields corresponding to input DEM.

CHAPTER 4

Tutorial

The github repo contains a full example to set up a first prototype and understand how the code runs.

CHAPTER 5

Documentation

5.1 TopoSCALE components

TopoSCALE has 5 components:

5.1.1 INI file

The INI file defines all required options for a run.

5.1.2 Input plugins

- preprocesses product (resampling, conversions)
- converts to generic python class structure

Available plugins: *era5*

5.1.3 Core engine

- accepts generic data structures
- algorithms
- outputs generic python class structure

5.1.4 Output plugins

- writes specific output formats (CSV, NetCDF)
- writes model specific outputs (SMET, GEotop, Cryogrid)
- writes grids (NetCDF, tiff)

5.1.5 Optional modules

- retrieve and preprocess data products (various api dependencies here)

5.2 fetch_era5

‘ Retrieve ecmwf data from new CDS api.

fetch_era5.**era5_request_plev**(*year*, *month*, *bbox*, *target*, *product*, *time*, *plevels*)
CDS plevel api call

fetch_era5.**era5_request_surf**(*year*, *month*, *bbox*, *target*, *product*, *time*)
CDS surface api call

fetch_era5.**eraCat**(*wd*, *grepStr*)
*** CDO *** Concats monthly era (interim or 5) files by some keyword *grepStr*. Depends on CDO. - reads monthly ERA5 data files (MARS efficiency) - concats to single file timeseries

Syntax: cdo -b F64 -f nc2 mergetime SURF* SURF.nc

Parameters

- **wd** – directory of era monthly datafiles
- **grepStr** – “PLEV” or “SURF”

Example

wd=/home/eraDat/ grepStr= “PLEV” eraCat(“/home/joel/mnt/myserver/sim/wfj_era5/eraDat/”, “PLEV”)

fetch_era5.**eraCat5d**(*wd*, *grepStr*)
*** NCO *** Updated method to deal with 5D of ensemble datasets NOT supported by CDO Concats monthly era (interim or 5) files by some keyword *grepStr*. Depends on NCO. sudo apt-get install nco - reads monthly ERA5 data files (MARS efficiency) - concats to single file timeseries - 2 steps

- assign time to “record” dimeanion in firstfile
- concat files

Parameters

- **wd** – directory of era monthly datafiles
- **grepStr** – “PLEV” or “SURF”

Example

wd=/home/eraDat/ grepStr= “PLEV” eraCat(“/home/joel/mnt/myserver/sim/wfj_era5/eraDat/”, “PLEV”)

fetch_era5.**retrieve_era5_plev**(*product*, *startDate*, *endDate*, *eraDir*, *latN*, *latS*, *lonE*, *lonW*, *step*, *plevels*)
Sets up era5 pressure level retrieval. * Creates list of year/month pairs to iterate through. * MARS retrievals are most efficient when subset by time. * Identifies preexisting downloads if restarted. * Calls api using parallel function.

Parameters

- **config** – config object defining INI

- **eraDir** – directory to write output

latN: north latitude of bbox latS: south latitude of bbox lonE: easterly lon of bbox lonW: westerly lon of bbox

Returns Monthly era pressure level files.

fetch_era5.**retrieve_era5_surf**(*product*, *startDate*, *endDate*, *eraDir*, *latN*, *latS*, *lonE*, *lonW*, *step*)
Sets up era5 surface retrieval. * Creates list of year/month pairs to iterate through. * MARS retrievals are most efficient when subset by time. * Identifies preexisting downloads if restarted. * Calls api using parallel function.

