PyCAPS Documentation

Release 0.1

David John Gagne **Nate Snook** **Tim Supinie**
Bryan Putnam **Jon Labriola**

May 20, 2016

CONTENTS

CHAPTER

ONE

PYCAPS.DERIVE PACKAGE

1.1 Submodules

1.2 pycaps.derive.derive_functions module

air_density(***kwargs*)

Calculates air density for given pressure and potential temperature arrays of arbitrary shape.

Parameters ****kwargs** – Must contain 'p' (pressure in Pa) and 'pt' or 't' (potential temperature or temperature, either in K). May also contain 'qv' (water vapor mixing ratio in kg/kg). Any arguments other than these are ignored.

Returns Air density in kg/m^3 as an array of the same shape as p and pt

bunkers_motion(*direction='RM'*, ***kwargs*)

Defines the Storm motion based upon a vertical sounding using the Bunkers et al. (2000) storm motion approximation.

Args: direction: The direction fo the storm motion, 'RM' or 'LM.

****kwargs: Must contain 'u' (u wind component), 'v' (v wind component), 'zp' (physical model height), and 'zps**
u,v,zp,zpsoil must be numpy arrays with 3 dimensions. and the axes are assumed to be in (NZ, NY, NX) order. Any kwargs not 'u', 'v','zp','zpsoil' will be ignored

Returns: A numpy array containing the storm motion (m/s). The array is two dimensions (NY,NX)

calc_mesh(***kwargs*)

Calculated the Maximum Estimated Size of Hail (MESH) Witt et al. (1998)

Parameters ****kwargs** – 'dBZ' (the reflectivity), 'pt' (potential temperature), 'p' (the pressure),'zp' (the physical height). dBZ,pt,p,and zp must be numpy arrays with 3 dimensions and the axes are assumed to be in (NZ, NY, NX) order. Any kwargs not 'dBZ','pt','p','zp' will be ignored

Returns: The numpy matrix of MESH (mm). The arrays has two dimensions (NY,NX)

closest_grdpt(*convert_scalar=True*, ***kwargs*)

Find the closest grid point above the given height, fast when interpolation is not desired

Parameters

- **convert_scalar** – Convert from scalar to physical height
- ****kwargs** – 'z' (the vertical height), 'zp' (physical model height), z must be a scalar value in meters zp must be numpy arrays with 3 dimensions and the axes are assumed to be in (NZ, NY, NX) order. Any kwargs not 'z','zp' will be ignored

Returns: The numpy matrix with the vertical levels of said height. The arrays has two dimensions (NY,NX)

compute_srh (*zOne=1000*, *zTwo=3000*, *direction='RM'*, ***kwargs*)

Computes storm relative helicity

Parameters

- **zOne** – The bottom height for which the integral begins.
- **zTwo** – The top height for which the integral ends.
- **direction** – The direction fo the storm motion, 'RM' or 'LM.
- ****kwargs** – Must contain 'u' (u wind component), 'v' (v wind component), 'zp' (physical model height), and 'zpsoil' (soil height), u,v,zp,zpsoil must be numpy arrays with 3 dimensions. and the axes are assumed to be in (..., NY, NX) order. Any kwargs not 'u', 'v','zp','zpsoil' will be ignored

Returns: A numpy array containing Storm Relative Helicity (m^2/s^2). The array is the same shape as the input arrays.

dewp_from_qv (***kwargs*)

Get dewpoint temperature from water vapor mixing ratio, using the Clausius-Clapeyron equation.

Parameters ****kwargs** – Must contain 'qv' (water vapor mixing ratio in kg/kg) and 'p' (pressure in Pa). Any other keyword arguments are ignored.

Returns Dewpoint temperature in K.

dewp_from_rh (***kwargs*)

Get dewpoint temperature from relative humidity.

Parameters ****kwargs** – Must contain 'rh' (relative humidity as a fraction from 0-1), 'p' (pressure in Pa), and 'pt' or 't' (potential temperature or temperature, either in K). Any other keyword arguments are ignored.

Returns Dewpoint temperature in K.

dict_fm_recarray (*rec*)

Convert a numpy record array to a Python dictionary.

Parameters **rec** – The record array to convert

Returns A Python dictionary with keys taken from the array data type.

finite_diff (*data*, *coords*, *axis=0*)

Compute a finite difference along a given axis.

Parameters

- **data** – A numpy array of which to take the finite difference.
- **coords** – The coordinates for the axis along which to take the derivative, specified as a numpy array.
- **axis** – The axis along which to take the finite difference; optional, defaults to 0.

Returns An array of the same shape as the data array containing the finite difference.

horiz_convergence (***kwargs*)

Computes horizontal convergence.

Parameters ****kwargs** – Must contain 'u' (u wind component), 'v' (v wind component), 'dx' (grid spacing in the x direction), and 'dy' (grid spacing in the y direction). u and v must be numpy arrays with 2 or more dimensions, and the axes are assumed to be in (..., NY, NX) order. dx and dy must be scalar floats. Any kwargs not 'u', 'v', 'dx', or 'dy' are ignored.

Returns A numpy array containing horizontal convergence in 1/s. The array is the same shape as the input arrays.

hydrometeor_classify(*qc_opt*, *x*, *y*, *ref*, *zdr*, *rhv*, *vel*, *pt*, *p*, *qs*, *hgt*, *zp*)

isotherm_hgt(*isotherm=0*, *convert_scalar=True*, ***kwargs*)

Calculate the the physical height of the gridpoint, one gridlevel above isotherm

Parameters

- **isotherm** – The isotherm height to analyze
- **convert_scalar** – Convert from scalar to physical height
- ****kwargs** – 'p' (the pressure), 'pt' (the potential temperature), 'zp' (physical model height), p,pt,zp must be numpy arrays with 3 dimensions and the axes are assumed to be in (NZ, NY, NX) order. Any kwargs not 'z','zp' will be ignored

  If the temperature is known may use 'T' (the temperature) instead of 'p' and 'pt'

Returns: The numpy matrix with the vertical levels of said isotherm. The arrays has two dimensions (NY,NX)

moist_lapse(***kwargs*)

Compute the moist adiabatic lapse rate at a given temperature and pressure.

Parameters ****kwargs** – Must contain 'p' (pressure in Pa) and either 't' or 'pt' (temperature or potential temperature, either in K). Any other keyword arguments are ignored.

Returns The moist adiabatic lapse rate in K/Pa.

pbl_depth(***kwargs*)

ARPS planetary boundary layer depth calculation. Translated directly from pbldpth() in sfcphy3d.f90.

Parameters ****kwargs** – Must contain 'pt' (potential temperature in K), 'qv' (water vapor mixing ratio in kg/kg), and 'z' (height in m). The arguments must be arrays with the first dimension being height; axis order (NZ, ...). Any other keyword arguments are ignored

Returns Depth of the planetary boundary layer in m.

pmsl(***kwargs*)

Compute pressure at mean sea level, [check my assumption].

Parameters ****kwargs** – Must contain 'z' (height in meters of each point above sea level), 'p' (pressure in Pa at each point), 'pt' or 't' (potential temperature or temperature, either in K), and 'qv' (water vapor mixing ratio in kg/kg).

Returns Pressure at mean sea level in Pa as a numpy array of the same shape as the input arrays.

qv_from_vapr(***kwargs*)

Get water vapor mixing ratio from vapor pressure.

Parameters ****kwargs** – Must contain 'p' (pressure in Pa) and either 'e' (vapor pressure in Pa), 't' (temperature in K), or 'pt' (potential temperature in K). Any other keyword arguments are ignored.

Returns Water vapor mixing ratio in kg/kg

recarray_fm_dict(***dct*)

Convert a Python dictionary to a numpy record array.

Parameters ****dct** – Dictionary keys to convert.

Returns A numpy array with a record data type.

reflectivity_dualmom(***kwargs*)

Compute reflectivity from a dual-moment microphysics scheme.

Parameters ****kwargs** – Must contain 'qv' (water vapor mixing ratio in kg/kg), 'qr' (rain mixing ratio in kg/kg), 'qs' (snow mixing ratio in kg/kg), 'qh' (hail mixing ratio in kg/kg), 'qg' (graupel mixing ratio in kg/kg), 'nr' (total number concentration of rain in #/m^3), 'ns' (total number concentration of snow in #/m^3), 'nh' (total number concentration of hail in #/m^3), 'ng' (total number concentration of graupel in #/m^3), 'p' (air pressure in Pa), and 'pt' or 't' (potential temperature or temperature, either in K). Any other keyword arguments are ignored.

Returns Radar reflectivity in dBZ

reflectivity_lin(***kwargs*)

Compute reflectivity from a Lin microphysics scheme

Parameters ****kwargs** – Must contain 'qv' (water vapor mixing ratio in kg/kg), 'qr' (rain mixing ratio in kg/kg), 'qs' (snow mixing ratio in kg/kg), 'qh' (hail mixing ratio in kg/kg), 'n0rain' (n0 for the rain distribution in 1/m^4), 'n0snow' (n0 for the snow distribution in 1/m^4), 'n0hail' (n0 for the hail distribution in 1/m^4), 'rhosnow' (assumed density for snow in kg/m^3), 'rhoice' (assumed density for ice in kg/m3), 'rhohail' (assumed density for hail in kg/m^3), 'p' (air pressure in Pa), and 'pt' or 't' (potential temperature or temperature, either in K). Any other keyword arguments are ignored.

Returns Radar reflectivity in dBZ

temp_from_theta(***kwargs*)

Get temperature from potential temperature.

Parameters ****kwargs** – Must contain 'p' (pressure in Pa) and 'pt' (potential temperature in K) either as scalars or numpy arrays. Any other keyword arguments are ignored.

Returns Temperature in K

temp_from_vapr(***kwargs*)

Get temperature from vapor pressure, using the Clausius-Clapeyron equation.

Parameters ****kwargs** – Must contain one of two sets of arguments. The first is 'e' (vapor pressure in Pa). The second is both 'qv' (water vapor mixing ratio in kg/kg) and 'p' (pressure in Pa). Any arguments other than these are ignored.

Returns Temperature in K.

theta_e(***kwargs*)

Compute equivalent potential temperature.

Parameters ****kwargs** – Must contain 'pt' (potential temperature in K), 'p' (pressure in Pa), and 'qv' (water vapor mixing ratio in kg/kg).

Returns Equivalent potential temperature in K.

updraft_helicity(*z_bottom=2000.0*, *z_top=5000.0*, ***kwargs*)

Calculates updraft helicity (m^2/s^2) given 3d winds and physical height.

u,v,w should have dimensions of [nz,ny,nx] dx and dy are scalars zp is a matrix of dimensions of [nz,ny,nx]

Parameters

- **u** – u-wind (east-west wind component)
- **v** – v-wind (north-south wind component)
- **w** – w-wind (vertical wind component)
- **dx** – horizontal (east-west) grid spacing in meters
- **dy** – horizontal (north-south) grid spacing in meters
- **zp** – physical height

- **z_top** – Bottom of layer over which to calculate updraft helicity (OPTIONAL: default 2000m)
- **z_bottom** – Top of layer over which to calculate updraft helicity (OPTIONAL: default 5000m)

Returns The updraft helicity in m^2/s^2 in the layer z_bottom to z_top.

vapr_from_qv(***kwargs*)

Get vapor pressure from water vapor mixing ratio.

Parameters ****kwargs** – Must contain 'qv' (water vapor mixing ratio in kg/kg) and 'p' (pressure in Pa). Any other keyword arguments are ignored.

Returns Vapor pressure in Pa

vapr_from_temp(***kwargs*)

Get vapor pressure from temperature, using the Clausius-Clapeyron equation.

Parameters ****kwargs** – Must contain 't' (temperature in K) or 'pt' and 'p' (potential temperature in K and pressure in Pa). Any other keyword arguments are ignored.

Returns Vapor pressure in Pa.

vert_vorticity(***kwargs*)

Computes vertical vorticity.

Parameters ****kwargs** – Must contain 'u' (u wind component), 'v' (v wind component), 'dx' (grid spacing in the x direction), and 'dy' (grid spacing in the y direction). u and v must be numpy arrays with 2 or more dimensions, and the axes are assumed to be in (..., NY, NX) order. dx and dy must be scalar floats. Any kwargs not 'u', 'v', 'dx', or 'dy' are ignored.

Returns A numpy array containing vertical vorticity in 1/s. The array is the same shape as the input arrays.

vert_windshear(*z_one=1000*, *z_two=3000*, ***kwargs*)

Defines a vertical wind shear vector between two specfied heights. Useful in calculating SRH - I integrate between 1-3km.

Parameters

- **zOne** – The bottom height which is considered
- **zTwo** – The top height which is considered
- ****kwargs** – Must contain 'u' (u wind component), 'v' (v wind component), 'zp' (physical model height), and 'zpsoil' (soil height), u,v,zp,zpsoil must be numpy arrays with 3 dimensions. and the axes are assumed to be in (NZ, NY, NX) order. Any kwargs not 'u', 'v','zp','zpsoil' will be ignored

Returns: Two arrays of the U and V components of vertical wind shear (m/s). The arrays are two dimensional (NY,NX).

vg_tensor(***kwargs*)

Compute the components of the velocity gradient tensor.

Parameters ****kwargs** – Must contain 'u', 'u', 'w', 'x', 'y', and 'z'. u, v, and w are the zonal, meridional, and vertical components of wind, respectively, and must be 3-dimensional numpy arrays. The order of the axes is assumed to be (NZ, NY, NX). x and y should be 1-dimensional arrays containing the x and y coordinates for the domain, and z should be 3-dimensional arrays containing the vertical coordinates for the domain (e.g. 'zp' from an ARPS history file). All keywords that are not the aforementioned are ignored.

Returns The 9 components of the velocity gradient tensor as a numpy record array. The records are 'dudx', 'dudy', 'dudz' (change in u wind in the x, y, and z directions), 'dvdx', 'dvdy', 'dvdz' (change in v wind in the x, y, and z directions), 'dwdx', 'dwdy', and 'dwdz' (change in w wind in the x, y, and z directions). The returned array is the same shape as the input arrays.

virt_temp(***kwargs*)

Compute virtual temperature.

Parameters ****kwargs** – Must contain two things. The first is either 't' (temperature in K) or 'pt' and 'p' (potential temperature in K and pressure in Pa). The second is either 'sph' (specific humidity in kg/kg) or 'qv' (water vapor mixing ratio in kg/kg). Any other keyword arguments are ignored.

Returns Virtual temperature in K.

virt_theta(***kwargs*)

Compute virtual potential temperature.

Parameters ****kwargs** – Must contain 'pt' (potential temperature in K) and either 'qv' or 'sph' (water vapor mixing ratio or specific humidity, either in kg/kg). Any other keyword arguments are ignored.

Returns Virtual potential temperature in K

vort_components(***kwargs*)

Compute all components of vorticity on a 3-dimensional grid.

Parameters ****kwargs** – Must contain 'u', 'u', 'w', 'x', 'y', and 'z'. u, v, and w are the zonal, meridional, and vertical components of wind, respectively, and must be 3-dimensional numpy arrays. The order of the axes is assumed to be (NZ, NY, NX). x and y should be 1-dimensional arrays containing the x and y coordinates for the domain, and z should be 3-dimensional arrays containing the vertical coordinates for the domain (e.g. 'zp' from an ARPS history file). All keywords that are not the aforementioned are ignored.

Returns The 3 components of vorticity as a numpy record array. The records are 'xvort' (vorticity in the x direction), 'yvort' (vorticity in the y direction), and 'zvort' (vertical vorticity). The returned array is the same shape as the input arrays.

zp_to_scalar(*zp*)

Converts ARPS physical height (zp) to height of scalar values (zps).

Parameters **zp** – ARPS physical height array (dimensions: (nz, ny, nx))

Returns zp_scalar: zp converted to the height of scalar values (similar to ARPS variable zps)

1.3 pycaps.derive.dropsizedist module

class GraupelPSD(*qgraupel*, *density=400.0*, *shape_param=0.0*, *intercept_param=None*, *num_concentration=None*)

Bases: `pycaps.derive.dropsizedist.PSD`

Represents one or more particle size distributions for graupel.

Parameters

- **qgraupel** (*np.array*) – Graupel mixing ratio (kg/kg)
- **density** (*np.array*) – Hydrometeor density (kg/m^3). Optional, default is 400 kg/m^3.
- **shape_param** (*np.array*) – Shape parameter for the PSD. Optional, default is 0.

- **intercept_param** (*np.array*) – Intercept parameter for the PSD in 1/m^4.
- **num_concentration** (*np.array*) – Total number concentration for the PSD in #/m^3.

reflectivityJZX (*density_air*, *rain_psd*, *max_mixed_frac*)
Compute the graupel PSD's contribution to linear reflectivity using the method of Jung, Zhang, and Xue (2008).

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **rain_psd** (PSD) – Rain PSD to compute water fractions with.
- **max_mixed_frac** (*float*) – Maximum mixed fraction for the graupel PSD.

Returns Graupel contribution to linear reflectivity

reflectivityZhang (*density_air*, *iswet*)
Compute the graupel PSD's contribution to linear reflectivity using the Zhang method.

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **is_wet** (*np.array*) – Array of integers (either 0 or 1) telling whether or not this region is above 0 C.

Returns Graupel contribution to linear reflectivity. Will be 0 for the Zhang method.

class HailPSD (*qhail*, *density=913.0*, *shape_param=0.0*, *intercept_param=None*, *num_concentration=None*)
Bases: *pycaps.derive.dropsizedist.PSD*

Represents one or more particle size distributions for hail.

Parameters

- **qhail** (*np.array*) – Hail mixing ratio (kg/kg)
- **density** (*np.array*) – Hydrometeor density (kg/m^3). Optional, default is 913 kg/m^3.
- **shape_param** (*np.array*) – Shape parameter for the PSD. Optional, default is 0.
- **intercept_param** (*np.array*) – Intercept parameter for the PSD in 1/m^4.
- **num_concentration** (*np.array*) – Total number concentration for the PSD in #/m^3.

reflectivityJZX (*density_air*, *rain_psd*, *max_mixed_frac*)
Compute the hail PSD's contribution to linear reflectivity using the method of Jung, Zhang, and Xue (2008).

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **rain_psd** (PSD) – Rain PSD to compute water fractions with.
- **max_mixed_frac** (*float*) – Maximum mixed fraction for the hail PSD.

Returns Hail contribution to linear reflectivity

reflectivityZhang (*density_air*, *is_wet*)
Compute the hail PSD's contribution to linear reflectivity using the Zhang method.

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)

- **is_wet** (*np.array*) – Array of integers (either 0 or 1) telling whether or not this region is above 0 C.

Returns Hail contribution to linear reflectivity

class PSD (*q_hydro*, *density_hydro*, *alpha_hydro=0.0*, *n0_hydro=None*, *nt_hydro=None*)

Bases: `object`

Represents one or more particle size distributions for any hydrometeor.

Parameters

- **q_hydro** (*np.array*) – Hydrometeor mixing ratio (kg/kg)
- **density_hydro** (*np.array*) – Hydrometeor density (kg/m^3)
- **alpha_hydro** (*np.array*) – Shape parameter for the PSD
- **n0_hydro** (*np.array*) – Intercept parameter for the PSD
- **nt_hydro** (*np.array*) – Total number concentration for the PSD

density

Hydrometeor density.

interceptParam (*density_air*)

Get the intercept parameter for the distribution.

Parameters **density_air** (*np.array*) – Density of the air (kg/m^3).

Returns The intercept parameter for the PSD.

mixingrat

Hydrometeor mixing ratio.

reflectivityJZX = <pycaps.util.util.abstract object>

reflectivityZhang = <pycaps.util.util.abstract object>

class PSDCollection (*rain=None*, *snow=None*, *hail=None*, *graupel=None*)

Bases: `object`

Uses several PSDs to compute total statistics for particles in a volume.

Parameters

- **rain** (PSD) – The rain PSD in the collection. Optional, default is no rain PSD.
- **snow** (PSD) – The rain PSD in the collection. Optional, default is no snow PSD.
- **hail** (PSD) – The rain PSD in the collection. Optional, default is no hail PSD.
- **graupel** (PSD) – The rain PSD in the collection. Optional, default is no graupel PSD.

reflectivity (**args*, ***kwargs*)

Computes the total logarithmic reflectivity for this PSD collection using different methods. The methods are described below.

Parameters

- ***args** – Differ based on which method you're using. When method='zhang', air temperature (K) and air density (kg/m^3) are required. For method='jzx', only air density (kg/m^3) is required.
- **method** (*string*) – The method to use when computing reflectivity. 'zhang' uses the Zhang reflectivity calculations, intended for single-moment microphysics with no graupel category. 'jzx' uses the method of Jung, Zhang, and Xue (2008).

Returns Logarithmic reflectivity (dBZ) computed for all PSDs in this collection.

class RainPSD (*qrain*, *density=1000.0*, *shape_param=0.0*, *intercept_param=None*, *num_concentration=None*)

Bases: `pycaps.derive.dropsizedist.PSD`

Represents one or more particle size distributions for rain.

Parameters

- **qrain** (*np.array*) – Rain mixing ratio (kg/kg)
- **density** (*np.array*) – Hydrometeor density (kg/m^3). Optional, default is 1000 kg/m^3.
- **shape_param** (*np.array*) – Shape parameter for the PSD. Optional, default is 0.
- **intercept_param** (*np.array*) – Intercept parameter for the PSD in 1/m^4.
- **num_concentration** (*np.array*) – Total number concentration for the PSD in #/m^3.

reflectivityJZX (*density_air*, **frozen_psds*)

Compute the rain PSD's contribution to linear reflectivity using the method of Jung, Zhang, and Xue (2008).

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **frozen_psds** (*list*) – List of tuples. Each tuple contains a frozen PSD to compute water fractions with and the maximum mixed fraction. May be called by reflectivityJZX(... *list_of_psds_and_mixed_fractions).

Returns Rain contribution to linear reflectivity

reflectivityZhang (*density_air*, *is_wet*)

Compute the rain PSD's contribution to linear reflectivity using the Zhang method.

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **is_wet** (*np.array*) – Array of integers (either 0 or 1) telling whether or not this region is above 0 C.

Returns Rain contribution to linear reflectivity

class SnowPSD (*qsnow*, *density=100.0*, *shape_param=0.0*, *intercept_param=None*, *num_concentration=None*)

Bases: `pycaps.derive.dropsizedist.PSD`

Represents one or more particle size distributions for snow.

Parameters

- **qsnow** (*np.array*) – Snow mixing ratio (kg/kg)
- **density** (*np.array*) – Hydrometeor density (kg/m^3). Optional, default is 100 kg/m^3.
- **shape_param** (*np.array*) – Shape parameter for the PSD. Optional, default is 0.
- **intercept_param** (*np.array*) – Intercept parameter for the PSD in 1/m^4.
- **num_concentration** (*np.array*) – Total number concentration for the PSD in #/m^3.

reflectivityJZX (*density_air*, *rain_psd*, *max_mixed_frac*)

Compute the snow PSD's contribution to linear reflectivity using the method of Jung, Zhang, and Xue (2008).

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **rain_psd** (PSD) – Rain PSD to compute water fractions with.
- **max_mixed_frac** (*float*) – Maximum mixed fraction for the snow PSD.