Parameters

- **product** – "reanalysis" (HRES) or "ensemble_members" (EDA)
- **startDate** –
- **endDate** –
- **eraDir** – directory to write output
- **latN** – north latitude of bbox
- **latS** – south latitude of bbox
- **lonE** – easterly lon of bbox
- **lonW** – westerly lon of bbox

Returns Monthly era surface files.

fetch_era5.**retrieve_era5_tpmm**(*startDate*, *endDate*, *eraDir*, *latN*, *latS*, *lonE*, *lonW*)
retrive monthly tp means to correct 6 or 3h TP retrieval documented era5 here: https://confluence.ecmwf.int/display/CKB/ERA5+data+documentation#ERA5datadocumentation-Monthlymeans values are mean daily for that month, weight by number of days in month eg annualtotal = sum(wfj_m*c(31,28,31,30,31,30,31,31,30,31,30,31))

5.3 main

5.4 era5

ERA5.py

ERA5 IO plugin

This module rquires input files:

- **PLEV.nc:** all pressure level variables for entire domain (single/multiple CGCs) [time X grid cell X level X variable]
- **SURF.nc:** all surface variables for entire domain (single/multiple CGCs) [time X grid cell X variable]

This plugin contains methods to:

- classes for both pressurelevel and surface objects
- **extract coarse grid cell (CGC) based on longitude latitude of station** or CGC centre
- convert values to toposcale standard
- writes single parameter files
- TopoSCALE standard input:

- air temperature - K
- precipitation - mmh*1
- shortwave Wm**2
- longwave Wm**2
- wind - U and V vectors
- time - iso
- relative humidity

Example

Initialise new era5 instance:

p=era5.Plev(fp, stat.lat, stat.lon)

Attributes:

Todo:

class era5.**Plev**(*fp*, *mylat*, *mylon*)

Makes a plev object which is array of all pressure level variables processed to standard toposcale units

Parameters

- **fp** – filepath to concatenated PLEVEL.nc file
- **mylat** – latitude of station or grid centre
- **mylon** – longitude of station or gride centre

Example

p=Plev(fp) varnames=p.varnames()

addShape()

adds two dimensions of time and levels

addTime()

add time vector and convert to ISO Return datetime objects given numeric time values. The units of the numeric time values are described by the units argument and the calendar keyword. The returned datetime objects represent UTC with no time-zone offset, even if the specified units contain a time-zone offset.

calender options defined here: http://unidata.github.io/netcdf4-python/#netCDF4.num2date

addVar(*varname*, *dat*)

rename attribute

extractCgc(*var*, *startIndex*, *endIndex*)

extract variable and cgc (now np array) dimension order: time, level, lat,lon

extractCgc5d(*var*, *member*, *startIndex*, *endIndex*)

extract variable, ensemble memeber and cgc (now np array) from 5d nc files eg era5 ensemble (extra dimension ensemble 'number')

dimension order: time, ensemble, level, lat,lon

getVar(*var*)

extract variables (remains an nc object)

getVarNames()
returns list of variables excluding dimension names time, lon,lat, level

class era5.**Plev_interp**(*fp*, *mylat*, *mylon*)
Makes a plev object which is array of all variables processed to standard toposcale units and interpolated to x, y, z

Parameters **fp** – filepath to concatenated PLEVEL.nc file

Example

p=Plev(fp) varnames=p.varnames()

interpCgc(*var*)
interp variable and cgc (now np array) !!NOT FINISHED!! Perhaps a different class as here includes z interpolation

class era5.**Surf**(*fp*, *mylat*, *mylon*)
Makes a plev object which is array of all surface variables processed to standard toposcale units

Parameters **fp** – filepath to concatenated PLEVEL.nc file

Example

p=Plev(fp) varnames=p.varnames()

extractCgc(*var*, *startIndex*, *endIndex*)
extract variable and cgc (now np array)

extractCgc4d(*var*, *member*, *startIndex*, *endIndex*)
extract variable, ensemble memeber and cgc (now np array)from 4d nc files eg era5 ensemble (extra dimension ensemble 'number')

dimension order: time, ensemble, lat,lon

gridEle()
compute surface elevation of coarse grid

instRad(*step*)
Convert SWin from accumulated quantities in J/m2 to instantaneous W/m2 see: https://confluence.ecmwf.int/pages/viewpage.action?pageId=104241513

Parameters **step** – timstep in seconds (era5=3600, ensemble=10800)

Note: both EDA (ensemble 3h) and HRES (1h) are accumulated over the timestep and therefore treated here the same. https://confluence.ecmwf.int/display/CKB/ERA5+data+documentation

tp2rate(*step*)
convert tp from m/timestep (total accumulation over timestep) to rate in mm/h

Parameters **step** – timstep in seconds (era5=3600, ensemble=10800)

Note: both EDA (ensemble 3h) and HRES (1h) are accumulated over the timestep and therefore treated here the same. https://confluence.ecmwf.int/display/CKB/ERA5+data+documentation

era5.**series_interpolate**(*time_out*, *time_in*, *value_in*, *cum=False*)
Interpolate single time series. Convenience function for usage in scaling kernels. time_out: Array of times [s] for which output is desired. Integer. time_in: Array of times [s] for which value_in is given. Integer. value_in: Value time series. Must have same length as time_in. cum: Is valiable serially cummulative like LWin? Default: False.