Returns Snow contribution to linear reflectivity

reflectivityZhang(*density_air*, *is_wet*)

Compute the snow PSD's contribution to linear reflectivity using the Zhang method.

Parameters

- **density_air** (*np.array*) – Density of the air (kg/m^3)
- **is_wet** (*np.array*) – Array of integers (either 0 or 1) telling whether or not this region is above 0 C.

Returns Snow contribution to linear reflectivity

1.4 pycaps.derive.iterative module

condensation_temp(*args*, *initial_args*)

Solve iteratively to find the temperature and pressure at which a parcel will condense if lifted adiabatically.

Parameters

- **args** (*list*) – A list containing temperature in K, pressure in kPa, and mixing ratio in g/g for this iteration.
- **initial_args** (*list*) – A list containing temperature in K, pressure in kPa, and mixing ratio in g/g from the initial iteration.

Returns The next guess at the temperature, pressure, and mixing ratio for the condensation temperature of a parcel lifted adiabatically.

iterative_main()

mixed_saturated_parcel(*args*, *initial_args*)

Solve iteratively to find the temperature and vapor pressure of a supersaturated parcel after condensation.

Parameters

- **args** (*list*) – A list containing temperature in K, pressure in Pa, and vapor pressure in Pa for this iteration.
- **initial_args** (*list*) – A list containing temperature in K, pressure in Pa, and vapor pressure in Pa from the initial iteration.

Returns The next guess at the temperature, pressure, and vapor pressure for a supersaturated parcel after condensation.

solve_iteratively(*func*, *args*, *first_guess=None*, *tolerance=1e-05*)

Solve a function iteratively.

Parameters

- **func** (*function*) – The function to solve. It should take two lists of arguments, the first for this iteration, the second from the initial iteration.
- **args** (*list*) – Initial set of arguments, representing the initial state of the parcel.

- **first_guess** (*list*) – A first guess at the solution. Optional, defaults to the initial state if not given.
- **tolerance** (*float*) – Tolerance on the change between successive solutions. Optional, defaults to 1e-5 if not given.

Returns An iterative solution to the provided function.

vapor_pressure (*temperature*, *rel_humidity*)

wet_bulb_temp (*args*, *initial_args*)

Solve iteratively to find the wet-bulb temperature for a parcel

Parameters

- **args** (*list*) – A list containing temperature in K, pressure in kPa, and mixing ratio in g/g for this iteration.
- **initial_args** (*list*) – A list containing temperature in K, pressure in kPa, and mixing ratio in g/g from the initial iteration.

Returns The next guess at temperature, pressure, and mixing ratio for the wet-bulb temperature of a parcel.

1.5 pycaps.derive.parcel module

compute_el (*parcel*, *snd*)

Computes EL given a parcel trace and environmental trace.

Parameters

- **parcel** (*tuple*) – A tuple of (parcel pressure in hPa, parcel temperature in degrees C). Both should be 1- dimensional arrays.
- **snd** (*tuple*) – A tuple of (environmental pressure in hPa, environmental temperature in degrees C). Both should be 1-dimensional arrays.

Returns The pressure of the parcel EL in hPa. In the case of multiple ELs, the highest one (lowest in pressure) is returned.

compute_lcl (*sfc_T*, *sfc_p*, *sfc_r=None*, *sfc_td=None*)

Compute the LCL pressure and temperature given parcel temperature, pressure, and mixing ratio or dewpoint.

Parameters

- **sfc_T** – Parcel temperature in degrees C
- **sfc_p** – Parcel pressure in hPa
- **sfc_r** – Parcel mixing ratio in g/g. Optional if sfc_td is specified.
- **sfc_td** – Parcel dewpoint in degrees C. Optional if sfc_r is specified.

Returns LCL temperature in degrees C and LCL pressure in hPa.

compute_lfc (*parcel*, *snd*)

Computes LFC given a parcel trace and environmental trace.

Parameters

- **parcel** (*tuple*) – A tuple of (parcel pressure in hPa, parcel temperature in degrees C). Both should be 1- dimensional arrays.

- **snd** (*tuple*) – A tuple of (environmental pressure in hPa, environmental temperature in degrees C). Both should be 1-dimensional arrays.

Returns The pressure of the parcel LFC in hPa. In the case of multiple LFCs, the highest one (lowest in pressure) is returned.

ml_parcel (*t_snd*, *p_snd*, *td_snd*, *ml_depth=100*)

Find the mixed-layer parcel given temperature, dewpoint, and pressure profiles.

Parameters

- **t_snd** (*np.ndarray*) – A 1-dimensional array of temperatures in degrees C.
- **p_snd** (*np.ndarray*) – A 1-dimensional array of pressures in hPa.
- **td_snd** (*np.ndarray*) – A 1-dimensional array of dewpoints in degrees C.
- **ml_depth** (*float*) – The depth of the mixed layer in hPa. Optional, defaults to 100 hPa.

Returns Temperature, in degrees C, pressure in hPa, and dewpoint in degrees C of the mixed-layer parcel.

mu_parcel (*t_snd*, *p_snd*, *td_snd*, *search_depth=300*)

Find the most-unstable parcel given temperature, dewpoint, and pressure profiles.

Parameters

- **t_snd** (*np.ndarray*) – A 1-dimensional array of temperatures in degrees C.
- **p_snd** (*np.ndarray*) – A 1-dimensional array of pressures in hPa.
- **td_snd** (*np.ndarray*) – A 1-dimensional array of dewpoints in degrees C.
- **search_depth** (*float*) – The depth of the layer to search in hPa. Optional, defaults to 300 hPa.

Returns Temperature, in degrees C, pressure in hPa, and dewpoint in degrees C of the most-unstable parcel.

sb_parcel (*t_snd*, *p_snd*, *td_snd*)

Find the surface-based parcel given temperature, dewpoint, and pressure profiles.

Parameters

- **t_snd** (*np.ndarray*) – A 1-dimensional array of temperatures in degrees C.
- **p_snd** (*np.ndarray*) – A 1-dimensional array of pressures in hPa.
- **td_snd** (*np.ndarray*) – A 1-dimensional array of dewpoints in degrees C.

Returns Temperature in degrees C, pressure in hPa, and dewpoint in degrees C of the surface-based parcel.

1.6 Module contents

CHAPTER

TWO

PYCAPS.DIAGNOSTIC PACKAGE

2.1 Submodules

2.2 pycaps.diagnostic.arps_gridinfo module

arps_gridinfo (*arpsfile*, *format='hdf'*, ***kwargs*)

Parameters

- **arpsfile** – the file for which you want grid information
- **format** – OPTIONAL – The format of your history file (valid choices are 'hdf' and 'netcdf')
- **grdbas** – OPTIONAL – A separate file (.netgrdbas or .hdfgrdbas) containing gridbase data, if needed.

Returns <<nothing>> (Prints grid information to terminal)

2.3 pycaps.diagnostic.listvars module

listvars (*source*, *shapefile=False*)

Gives a list of the variables, attributes, and dimensions contained in a NetCDF or HDF file. If called with shapefile = True, it will instead provide information on variables contained in a shapefile.

Parameters

- **source** – The file whose contents you wish to list (for best results, give full path)
- **shapefile** – OPTIONAL – If shapefile = True, listvars will attempt to open the source as a shapefile and give relevant information about the contents.

Returns <<nothing>> (prints information about file contents to the terminal window)

2.4 pycaps.diagnostic.patchinfo module

patchinfo (*format='hdf'*, ***kwargs*)

Given an ARPS history file or a specified domain size (nx by ny), returns information on valid patch configurations for MPI runs.

Parameters

- **format** – OPTIONAL – The format of your history file (valid choices are 'hdf' and 'netcdf')
- **file** – OPTIONAL – If you want to run patchinfo on an existing ARPS history dump file, the path to that file.
- **nx** – OPTIONAL – If you want to specify dimensions manually, # of x-gridpoints (remember ARPS adds 3 points to the edges!)
- **ny** – OPTIONAL – If you want to specify dimensions manually, # of y-gridpoints (remember ARPS adds 3 points to the edges!)
- **target** – OPTIONAL – A target number of processors; show detailed information on patch configurations near this target.
- **tolerance** – OPTIONAL – Given with a target, how close a configuration must be (in terms of # of procs) to be shown.

Returns <<nothing>> (Prints grid information to terminal)

2.5 pycaps.diagnostic.vardump module

var_dump (*var*, *source*, *filefmt='hdf'*)

Dumps the contents of a given variable, attribute, or dimension from a NetCDF or HDF file to the terminal window. Good for sanity checks or when you want to confirm a dimension or attribute is correctly set (var_info works well for this too.)

Parameters

- **var** – The name of the variable you want to dump.
- **source** – The HDF or NetCDF file containing the data you want to dump. For best results, provide a full path.
- **filefmt** – OPTIONAL – The format of the file you're reading from (default: hdf). Valid options are 'hdf' and 'netcdf'.

Returns <<nothing>> (dumps the requested information to the terminal window)

2.6 pycaps.diagnostic.varinfo module

var_info (*var*, *source*, *verbose=False*, *format='hdf'*)

Gives information on the dimensions, maximum, minimum, and average values of a variable in an HDF or NetCDF file. Good for spotting gross errors (e.g. NaN values, unrealistically high or low values). Verbose mode also includes information on NetCDF or HDF attritubes and dimensions.

Parameters

- **var** – the name of the variable to update (as stored in the file)
- **source** – the path to the file the data are stored in
- **verbose** – OPTIONAL – Passing verbose as True will give extra data on HDF/NetCDF variable attributes and dimensions. Default is False.
- **format** – OPTIONAL – The format your data are stored in (default: hdf). Valid options are 'hdf' and 'netcdf'.

Returns <<nothing>> (prints variable information to the terminal window)

2.7 Module contents

CHAPTER

THREE

PYCAPS.INTERP PACKAGE

3.1 Submodules

3.2 pycaps.interp.interp module

class **Interpolator**(*wrap*, *buffer*, *agl*)

Bases: `object`

Base class for the Interpolator objects. Needs to be re-implemented in a subclass.

Parameters

- **wrap** (*bool*) – What to do with points that are out of the domain. *wrap* == True means to assign the nearest point in the domain, *wrap* == False means to assign NaN to those points.
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).
- **agl** (*bool*) – Wether or not the heights are above ground level (*agl* == True) or above mean sea level (*agl* == False).

get_axes()

Return the axes used by this interpolator.

get_bounds = <pycaps.interp.interp.abstract object>

is_agl()

Return whether or not the heights in this interpolator refer to AGL (True) or MSL (False)

is_buffered()

Return whether or not the interpolator needs to keep a one-grid point buffer.

set_axes(*axes*)

Set the axes used by this interpolator.

Parameters **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional.

class **NullInterpolator**

Bases: *pycaps.interp.interp.Interpolator*

A null interpolator (that is: don't do any interpolation). This is useful for functions that expect an interpolator, but the user doesn't want to do any interpolation.

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated data as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

class PointInterpolator(*points*, *axes=None*, *coords='hght'*, *wrap=False*, *buffer=False*, *agl=False*)

Bases: `pycaps.interp.interp.Interpolator`

Interpolate to points.

Parameters

- **points** (*dict*) – A dictionary containing all the points to interpolate to. The dictionary must contain 'x', 'y', and 'z' keys pointing to numpy arrays or single floats containing the respective coordinates.
- **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional. Optional.
- **coords** (*str*) – String specifying which coordinate system to do the interpolation in. Valid values are 'hght' for height coordinates (default) or 'pres' for pressure coordinates. In the case of pressure coordinates, *axes['z']* is interpreted as pressure instead of height.
- **wrap** (*bool*) – What to do with points that are out of the domain. *wrap* == True means to assign the nearest point in the domain, *wrap* == False means to assign NaN to those points.
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).
- **agl** (*bool*) – Wether or not the heights are above ground level (*agl* == True) or above mean sea level (*agl* == False).