5.5 tscale

tscale.py

Downscale forcing from atmospheric grid.

This module downscales all required fields, TA, RH, WS, WD, LW, SW based on properties of a stat object that represents a station (Mode=POINT) or a cluster centroid (mode=TSUB) or a pixel (mode=GRID).

Example

Initialise a new tscale instance with:: $ p.plevels()

Attributes:

Todo:

- Precipitation look up table hardcoded in method

---

class tscale.**tscale**(*z*)

Interpolate pressure level vector from single cgc in the TopoSUB use case

Args:

Example:

Rh2Wvp(*RH*)

'compute water vapour pressure

Parameters

- **humidity (%** (*RH=relative*) – 0-100)
- **temperature** (*tair=air*) –

addVar(*varname*, *datInterp*)

Assign correct attribute name

lwin(*sob*, *tob*, *stat*)

Convert to RH (should be in a function). Following MG Lawrence DOI 10.1175/BAMS-86-2-225

relHumCalc(*tdew*)

convert ERA interim surface dewpoint temp to relative humidity based on d2m and t2m :param tdew=dewpoint temp: :type tdew=dewpoint temp: kelvin :param tair=air temperature: :type tair=air temperature: kelvin

swin(*pob*, *sob*, *tob*, *stat*, *dates*)

toposcale surface pressure using hypsometric equation - move to own class

EDITS Feb 20 2020:

-See https://www.evernote.com/Home.action?login=true#n=78e92684-4d64-4802-ab6c-2cb90b277e44&s=s500&ses=1&sh=5&sds=5&x=& -

EDITS Jul 11 2019:

- removed elevation scaling as this degrade results - at least in WFJ test- Kris?
- reimplemnt original additative ele method - better than beers, or at least less damaging
- removed mask, why is this causing problems?

- dotprod method (corripio) does not seem to work - I dont think it ever did as we used SWTopo ==FALSE
- use Dozier cos corrction method (as in paper right)
- So ... we have ele, illumination angle, self shading and svf correction. We do not have horizon correction (partly in self shading of course)

swin_joel()
PARTITION

tscale1D(*dat*, *stat*)
Example function with types documented in the docstring.

PEP 484 type annotations are supported. If attribute, parameter, and return types are annotated according to PEP 484, they do not need to be included in the docstring:

Interpolates vector of pressure level data to station elevation at each timestep to create timeseries of downscaled values at station elevation

Parameters

- **param1** (*int*) – The first parameter.
- **param2** (*str*) – The second parameter.

Returns The return value. True for success, False otherwise.

Return type bool

wind(*tob*)
methods for working with wind

windCorRough(*tob*, *pob*, *sob*, *stat*)
corrects wind speed according to true rougness values

5.6 solarGeom

Solar geometery

A set of methods that compute solar geometry objects. This is a Python implementation of Javier Corripio's R-package 'insol'. Documented here:

https://doi.org/10.1080/713811744

validated against original R-package code from Corripio

Example:

Attributes:

Todo:

solarGeom.**declination**(*jd*)

Computes the declination of the Sun for a given Julian Day.

jd: Julian Day and decimal fraction. cat("USAGE: declination(jd)

")

solarGeom.**eqtime**(*jd*)

Computes the equation of time for a given Julian Day.

jd: Julian Day and decimal fraction. cat("USAGE: eqtime(jd)

")

solarGeom.**hourangle**(*jd*, *longitude*, *timezone*)

Hour angle, internal function for solar position.

jd: Julian Day and decimal fraction. latitude: Latitude of observer in degrees and decimal fraction. longitude: Longitude of observer in degrees and decimal fraction. timezone: in hours, west of Greenwich is negative eg CH is "1"

cat("USAGE: hourangle(jd,longitude,timezone)

julian day, degrees, hours. Return radians

")

solarGeom.**hourangleMD**(*jd*, *longitude*, *timezone*, *statsize*, *timesize*)

Hour angle, internal function for solar position.