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

class SigmaInterpolator(*sigma*, *axes=None*, *buffer=False*)

Bases: `pycaps.interp.interp.Interpolator`

"Interpolate" to sigma levels in the model (that is: just pull out the relevant level, perhaps with a buffer).

Parameters

- **sigma** (*int*) – The model level to use.
- **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional. Optional.
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated data as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

class SoundingInterpolator(*point*, *axes=None*, *k_min=None*, *k_max=None*, *buffer=False*)

Bases: *pycaps.interp.interp.Interpolator*

Interpolate to single columns.

Parameters

- **point** (*dict*) – A dictionary containing the point to interpolate to. The dictionary must contain 'x' and 'y' keys pointing the respective coordinates in meters of the point.
- **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional. Optional.
- **k_min** (*int*) – Minimum sigma level to use in the cross section. Defaults to None (the bottom of the domain).
- **k_max** (*int*) – Maximum sigma level to use in the cross section. Defaults to None (the top of the domain).
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated data as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

class XSectInterpolator(*start_point*, *end_point*, *axes=None*, *k_min=None*, *k_max=None*, *buffer=False*)

Bases: `pycaps.interp.interp.Interpolator`

Interpolate to arbitrary vertical cross sections.

Parameters

- **start_point** (*dict*) – Dictionary containing the start point for the cross section. The dictionary should contain 'x' and 'y' keys pointing to the respective coordinates of the start point in meters.
- **end_point** (*dict*) – As in *start_point*, but for the end of the cross section. Because the cross section is given the same grid spacing as the input grid, the cross section may not end exactly on *end_point*. In that case, *end_point* will be between the last two columns on the cross section.
- **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional. Optional.
- **k_min** (*int*) – Minimum sigma level to use in the cross section. Defaults to None (the bottom of the domain).
- **k_max** (*int*) – Maximum sigma level to use in the cross section. Defaults to None (the top of the domain).
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated data as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

get_xsect_coords()

Get the x-y coordinates in meters of each column of the cross section.

Returns A tuple of 1D numpy arrays (xs, ys) giving the horizontal locations of the columns in the cross section in meters.

class ZInterpolator(*z_coord*, *axes=None*, *coords='hght'*, *wrap=False*, *buffer=False*, *agl=False*)

Bases: `pycaps.interp.interp.Interpolator`

Interpolate to specific heights (e.g. 500 m AGL, 6 km MSL) or pressures (e.g. 500 hPa).

Parameters

- **z_coord** (*float*) – The height or pressure at which to do the interpolation. Units must match the units of *axes['z']*.

- **axes** (*dict*) – A dictionary that must contain 'x', 'y', and 'z' keys corresponding to the x, y, and z axes. x and y must be one-dimensional, z must be three-dimensional. Optional.
- **coords** (*str*) – String specifying which coordinate system to do the interpolation in. Valid values are 'hght' for height coordinates (default) or 'pres' for pressure coordinates. In the case of pressure coordinates, *axes['z']* is interpreted as pressure instead of height.
- **wrap** (*bool*) – What to do with points that are out of the domain. *wrap* == True means to assign the nearest point in the domain, *wrap* == False means to assign NaN to those points.
- **buffer** (*bool*) – Wether to add a one grid point buffer around the data when it's loaded (usually used when you need a derivative across the plane of the data).
- **agl** (*bool*) – Wether or not the heights are above ground level (*agl* == True) or above mean sea level (*agl* == False).

__call__(*data*)

Do the interpolation.

Parameters **data** – The grid to interpolate. Note: this method assumes that you've already subsetted the grid according to the bounds returned by *getBounds()*. It also assumes that *data* is three-dimensional, even if one or more of those dimensions has length 1.

Returns Interpolated data as a Numpy array.

get_bounds()

Get the grid bounds of the data that needs to be loaded in from the data file. This tries to be as small as possible.

Returns tuple of slice objects (directly passable to numpy arrays or PyNIO variable objects) in Z, Y, X order.

Return type tuple

class abstract(*func*, *custom_name=None*)

Bases: `object`

interp_column(*data*, *axes*, *column*, *k_min=None*, *k_max=None*)

Interpolate 3-dimensional data to a column

Parameters

- **data** (*np.array*) – A 3-dimensional array (NZ x NY x NX) of data to interpolate
- **axes** (*dict*) – A dictionary containing the coordinate axes of the array. It must have 'x', 'y', and 'z' keys, whose values must be the x, y, and z coordinates, respectively, of each point in the grid. The x and y coordinate arrays must be 1-dimensional, and the z coordinate array must be 3-dimensional. Units are assumed to be meters.
- **column** (*dict*) – A dictionary containing the column to interpolate to. It must have 'x' and 'y' keys, whose values must be the x and y coordinates, respectively, of the column to interpolate to. Units are assumed to be meters.
- **k_min** (*int*) – Lowest sigma level to include in the column
- **k_max** (*int*) – Highest sigma level to include in the column

Returns A 1-dimensional array of interpolated data

interp_height(*data*, *z_coord*, *height*)

Interpolate 3-dimensional data to a given height

Parameters

- **data** (*np.array*) – A 3-dimensional array (NZ x NY x NX) of data to interpolate

- **z_coord** (*np.array*) – A 3-dimensional array containing the height of each point in the data grid. Units are assumed to be meters.
- **height** (*float*) – The height to interpolate to. Units are assumed to be meters.

Returns A 2-dimensional array of interpolated data.

interp_points (*data*, *axes*, *points*)

Interpolate 3-dimensional data to one or more points in space

Parameters

- **data** (*np.array*) – A 3-dimensional array (NZ x NY x NX) of data to interpolate
- **axes** (*dict*) – A dictionary containing the coordinate axes of the array. It must have 'x', 'y', and 'z' keys, whose values must be the x, y, and z coordinates, respectively, of each point in the grid. The x and y coordinate arrays must be 1-dimensional, and the z coordinate array must be 3-dimensional. Units are assumed to be meters.
- **points** (*dict*) – A dictionary containing the points to interpolate to. It must have 'x', 'y', and 'z' keys, whose values must be the x, y, and z coordinates, respectively, of the points to interpolate to. The coordinates can be either single floats or 1-dimensional arrays. Units are assumed to be meters.

Returns If the coordinates were specified as 1-dimensional arrays, returns a 1-dimensional array of the interpolated data. The order of this array is the same as the order of the points. Otherwise, returns a single float of interpolated data.

interp_xsect (*data*, *axes*, *start_point*, *end_point*, *k_min=None*, *k_max=None*)

Interpolate 3-dimensional data to a vertical cross section in an arbitrary direction. The interpolation creates a cross section grid that has the same horizontal grid interval as the source data. The user is not expected to provide an endpoint that occurs exactly at a grid interval. Rather, the cross section extends no more than 1 grid interval beyond the end point to provide an integer number of grid intervals in the interpolated data.

Parameters

- **data** (*np.array*) – A 3-dimensional array (NZ x NY x NX) of data to interpolate
- **axes** (*dict*) – A dictionary containing the coordinate axes of the array. It must have 'x', 'y', and 'z' keys, whose values must be the x, y, and z coordinates, respectively, of each point in the grid. The x and y coordinate arrays must be 1-dimensional, and the z coordinate array must be 3-dimensional. Units are assumed to be meters.
- **start_point** (*dict*) – A dictionary containing the coordinates of start column for the cross section. It must have 'x' and 'y' keys, whose values must be the x and y coordinates, respectively, of start column. Units are assumed to be meters.
- **end_point** (*dict*) – As in the start_point argument, but for the ending point. The cross section will not necessarily stop exactly at this column, but this column will be contained in the last grid interval.
- **k_min** (*int*) – Lowest sigma level to include in the cross section
- **k_max** (*int*) – Highest sigma level to include in the cross section

Returns A tuple containing the 2-dimensional grid of interpolated data, as well as the x and y coordinates of the cross section in meters.

3.3 pycaps.interp.setup_subdomain module

setup_subdomain(*xmin*, *xmax*, *ymin*, *ymax*, *dx*, *dy*, *trulat1*, *trulat2*, *var2d*, *fullmap*)

Takes data from a larger domain and maps it onto a smaller subdomain.

Parameters

- **xmin** – i-coordinate of the west boundary of the subdomain
- **ymin** – j-coordinate of the south boundary of the subdomain
- **xmax** – i-coordinate of the east boundary of the subdomain
- **ymax** – j-coordinate of the north boundary of the subdomain
- **dx** – the horizontal grid spacing in the east-west direction, in meters
- **dy** – the horizontal grid spacing in the north-south direction, in meters
- **trulat1** – the first trulat value of the Lambert conformal map projection (often 30.0)
- **trulat2** – the second trulat value of the Lambert conformal map projection (often 60.0)
- **var2d** – the variable you need to map to a subdomain (as a 2D x-y slice)
- **fullmap** – the Basemap object associated with the larger domain (within which you are putting the subdomain)

Returns a Basemap object for the new subdomain x: a 2D array containing the x-coordinate, suitable for plotting with matplotlib y: a 2D array containing the y-coordinate, suitable for plotting with matplotlib var2d: the 2D variable specified in the input, mapped to the subdomain

Return type map

3.4 Module contents

CHAPTER

FOUR

PYCAPS.IO PACKAGE

4.1 Submodules

4.2 pycaps.io.MRMSGrid module

class **MRMSGrid**(*start_date*, *end_date*, *variable*, *path_start*, *freq='1H'*)

Bases: `object`

MRMSGrid reads time series of MRMS grib2 files, interpolates them, and outputs them to netCDF4 format.

interpolate_grid(*in_lon*, *in_lat*)

Interpolates MRMS data to a different grid using cubic bivariate splines

interpolate_to_netcdf(*in_lon*, *in_lat*, *out_path*, *date_unit='seconds since 1970-01-01T00:00'*)

Calls the interpolation function and then saves the MRMS data to a netCDF file. It will also create separate directories for each variable if they are not already available.

load_data()

Loads data from MRMS GRIB2 files and handles compression duties if files are compressed.

MRMS_main()

interpolate_mrms_day(*start_date*, *variable*, *mrms_path*, *map_filename*, *out_path*)

For a given day, this module interpolates hourly MRMS data to a specified latitude and longitude grid, and saves the interpolated grids to CF-compliant netCDF4 files.

Parameters

- **start_date** (*datetime.datetime*) – Date of data being interpolated
- **variable** (*str*) – MRMS variable
- **mrms_path** (*str*) – Path to top-level directory of MRMS GRIB2 files
- **map_filename** (*str*) – Name of the map filename. Supports ARPS map file format and netCDF files containing latitude and longitude variables
- **out_path** (*str*) – Path to location where interpolated netCDF4 files are saved.

load_map_coordinates(*map_file*)

Loads map coordinates from netCDF or pickle file created by util.makeMapGrids.

Parameters **map_file** – Filename for the file containing coordinate information.

Returns Latitude and longitude grids as numpy arrays.

4.3 pycaps.io.ModelGrid module

class **ModelGrid**(*filenames*, *run_date*, *start_date*, *end_date*, *variable*, *frequency='1H'*)

Bases: `object`

Base class for reading 2D model output grids from netCDF files.