THIS IS MLTIDIMENSIONAL VERSION LONGITUDE/TIMEZONE are vectors (of stations) not scalars

jd: Julian Day and decimal fraction. latitude: Latitude of observer in degrees and decimal fraction. longitude: Longitude of observer in degrees and decimal fraction. timezone: in hours, west of Greenwich is negative eg CH is "1"

cat("USAGE: hourangle(jd,longitude,timezone)

julian day, degrees, hours. Return radians

")

solarGeom.**normalvector**(*slope*, *aspect*)

Calculates a unit vector normal to a surface defined by slope inclination and slope orientation.

slope: slope of position in degrees aspect: aspect of position in degrees print("USAGE: normalvector(slope,aspect)

")

solarGeom.**sunpos**(*sunv*)

Returns a matrix of azimuth and zenith angles of the sun given the unit vectors from the observer to the direction of the sun. Plus sun elevation.

sunv: sunvector #print("USAGE: sunpos(sunvector) 3D vector")

solarGeom.**sunposMD**(*sunx*, *suny*, *sunz*)

Returns a matrix of azimuth and zenith angles of the sun given the unit vectors from the observer to the direction of the sun. Plus sun elevation.

sunv: sunvector #print("USAGE: sunpos(sunvector) 3D vector")

solarGeom.**sunvector**(*jd*, *latitude*, *longitude*, *timezone*)

Calculates a unit vector in the direction of the sun from the observer position

jd: Julian Day and decimal fraction. latitude: Latitude of observer in degrees and decimal fraction. longitude: Longitude of observer in degrees and decimal fraction. timezone: Time zone in hours, west is negative. # cat("USAGE: sunvector(jd,latitude,longitude,timezone)

values in jd, degrees, hours

")

solarGeom.**sunvectorMD**(*jd*, *latitude*, *longitude*, *timezone*, *statsize*, *timesize*)

Calculates a unit vector in the direction of the sun from the observer position

jd: Julian Day and decimal fraction. latitude: Latitude of observer in degrees and decimal fraction. longitude: Longitude of observer in degrees and decimal fraction. timezone: Time zone in hours, west is negative. # cat("USAGE: sunvector(jd,latitude,longitude,timezone)

values in jd, degrees, hours

")

solarGeom.**to_jd**(*dt*)

dt: python dattime object Converts a given datetime object (dt) to Julian date. Algorithm is copied from https://en.wikipedia.org/wiki/Julian_day All variable names are consistentdatetime.datetime.now() with the notation on the wiki page.

cite:https://github.com/dannyzed/julian/blob/master/julian/julian.py

test : dt = datetime.datetime.now()

Parameters

- **fmt** –
- **dt** (*datetime*) – Datetime object to convert to MJD
- **Returns** –
- **-------** –
- **jd** (*float*) –

5.7 Editing docs

5.7.1 restructuredtext

http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

5.7.2 Docstrings

Follow google style eg.

"""Example Google style docstrings.

This module demonstrates documentation as specified by the Google Python Style Guide. Docstrings may extend over multiple lines. Sections are created with a section header and a colon followed by a block of indented text.

Example: Examples can be given using either the `Example` or `Examples` sections. Sections support any reStructuredText formatting, including literal blocks:

```
$ python example_google.py
```

Section breaks are created by resuming unindented text. Section breaks are also implicitly created anytime a new section starts.

Attributes:

module_level_variable1 (int): Module level variables may be documented in either the `Attributes` section of the module docstring, or in an inline docstring immediately following the variable.

Either form is acceptable, but the two should not be mixed. Choose one convention to document module level variables and be consistent with it.

Todo:

- For module TODOs
- You have to also use `sphinx.ext.todo` extension

“””

CHAPTER 6

Search code

- genindex
- modindex
- search

Python Module Index

Index