Given a list of file names, loads the values of a single variable from a model run. Supports model output in netCDF format.

filenames
list of str

List of netCDF files containing model output

run_date
ISO date string or datetime.datetime object

Date of the initialization time of the model run.

start_date
ISO date string or datetime.datetime object

Date of the first timestep extracted.

end_date
ISO date string or datetime.datetime object

Date of the last timestep extracted.

freqency
str

spacing between model time steps.

valid_dates
DatetimeIndex of all model timesteps

forecast_hours
array of all hours in the forecast

file_objects
list

List of the file objects for each model time step

__enter__()
Open each file for reading.

__exit__()
Close links to all open file objects and delete the objects.

close()
Close links to all open file objects and delete the objects.

static **format_var_name**(*variable*, *var_list*)
Searches var list for variable name, checks other variable name format options.

Parameters

- **variable** (*str*) – Variable being loaded
- **var_list** (*list*) – List of variables in file.

Returns Name of variable in file containing relevant data, and index of variable z-level if multiple variables contained in same array in file.

load_data()

Load data from netCDF file objects or list of netCDF file objects. Handles special variable name formats.

Returns Array of data loaded from files in (time, y, x) dimensions, Units

load_data_old()

Loads time series of 2D data grids from each opened file. The code handles loading a full time series from one file or individual time steps from multiple files. Missing files are supported.

4.4 pycaps.io.NCARModelGrid module

class NCARModelGrid(*member*, *run_date*, *variable*, *start_date*, *end_date*, *path*, *single_step=False*)

Bases: `pycaps.io.ModelGrid.ModelGrid`

Extension of the ModelGrid class for interfacing with the NCAR ensemble.

Parameters

- **member** (*str*) – Name of the ensemble member
- **run_date** (*datetime.datetime object*) – Date of the initial step of the ensemble run
- **start_date** (*datetime.datetime object*) – First time step extracted.
- **end_date** (*datetime.datetime object*) – Last time step extracted.
- **path** (*str*) – Path to model output files.
- **single_step** (*boolean (default=False*) – Whether variable information is stored with each time step in a separate file or one file containing all timesteps.

4.5 pycaps.io.SSEFModelGrid module

class SSEFModelGrid(*member*, *run_date*, *variable*, *start_date*, *end_date*, *path*, *single_step=False*)

Bases: `pycaps.io.ModelGrid.ModelGrid`

Extension of ModelGrid to the CAPS Storm-Scale Ensemble Forecast system.

Parameters

- **member** (*str*) – Name of the ensemble member
- **run_date** (*datetime.datetime object*) – Date of the initial step of the ensemble run
- **start_date** (*datetime.datetime object*) – First time step extracted.
- **end_date** (*datetime.datetime object*) – Last time step extracted.
- **path** (*str*) – Path to model output files.
- **single_step** (*boolean (default=False*) – Whether variable information is stored with each time step in a separate file or one file containing all timesteps.

4.6 pycaps.io.binfile module

class BinFile (*file_name*, *mode='r'*, *byteorder='<'*)

Bases: `object`

Handles the low-level reading and writing of binary files. Inherit from this class for all your binary file I/O needs.

Parameters

- **file_name** (*str*) – Name of the file to open.
- **mode** (*str*) – Read/write mode for the file. Default is 'r' (for reading).
- **byteorder** (*str*) – Endianness for the file. Acceptable values are '<' for little-endian, '>' for big endian. Default is '<' (little endian).

ANCH_CURPOS = 1

ANCH_FILEBEG = 0

ANCH_FILEEND = 2

_ateof ()

Returns True if the pointer has reached the end of the file, returns False otherwise.

_compute_block_size (*type_dict*)

_peek (*type_string*)

Peek at a value in the file.

Parameters **type_string** (*str*) – See *_read()*.

Returns Data from the file (see *_read()*).

_read (*type_string*, *_peeking=False*)

Read values from the file and return as a specific type.

Parameters **type_string** (*str*) – The number and type of data to read from the file. The form of the string is *nt*, where *n* is the number of values to read, and *t* is the type of the values. If *n* is omitted, it is assumed to be 1. Acceptable values for *t* can be found in the table below.

Returns Data from the file as the specified type. For example, a type string of '4f' will return a list of 4 floats. A type string of 'i' will return a single integer.

Some possible values for the type character are as follows:

Type Character	Meaning
i	signed 32-bit integer
h	signed 16-bit integer
b	signed 8-bit integer
f	single-precision float
d	double-precision float
c	single character
s	character string

The *s* character may be post-fixed with a number that tells the length of the string. For example, *s10* refers to a string of length 10. See https://docs.python.org/2/library/struct.html#format-characters for a full description.

_read_block (*type_dict*, *dest_dict*, *fortran=False*)

Read a block of data from the file.

Parameters

- **type_dict** (*OrderedDict*) – An ordered dictionary with variable names as keys and type strings (see *_read()*) as values.
- **dest_dict** (*dict*) – A dictionary in which to place the values when they've been read from the file.
- **fortran** (*bool*) – Whether or not to read the block as a Fortran-formmated block. Default is false.

_read_grid(*type_string*, *shape*, *fortran=False*)

Read a grid from the file.

Parameters

- **type_string** (*str*) – As in :py:meth'_read()', but only the type character. The number is determined by the *shape* argument.
- **shape** (*tuple*) – The shape of the grid to return.
- **fortran** (*bool*) – Whether or not to write the grid as a Fortran-formatted block. Default is False.

Returns A grid of data as a numpy array.

_seek(*location*, *anchor=0*)

Move the file pointer to a particular location.

Parameters

- **location** (*int*) – Location in the file in number of bytes.
- **anchor** (*int*) – The point in the file to which *location* is relative. For example, *BinFile.ANCH_FILEBEG* means that *location* is relative to the beginning of the file. Default value is *BinFile.ANCH_FILEBEG*.

_tell()

Get the current location of the file pointer in bytes from the start of the file.

_write(*value*, *type_string*)

Write values to the file.

Parameters **type_string** (*str*) – See *_read()* for a full description.

_write_block(*type_dict*, *src_dict*, *fortran=False*)

Write a block of data from the file.

Parameters

- **type_dict** (*OrderedDict*) – An ordered dictionary with variable names as keys and type strings (see *_read()*) as values.
- **src_dict** (*dict*) – A dictionary containing the variable names as keys and their values.
- **fortran** (*bool*) – Whether or not to read the block as a Fortran-formmated block. Default is false.

_write_grid(*type_string*, *grid*, *fortran=False*)

Write a grid to the file.

Parameters

- **type_string** (*str*) – As in :py:meth'_read()', but only the type character. The number is determined by the *shape* argument.
- **shape** (*tuple*) – The shape of the grid to return.

- **fortran** (*bool*) – Whether or not to write the grid as a Fortran-formatted block. Default is False.

close()
Close the file

4.7 pycaps.io.coltilt module

class ColumnTiltFile(*file_name*, *vars=('vr', 'Z')*, *mode='r'*)
Bases: *pycaps.io.binfile.BinFile*

Read an ARPS EnKF column-tilt formatted radar observation file.

Parameters

- **file_name** (*str*) – The name of the file to open.
- **variables** (*list*) – A list containing the names and order of the variables in the file. Defaults to ['vr', 'Z'] (radial velocity, then reflectivity).
- **mode** (*str*) – Read/write mode for this file. The default is 'r' for reading, currently the only supported option.

radar_id
str

4-character radar ID.

timestamp
datetime

The valid time for the data file.

elevations
np.array

List of elevation angles in the file.

__getitem__(*var_name*)
Retrieve data from the file.

Parameters **var_name** (*str*) – Name of the variable to retrieve. Acceptable values are 'z' (height), 'r' (slant range), or any of the names passed to the *vars* keyword argument in `ColumnTiltFile.__init__()`.

Returns A three-dimensional numpy array (NTILT × NY × NX)

Examples

```
>>> ctf = ColumnTiltFile("/path/to/columntiltfile/KTLX.20110524.210000")
>>> ctf['Z'].shape # Print the shape of the reflectivity array. Will be the same horizontal
(14, 303, 303)
>>> ctf['Z'].max() # Print the maximum of the reflectivity array
68.41645
```

4.8 pycaps.io.dataload module

get_axes (*base_path*, *base_name*, *agl=True*, *z_coord_type=''*, *split=None*, *fcst=False*)

Get the axes from a grdbas file.

Parameters

- **base_path** (*str*) – Path to the grdbas file.
- **base_name** (*str*) – Base name of the grdbas file (e.g. 'enf001').
- **agl** (*bool*) – Whether to return the z coordinates relative to ground level (True) or MSL (False)
- **z_coord_type** (*str*) – Which set of z coordinates to use. "" refers to the atmospheric coordinates, and "soil" refers to the soil z coordinates. Default is "" (atmospheric coordinates).
- **split** (*tuple*) – If the domain is split, then this is tuple contains the domain decomposition in (NX, NY).

Returns A dictionary containing the x and y coordinates in the 'x' and 'y' keys, and either 'z' and 'z_MSL' (if agl=True) or 'z' and 'z_AGL' (if agl=False)

load_domain (*base_path*, *data_file*, *derived*, *interpolator*, *coords='hght'*, *aggregator=None*, *split=None*)

Load one timestep of one ensemble member.

load_ensemble (*base_path*, *members*, *times*, *var_names*, *derived=<function recarray_fm_dict>*, *interpolator=<pycaps.interp.interp.NullInterpolator object>*, *aggregator=None*, *max_concurrent=-1*, *single_var=False*, *z_coord_type='atmos'*, *coords='hght'*, *fcst=False*, *split=None*)

Load an entire ensemble into memory. Optionally, do interpolation, compute derived variables, and aggregate the ensemble.

Parameters

- **base_path** (*str*) – Path to the data to load.
- **members** (*int or list*) – If an integer, determines the number of members to load (e.g. load members 1 through *members*). If a list, determines which members to load (e.g. load members 1, 4, 10, and 19).
- **times** (*list*) – A list of times (in seconds since initialization) at which to load data.
- **var_names** (*list*) – A list of variable names to load from the file.
- **derived** (*function*) – A function to compute derived variables. The function must take a collection of keyword arguments (e.g. **kwargs) and return a single numpy array. Optional, default is to return all the variables in a numpy record array.
- **interpolator** (Interpolator) – An interpolator object (as defined in pycaps.interp.interp) specifying how to interpolate each member at each time. Optional, default is to do no interpolation (return the full three- dimensional domain)
- **aggregator** (*function*) – A function describing how to aggregate the ensemble members *prior* to computing derived variables. The function must take a single numpy array, the first dimension of which is the ensemble, and return another numpy array. Could be used for computing an ensemble mean prior to computing reflectivity. Optional, default is to do no aggreggation.

- **max_concurrent** (*int*) – The number of processes to run concurrently. Optional, default is to load all ensemble members at the same time (or in some configurations, all time steps from an individual member at the same time).
- **single_var** (*bool*) – Whether or not to load data from a single variable file. If true, then var_names must be of length 1. Optional, default is False.
- **z_coord_type** (*str*) – Specifies which z coordinates to use. Acceptable values are "atmos" for atmospheric z coordinates or "soil" for soil z coordinates. Optional, default is "atmos".
- **coords** (*str*) – Specifies which vertical coordinate to use in the atmosphere. Acceptable values are "hght" for for height coordinates, and "pres" for pressure coordinates. Optional, default is "hght".
- **fcst** (*bool*) – Specifies whether to load the forecast (True) or analysis (False) ensemble. Optional, default is False (loads analysis ensemble).
- **split** (*tuple*) – Specifies the domain configuration for split ensembles. Must be a tuple (NPX, NPY), where NPX is the number of subdomains in the x direction, and NPY is the same for the y direction. Optional, default is an already-joined ensemble.

Returns A numpy array containing the data in the ensemble. For the full ensemble with no interpolation or aggregation, the order of dimensions will be (NE, NT, NZ, NY, NX). Aggregation will remove the NE dimension, and interpolation will change the last three according to which interpolation is being done (for example, interpolation to a height will remove the NZ dimension, while interpolating to a set of points will replace the NZ, NY, and NX dimensions with NP).

load_run(*base_path*, *base_name*, *times*, *var_names*, *derived=<function recarray_fm_dict>*, *interpolator=<pycaps.interp.interp.NullInterpolator object>*, *max_concurrent=-1*, *single_var=False*, *z_coord_type='atmos'*, *coords='hght'*, *split=None*)

Load a single run into memory. Optionally, do interpolation and compute derived variables.

Parameters

- **base_path** (*str*) – Path to the data to load.
- **base_name** (*str*) – The
- **times** (*list*) – A list of times (in seconds since initialization) at which to load data.
- **var_names** (*list*) – A list of variable names to load from the file.
- **derived** (*function*) – A function to compute derived variables. The function must take a collection of keyword arguments (e.g. **kwargs) and return a single numpy array. Optional, default is to return all the variables in a numpy record array.
- **interpolator** (Interpolator) – An interpolator object (as defined in pycaps.interp.interp) specifying how to interpolate each member at each time. Optional, default is to do no interpolation (return the full three- dimensional domain)
- **max_concurrent** (*int*) – The number of processes to run concurrently. Optional, default is to load all time steps at the same time.
- **single_var** (*bool*) – Whether or not to load data from a single variable file. If true, then var_names must be of length 1. Optional, default is False.
- **z_coord_type** (*str*) – Specifies which z coordinates to use. Acceptable values are "atmos" for atmospheric z coordinates or "soil" for soil z coordinates. Optional, default is "atmos".

- **coords** (*str*) – Specifies which vertical coordinate to use in the atmosphere. Acceptable values are "hght" for for height coordinates, and "pres" for pressure coordinates. Optional, default is "hght".
- **split** (*tuple*) – Specifies the domain configuration for split ensembles. Must be a tuple (NPX, NPY), where NPX is the number of subdomains in the x direction, and NPY is the same for the y direction. Optional, default is an already-joined ensemble.

Returns A numpy array containing the data for the run. For the full run with no interpolation, the order of dimensions will be (NT, NZ, NY, NX). Iinterpolation will change the last three according to which interpolation is being done (for example, interpolation to a height will remove the NZ dimension, while interpolating to a set of points will replace the NZ, NY, and NX dimensions with NP).

4.9 pycaps.io.gridtilt module

class GridTiltFile (*file_name*, *vars=('vr', 'Z')*, *mode='r'*)

Bases: *pycaps.io.binfile.BinFile*

Read an ARPS EnKF grid-tilt formatted radar observation file.

Parameters

- **file_name** (*str*) – The name of the file to open.
- **vars** (*list*) – A list containing the names and order of the variables in the file. Defaults to ['vr', 'Z'] (radial velocity, then reflectivity).
- **mode** (*str*) – Read/write mode for this file. The default is 'r' for reading, 'w' for writing is also supported.

timestamp

datetime

The valid time for the data file.

n_tilts

int

Number of elevations in the radar data.

n_gridx

int

Number of grid points in the x direction.

n_gridy

int

Number of grid points in the y direction.

radar_name

str

Name of the radar.

radar_lat

float

Latitude of the radar.

radar_lon
float

Longitude of the radar.

radar_x
float

x coordinate of the radar location on the domain.

radar_y
float

y coordinate of the radar location on the domain.

d_azimuth
float

Azimuthal spacing for the raw radar data.

range_min
float

Minimum range for the raw radar data.

range_max
float

Maximum range for the raw radar data.

elevations
np.array

List of elevation angles.

__getitem__(*var_name*)
Retrieve data from the file.

Parameters **var_name** (*str*) – Name of the variable to retrieve. Acceptable values are 'z' (height in meters), 'r' (slant range in meters), or any of the names passed to the *vars* keyword argument in `GridTiltFile.__init__()`.

Returns A three-dimensional numpy array (NTILT × NY × NX)

Examples

```
>>> gtf = GridTiltFile("/path/to/gridtiltfile/KTLX.20110524.210000")
>>> gtf['Z'].max() # Pull the reflectivity out of the file and take the maximum.
68.41645
```

__setitem__(*var_name*, *data*)
Set data in the file.

Parameters

- **var_name** (*str*) – Name of the variable to set. Use 'z' for height and 'r' for slant range.
- **data** (*np.array*) – The data put into the file. Must be the same shape as the rest of the variables.

Examples

```
>>> gtf = GridTiltFile("/path/to/gridtiltfile/KTLX.20110524.210000")
>>> gtf['Z'] = new_reflectivity_data
```

close()
Close the file. Write data if opened for writing.

copy_headers(*gtf*)
Copy the headers from another gridtilt file to this one.

Parameters **gtf** (*GridTileFile*) – The grid tilt file from which to take the header information.

4.10 pycaps.io.io_modules module

grdbas_read(*grdbas_filename*, *format='hdf'*)
Reads in basic grid data from an ARPS grdbas file (in HDF format)

Parameters

- **grdbas_filename** – The full path to the grdbas file to read from
- **format** – OPTIONAL – The format of your input data (currently valid options are 'hdf' (default) and 'netcdf')

Returns The latitude of the domain center ctrlon: The longitude of the domain center trulat1: The first true latitude value for the lambert conformal map projection trulat2: The second true latitude value for the lambert conformal map projection trulon: The true longitude value for the lambert conformal map projection nx: The number of gridpoints in the east-west direction ny: The number of gridpoints in the north-south direction nz: The number of gridpoints in the vertical direction dx: The grid spacing in the east-west direction dy: The grid spacing in the north-south direction width_x: The width of the domain in the east-west direction, in meters width_y: The width of the domain in the north-south direction, in meters

Return type ctrlat

grdbas_read_patch(*grdbas_filename*, *xpatches*, *ypatches*, *format='hdf'*)
Reads in basic grid data from ARPS grdbas patch files (in HDF format)

Parameters

- **grdbas_filename** – The full path to the grdbas file to read from
- **xpatches** – The number of patches in the x-direction (in arps.input, nproc_x)
- **ypatches** – The number of patches in the y-direction (in arps.input, nproc_y)
- **format** – OPTIONAL – The format of your file. Valid options include netcdf and HDF (default HDF).

Returns The latitude of the domain center ctrlon: The longitude of the domain center trulat1: The first true latitude value for the lambert conformal map projection trulat2: The second true latitude value for the lambert conformal map projection trulon: The true longitude value for the lambert conformal map projection nx: The number of gridpoints in the east-west direction in the full domain ny: The number of gridpoints in the north-south direction in the full domain nz: The number of gridpoints in the vertical direction in the full domain dx: The grid spacing in the east-west direction dy: The grid spacing in the north-south direction width_x: The width of the domain in the east-west direction, in meters width_y: The width of the domain in the north-south

direction, in meters nx_patch: The number of gridpoints in the east-west direction in a single patch ny_patch: The number of gridpoints in the north-south direction in a single patch

Return type ctrlat

read_xy_slice (*field*, *source*, *level*, *format='hdf'*, ***kwargs*)

Reads the data needed for xyplot to generate an x-y variable field plot.

Parameters

- **field** – The variable name associated with the field to be plotted (e.g. u, v, pt, qr)
- **source** – The FULL PATH to the file containing the data to be plotted.
- **level** – The vertical (k) model layer for which data should be read and plotted.
- **format** – OPTIONAL – The input data format (currently valid options are 'hdf' (default) and 'netcdf')
- **grdbas** – OPTIONAL – A history file to read grid information from (if different from source)
- **h2** – OPTIONAL – A second source file containing other data needed for plotting, if you have one. (example: Read reflectivity from one file, and wind data from another).
- **truref** – OPTIONAL – If reading from an ARPS file processed using ossedata (affects variable name and filename) this should be set to the full path to the truref file.
- **decompress** – OPTIONAL – A flag to set to True if you're using ARPS data with 32 bit integers mapped to 16 bit integers. Will call the decompression function if set to True.

Returns The 2D slice of data to be plotted

Return type var_sfc

4.11 pycaps.io.level2 module

class NCDCLevel2File (*file_name*, *mode='r'*, *byteorder='>'*)

Bases: *pycaps.io.binfile.BinFile*

Read a raw Level II file downloaded from NCDC.

Parameters

- **file_name** (*str*) – The name of the file to load.
- **mode** (*str*) – Read/write mode of the file. Default value is 'r', the only currently supported option.

__getitem__ (*var_name*)

Retrieve data from the file.

Parameters **var_name** (*str*) – Name of the variable to retrieve from the file. Acceptable values are 'REF' for reflectivity and 'VEL' for radial velocity.

Returns A three-dimensional numpy array (NTILT × NAZIM × NRANGE)

get_coords (*var_name*, *sweep_no*)

Get the azimuth and range for a sweep in the data file.

Parameters

- **var_name** (*str*) – The variable name for which to retrieve the azimuth and range.

- **sweep_no** (*int*) – The tilt for which to retrieve azimuth and range.

Returns A tuple of 1-dimensional numpy arrays of azimuth (in degrees) and range (in meters).

get_elevations(*var_name*)

Get the elevation angles for a particular variable.

Parameters **var_name** (*str*) – The variable name for which to retrieve the elevation angles.

Returns A 1-dimensional numpy array containing the elevation angles in degrees.

4.12 pycaps.io.modelobs module

class ARPSModelObsFile(*file_name*, *vars=('vr', 'Z')*, *mpi_config=(1, 1)*, *mode='r'*)

Bases: *pycaps.io.binfile.BinFile*

Read an ARPS EnKF model observation file (such as that created by arpsenkf or postinnov).

Parameters

- **file_name** (*str*) – The name of the file to open.
- **vars** (*list*) – A list containing the names and order of the variables in the file. Defaults to ['vr', 'Z'] (radial velocity, then reflectivity).
- **mpi_config** (*tuple*) – A tuple containing the MPI configuration of the run that generated this file. Defaults to (1, 1), signifying no MPI.
- **mode** (*str*) – Read/write mode for this file. The default is 'r' for reading, currently the only supported option.

timestamp

datetime

The valid time for the data in the file.

n_tilts

int

Number of elevation angles in the file.

n_gridx

int

Number of grid points in the x direction.

n_gridy

int

Number of grid points in the y direction.

radar_id

str

4-character ID for the radar.

radar_lat

float

Latitude of the radar.

radar_lon

float

Longitude of the radar.

radar_x
float

x coordinate of the radar on the domain.

radar_y
float

y coordinate of the radar on the domain.

d_azim
float

Azimuthal spacing of the raw data.

range_min
float

Minimum range of the raw data.

range_max
float

Maximum range of the raw data.

elevations
np.array

List of elevation angles in the file.

__getitem__(*var_name*)
Retrieve data from the file.

Parameters **var_name** (*str*) – Name of the variable to retrieve. Acceptable values are 'z' (height), 'r' (slant range), or any of the names passed to the *vars* keyword argument in `ARPSModelObsFile.__init__()`.

Returns A three-dimensional numpy array (NTILT × NY × NX)

4.13 Module contents

CHAPTER

FIVE

PYCAPS.PLOT PACKAGE

5.1 Submodules

5.2 pycaps.plot.colormap_setup module

from_levels_and_colors(*levels*, *colors*, *extend='neither'*)

A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments. The contents of this code are identical to the from_levels_and_colors in more recent versions of matplotlib (which are not available in the python2.7 installation on the CAPS servers as of 21 Oct. 2015)

Parameters

- **levels** – sequence of numbers – The quantization levels used to construct the `BoundaryNorm`. Values `v` are quantizized to level `i` if `lev[i] <= v < lev[i+1]`.
- **colors** – sequence of colors – The fill color to use for each level. If *extend* is "neither" there must be `n_level - 1` colors. For an *extend* of "min" or "max" add one extra color, and for an *extend* of "both" add two colors.
- **extend** – {'neither', 'min', 'max', 'both'}, optional. The behaviour when a value falls out of range of the given levels. See `contourf()` for details.

Returns tuple containing a `Colormap` and a `Normalize` instance

Return type (cmap, norm)

thresh_setup(*var*, *max=100.0*, *min=-100.0*, *tornado=False*)

A function to define default thresholds and colorbar for plotting various fields.

For unknown fields, passing in a max and min value will generate a generic 20-gradation colorbar. If max and min data are missing, -100 to 100 will be assumed as a last resort.

NOTE: "max" and "min" are only used when "var" is not found in the presets.

Parameters

- **var** – the name of the variable, passed as a string
- **max** – OPTIONAL – the maximum value you'd want to plot (default: -100.0)
- **min** – OPTIONAL – the minimum value you'd want to plot (default: 100.0)
- **tornado** – OPTIONAL – A flag to set True if you're plotting fields for a tornado and thus need to use otherwise extreme values for some fields (default: False)

Returns a list of threshold values for contouring using contourf colormap: the name of a colormap to use from ns_colormaps.py cb_ticks: which ticks to show in the colorbar for this field no_proportional: A flag to set to select equal spacing in the colorbar

Return type thresholds

5.3 pycaps.plot.nexrad_color_tables module

remap(*x*)

5.4 pycaps.plot.ns_colormap module

5.5 pycaps.plot.plot_arps_xy module

xyplot(*data*, *field*, *source*, *level*, *format='hdf'*, ***kwargs*)

Plots a single HDF or NetCDF variable field (or field derived from history data) in the x-y plane, with options for various overlays. Grid data, such as that stored in an ARPS grdbas file, are required. Lambert conformal map projection will be used.

Parameters

- **data** – The 2D x-y slice to be plotted (e.g. u, v, pt, qr). Can be generated quickly using read_xy_slice.
- **field** – The variable name (e.g. 'u', 'v', 'pt', 'qr'). Needed to properly format your plot. Enclose it in single quotes.
- **source** – The FULL PATH to the file containing the data for plotting.
- **level** – The vertical (k) model layer on which to plot an x-y slice.
- **format** – OPTIONAL – The file format for your input (currently, valid choices are "hdf" (default) and "netcdf")
- ****kwargs** – grdbas: The history file from which to read grid information (if different from 'source') h2: A second source file containing other data needed for plotting, if you have one. (example: Read reflectivity from one file, and wind data from another). name: A name to identify the data (may be used in plot title and output filename) truref: If reading from an ARPS file processed using ossedata (affects variable name and filename) this should be set to the full path to the truref file. addshape: Specify a shapefile to overlay data from. Requires the shapefile (and options) to be included in the shapefiles block in plot_arps_xy.py within the pycaps package. colormap: Used to specify a colormap from ns_colormap.py for use with this plot. If not provided, a default colormap choice appropriate to the variable will be used. pixels: A flag to set to True if you want a pixel (colormesh) plot instead of a contour plot. tornado: A flag to set to True if you are plotting values from a high-resolution tornadic dataset. Will choose appropriate values for relevant variables (e.g. wspd, p) imin: The i-coordinate of the west edge of your plot. If not given, default = 0 imax: The i-coordinate of the east edge of your plot. If not given, default = nx jmin: The j-coordinate of the south edge of your plot. If not given, default = 0 jmax: The j-coordinate of the north edge of your plot. If not given, default = ny qcbelow: If given, values below this threshold will be marked as missing/bad data. qcabove: If given, values above this threshold will be marked as missing/bad data. urban: A flag to set to True if you want to plot urban data (if not given, False by default). counties: A flag to set to False if you don't want to plot counties (if not given, True by default). states: A flag to set to False if you don't want to plot states (if not given, True by default). coastlines: A flag to set to False if you don't want to plot coastlines (if not given, True by default). lat: If given, will plot lines of latitude at the given interval (in degrees). lon: If given, will plot lines of longitude at the given interval (in degrees).

decompress: A flag to set to True if you're using ARPS data with 32 bit integers mapped to 16 bit integers. Will call the decompression function if set to True.

Returns <<nothing>> (Generates and saves the requested plot to a .png file)

5.6 pycaps.plot.pubfig module

get_multiplier(*fig=None*)

Get a multipler for the size of text objects, a function of the diagonal length on the figure.

Parameters **fig** (*matplotlib.figure.Figure*) – The figure to use. Defaults to none, meaning to use the current figure.

Returns The size multiplier for the figure.

get_subplot_position(*gs*)

Get a subplot's position in the figure layout.

Parameters **gs** (*gridspec*) – A matplotlib gridspec object, perhaps obtained from the *gs_parent* keyword argument to your subpanel function.

Returns A tuple containing the row and column of the subplot. Rows and columns are zero-based.

publication_figure(*subfigures*, *layout*, *corner='ul'*, *colorbar=None*, *hanging='bottom'*, *dpi=300*)

Make publication figures out of many panels, automatically adding the subfigure letter and generating a figure colorbar.

Parameters

- **subfigures** (*list*) – A list of functions, each of which plots a panel of the figure. Each function should take **kwargs as arguments.
- **layout** (*tuple*) – A tuple specifying how the panels are arranged (e.g. (2, 4) for a figure with 2 rows and 4 columns)
- **corner** (*str*) – The corner in which to put the figure letter. Acceptable values are 'ul' for the upper-left corner, 'lr' for the lower-right corner, etc.
- **colorbar** (*dict*) – A dictionary specifying how to create the figure colorbar.
- **hanging** (*str*) – If the number of panels does not fit into the layout (e.g. 3 panels in a 2 $\times$ 2 layout), this specifies where to have the incomplete row/column. Acceptable values are 'left', 'bottom', 'right', and 'top'.
- **dpi** (*float*) – Set the figure dpi to be at least this amount. If it's already more than *dpi*, don't mess with it.

5.7 pycaps.plot.skewTlib module

Skew-T Plotting Library

class SkewTPlotter

Bases: `object`

Plot Skew-T log-p diagrams

p_lines

list

A list of the pressure lines (in hPa) to plot.

p_color
str

The color of the pressure lines

T_plot_min
float

Minimum isotherm on the plot in degrees C.

T_step
float

The temperature difference (in degrees C) between adjacent isotherms.

T_color_below_0
str

The color of isotherms below 0 C

T_color_0
str

The color of the 0 C isotherm

T_color_above_0
str

The color of isotherms above 0 C

th_min
float

Minimum dry adiabat on the plot in degrees C (references temperature at 1000 mb).

th_max
float

Maximum dry adiabat on the plot in degrees C (references temperature at 1000 mb).

th_step
float

Temperature difference (in degrees C) between adjacent dry adiabats.

th_color
str

The color of the dry adiabats.

the_min
float

Minimum moist adiabat on the plot in degrees C (references temperature at 1000 mb).

the_max
float

Maximum moist adiabat on the plot in degrees C (references temeprature at 1000 mb).

the_step
float

Temperature difference (in degrees C) between adjacent moist adiabats.

the_color
float

The color of the moist adiabats.

w_lines
list

A list of mixing ratio lines (in g/kg) to plot.

w_min_p
float

The starting pressure (in hPa) for mixing ratio lines.

w_max_p
float

The ending pressure (in hPa) for mixing ratio lines.

w_color
str

The color of the mixing ratio lines.

T_max
float: The maximum temperature of the plot (at 1000 hPa) in degrees C.

T_min
float: The minimum temperature of the plot (at 1000 hPa) in degrees C.

p_max
float: The maximum pressure of the plot in hPa

p_min
float: The minimum pressure of the plot in hPa

plot_hodograph(*u_snd*, *v_snd*, *p_snd*, *clip_p=200.0*, *bounds=(-20*, *-20*, *40*, *40)*, ***kwargs*)
Plot a hodograph inset in the upper-right corner

Parameters

- **u_snd** (*np.array*) – A 1-dimensional array containing the u wind component
- **v_snd** (*np.array*) – A 1-dimensional array containing the v wind component
- **p_snd** (*np.array*) – A 1-dimensional array containing the pressure in hPa.
- **clip_p** (*float*) – The pressure in hPa at which to cut off the line on the hodograph. The default is 200 hPa.
- **bounds** (*tuple*) – The bounds for the hodograph as a tuple (u_min, v_min, u_max, v_max). The default is (-20, -20, 40, 40)
- ****kwargs** – Any keywords that can be passed to matplotlib.pyplot.plot() (see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot).

plot_profile(*t_snd*, *p_snd*, ***kwargs*)
Plot a temperature or dewpoint profile on the skew-T

Parameters

- **t_snd** (*np.array*) – A 1-dimensional array containing the profile temperature in degrees C.
- **p_snd** (*np.array*) – A 1-dimensional array containing the profile pressure in hPa.

- ****kwargs** – Any keywords that can be passed to matplotlib.pyplot.plot() (see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot; note that if you pass in a transform, it will be overwritten).

plot_skewt_background(*staff=True*, *clip=False*)

Plots the background for the Skew-T diagram. Must be called first to initialize the plotting.

Parameters

- **staff** (*bool*) – If True, draw a line for the wind staff. Default is True.
- **clip** (*bool*) – If True, clip the top right corner of the plot. Default is False.

plot_winds(*u_snd*, *v_snd*, *p_snd*, ***kwargs*)

Plot a wind profile on the right side of the skew-T

Parameters

- **u_snd** (*np.array*) – A 1-dimensional array containing the u wind profile
- **v_snd** (*np.array*) – A 1-dimensional array containing the v wind profile
- **p_snd** (*np.array*) – A 1-dimenionsal array containing the pressure profile
- ****kwargs** – Any keywords that can be passed to matplotlib.pyplot.barbs() (see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.barbs; note that if you pass in clip_on or z_order, it will be overwritten)

trans

The skew-T log-p transform as a matplotlib transform object.

compute_parcel_trace(*t_snd*, *td_snd*, *p_snd*, *prc_func=<function sb_parcel>*)

Compute the parcel temperature trace

Parameters

- **t_snd** (*np.array*) – A 1-dimensional float array containing the environmental temperature
- **td_snd** (*np.array*) – A 1-dimensional float array containing the environmental dewpoint
- **p_snd** (*np.array*) – A 1-dimensional float array containing the pressure
- **prc_func** (*function*) – A function that returns the initial parcel temperature, pressure, and dewpoint (in that order) given the sounding temperature, pressure, and dewpoint (in that order). Some pre-defined functions are available in derived.parcel.

Returns A 1-dimensional array containing the parcel temperature trace.

plot_sounding(*u*, *v*, *p*, *t*, *td*, *file_name*, *hodograph=False*, *hodo_bounds=(-20, -20, 40, 40)*)

Plot a single sounding, optionally with a hodograph, and save to a file

Parameters

- **u** (*np.array*) – A 1-dimensional array containing the u wind component
- **v** (*np.array*) – A 1-dimensional array containing the v wind component
- **p** (*np.array*) – A 1-dimensional array containing the pressure in hPa
- **t** (*np.array*) – A 1-dimensional array containing the temperature in degrees C
- **td** (*np.array*) – A 1-dimensional array containing the dewpoint in degrees C
- **file_name** (*str*) – The name of the file in which to save the image

- **hodograph** (*bool*) – If True, plot a hodograph in the upper-right corner
- **hodo_bounds** (*tuple*) – Bounds on the hodograph as a tuple (u_min, v_min, u_max, v_max), if one is plotted. The default is (-20, -20, 40, 40)

5.8 Module contents

CHAPTER

SIX

PYCAPS.UTIL PACKAGE

6.1 Submodules

6.2 pycaps.util.decompress module

arps_decompress(*hd_var*, *dindex=None*)

Decompresses variable fields from ARPS HDF files that use high compression options (specifically, the mapping of 32 bit –> 16 bit integers).

Parameters

- **hd_var** – The compressed variable field
- **dindex** – Option to return a slice of the compressed data if only a slice is desired. The index may be either an integer (interpreted as an element of the first dimension) or a tuple. The tuple must have no more elements than the data has dimensions. Each element must be either an integer (a specific element along the given axis) or a Python slice object (a slice along the given axis). For example, dindex=(0, slice(None), slice(2, 6)) would return the first element along the first axis, all elements along the second axis, and elements 3-7 along the third axis (equivalent to asking for ary[0, :, 2:6], where ary is a numpy array).

Returns The decompressed equivalent of hd_var or the requested slice thereof.

6.3 pycaps.util.grid module

class ExperimentGrid(***kwargs*)

Bases: `object`

Represents an ARPS Arakawa C grid, with provisions for subsets.

Parameters

- **grid_size** (*tuple*) – Tuple of integers (NX, NY) representing the number of grid points in each dimension of the grid.
- **grid_spacing** (*tuple*) – Tuple of floats (DX, DY) representing the grid spacing in meters in each dimension of the grid.
- **bounds** (*tuple*) – Tuple of slice objects (SLICEX, SLICEY) representing the bounds in each direction of a subset of the grid. Default is (slice(None), slice(None))
- **mpi** (*tuple*) – Tuple of integers (NPROCX, NPROCY) representing the MPI configuration for this grid. Default is (1, 1), meaning no MPI.

- **state_list** (*list*) – An optional list of strings containing the two-letter postal codes for each state visible in the domain. This is used for optimizing the drawPolitical() method.
- **projection** (*str*) – Map projection to use. Only tested value is 'lcc' for Lambert conic conformal.
- **resolution** (*str*) – Resolution of the mapping data to load. Acceptable values are 'c' (crude), 'l' (low), and 'i' (intermediate). Recommended value is 'i'.
- **lat_0** – Center latitude for the Lambert projection (degrees N)
- **lat_1** – True latitude 1 for the Lambert projection (degrees N)
- **lat_2** – True latitude 2 for the Lambert projection (degrees N)
- **lon_0** – True longitude for the Lambert projection (degrees E)
- **ctr_lat** – Center latitude for the domain (degrees N)
- **ctr_lon** – Center longitude for the domain (degrees E)

__call__(**args*, ***kwargs*)

Convert between latitude, longitude coordinates and x, y coordinates on the domain.

The only meaningful keyword argument is *inverse*, which should be a boolean specifying whether to convert from latitudes and longitudes to x, y (False) or vice-versa (True). Default is False.

If inverse is False, *args is two numpy arrays of longitudes and latitudes in degrees. If inverse is True, *args is two numpy arrays of x and y coordinates in meters.

Returns A tuple of numpy arrays of either x and y coordinates in meters or longitudes and latitudes in degrees.

core_bounds(*icore*, *jcore*)

Get the grid point bounds for a particular core.

Parameters

- **icore** (*int*) – i-index of the core on the domain.
- **jcore** (*int*) – j-index of the core on the domain.

Returns A tuple of slice objects (BOUNDX, BOUNDY).

draw_political(*scale_len=0*, *overlays=[]*, *color='k'*, *lw=1*)

Draw political boundaries on the current matplotlib axes.

Parameters

- **scale_len** (*float*) – Length of a scale bar in km to draw in the bottom-left corner of the plot. Default is 0, meaning no scale.
- **overlays** (*list*) – A list of strings pointing to shapefiles to plot as overlays.
- **color** (*str*) – Color in which to draw the political boundaries. Defaults to 'k' (black).
- **lw** (*float*) – Base line width for the political boundaries. Country boundaries are drawn with width *lw*, states with width *1.5 * lw*, and counties are drawn with width *0.5 * lw*

static **from_file**(*fobj*, *projection='lcc'*, *resolution='l'*, *mpi=(1, 1)*, *bounds=None*)

Construct an ExperimentGrid object from information found in a history file. Should work for WRF and ARPS.

Parameters

- **fobj** – File object (such as a NetCDF or HDF file) from which to take the information.

- **projection** (*str*) – Map projection to use. Currently, the only supported value is 'lcc' for Lambert Conic Conformal.
- **resolution** (*str*) – Resolution of the mapping data to load. Acceptable values are 'c' (crude), 'l' (low), and 'i' (intermediate). Recommended value is 'i'.
- **mpi** (*tuple*) – Tuple of integers (NPROCX, NPROCY) representing the MPI configuration for this grid. Default is (1, 1), meaning no MPI.
- **bounds** (*tuple*) – Tuple of slice objects (SLICEX, SLICEY) representing the bounds in each direction of a subset of the grid. Default is (slice(None), slice(None))

Returns An ExperimentGrid object with parameters matching those in the input file.

Return type *ExperimentGrid*

get_boundary_coords()

Get the latitudes and longitudes of the boundary of this domain (useful for drawing this domain using another projection).

Returns A tuple of 1-dimensional numpy arrays containing the latitudes and longitudes of the domain boundary in degrees.

get_bounds(*pcm=False*)

Get the grid point bounds for this subdomain

Parameters **pcm** (*bool*) – Whether to assume this will be used in matplotlib's *pcolormesh()* function. Default is False.

Returns The bounds as a tuple of slice objects (SLICEY, SLICEX)

get_center()

Get the center lat and lon in the domain.

Returns A tuple of floats (LAT, LON) in degrees.

get_cores()

Get a list of cores that are in the domain subset.

Returns A list of tuples containing the i and j indices of the cores in the domain subset.

get_grid_size()

Get the number of grid points.

Returns The number of grid points in the full domain as a tuple (NX, NY).

get_grid_spacing()

Get the grid spacing.

Returns The grid spacing in meters as a tuple (DX, DY).

get_width_height(*override=False*)

Get the width and height of the domain.

Returns The width and height of the domain in meters of the subsetted domain as a tuple (WIDTHX, WIDTHY)

get_xy(*ij=None*, *pcm=False*)

Get x, y coordinates of the center each grid point or a specific grid point.

Parameters

- **ij** (*tuple*) – Get the coordinates of the grid point specified by (i, j). Optional.
- **pcm** (*bool*) – Get the coordinates of the corners of the cells, rather than the centers.

Returns If ij is not specified, a tuple of 2-dimensional numpy arrays (x, y). Axes are ordered (NX, NY). If ij is specified, returns a tuple of floats. Both are in meters.

6.4 pycaps.util.make_proj_grids module

get_proj_obj(*proj_dict*)

make_proj_grids(*proj_dict*, *grid_dict*)

proj_main()

read_arps_map_file(*map_filename*)

read_ncar_map_file(*map_filename*)

6.5 pycaps.util.progress module

class ProgressBar(*descr*, *n_steps*, *width=20*)

Bases: `object`

complete_step(*print_bar=True*)

Tell a progress bar that one of the tasks is complete.

Parameters **print_bar** (*bool*) – Whether or not to print the bar after this step (defaults to True).

initialize()

Starts timing the task and prints the initial progress bar to the screen.

is_complete()

Is the task complete?

Returns A boolean specifying whether or not the task is complete.

set_step(*step*, *print_bar=True*)

Tell the progress bar that we have completed a certain number of tasks.

Parameters

- **step** (*int*) – The number of tasks we have completed.
- **print_bar** (*bool*) – Whether or not to print the bar after this step (defaults to True).

6.6 pycaps.util.temporal module

class PatchedTemporal(**args*)

Bases: `object`

Keep track of time in an experiment and do time conversions, but for timelines in which the time step is not constant.

__getitem__(*index*)

Get the time at a particular index.

Parameters **index** (*int*) – The index at which to return the time.

Returns The number of seconds at that index since the experiment reference time.

Return type int

Examples

```
>>> temp1 = Temporal(0, 1200, 300)
>>> temp2 = Temporal(1200, 1800, 150)
>>> temp = PatchedTemporal(temp1, temp2)
>>> temp[6]
1500
```

__iter__()

Iterate over the experiment times.

Yields *int* – The next time in seconds since the experiment reference time.

Examples

```
>>> temp1 = Temporal(0, 1200, 300)
>>> temp2 = Temporal(1200, 1800, 150)
>>> temp = PatchedTemporal(temp1, temp2)
>>> [ t for t in temp ]
[ 0, 300, 600, 900, 1200, 1350, 1500, 1650, 1800 ]
```

__len__()

Get the number of times in the experiment.

Returns The number of times in the experiment.

Return type int

Examples

```
>>> temp1 = Temporal(0, 1200, 300)
>>> temp2 = Temporal(1200, 1800, 150)
>>> temp = PatchedTemporal(temp1, temp2)
>>> len(temp)
9
```

get_datetimes(*aslist=False*)

Get a list of datetimes represented by the Temporal object.

Parameters **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of datetime objects.

get_epochs(*aslist=False*)

Get a list of times represented by the Temporal object.

Parameters **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of times in seconds since the epoch.

get_strings (*format*, *aslist=False*)
Get a list of times as formatted strings.

Parameters

- **format** (*str*) – The string format to use.
- **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of strings containing the formatted times.

get_times ()
Get the list of times represented by the Temporal object.

Returns A list of times in seconds since the experiment reference time.

sec_to_dt (*exp_seconds*)
Convert a number of seconds since the experiment reference time to a datetime object.

Parameters **exp_seconds** (*int*) – The number of seconds since the experiment reference time.

Returns A datetime object representing that time.

Return type datetime

sec_to_epoch (*exp_seconds*)
Convert a time in seconds since experiment reference time to a time in seconds since the epoch.

Parameters **exp_seconds** (*int*) – The number of seconds since the experiment reference time.

Returns A number of seconds since the epoch.

Return type int

class Temporal (*base_time*, *t_ens_start*, *t_ens_end*, *t_ens_step*)
Bases: `object`

Keep track of time in an experiment and convert times between seconds since experiment start, datetime objects and seconds since the epoch.

__getitem__ (*index*)
Get the time at a particular index.

Parameters **index** – The index at which to return the time.

Returns If *index* is an integer, return the number of seconds at *index* since the experiment reference time. If *index* is a slice, return a new Temporal object with subsetted according to the slice.

Examples

```
>>> dt = datetime(2011, 5, 24, 18)
>>> temp = Temporal(dt, 0, 1800, 300)
>>> temp[3]
900
```

__iter__ ()
Iterate over the experiment times.

Yields *int* – The next time in seconds since the experiment reference time.

Examples

```
>>> dt = datetime(2011, 5, 24, 18)
>>> temp = Temporal(dt, 0, 1800, 300)
>>> [ t for t in temp ]
[ 0, 300, 600, 900, 1200, 1500, 1800 ]
```

__len__()

Get the number of times in the experiment.

Returns The number of times in the experiment.

Return type int

Examples

```
>>> dt = datetime(2011, 5, 24, 18)
>>> temp = Temporal(dt, 0, 1800, 300)
>>> len(temp)
7
```

get_datetimes(*aslist=False*)

Get a list of datetimes represented by the Temporal object.

Parameters **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of datetime objects.

get_epochs(*aslist=False*)

Get a list of times represented by the Temporal object.

Parameters **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of times in seconds since the epoch.

get_strings(*str_format*, *aslist=False*)

Get a list of times as formatted strings.

Parameters

- **str_format** (*str*) – The string format to use.
- **aslist** (*bool*) – Whether to return a list (True) or a Python generator (False). They can be used mostly interchangeably, but a generator is more efficient. Default is False (return a generator).

Returns A Python generator or list of strings containing the formatted times.

get_times()

Get the list of times represented by the Temporal object.

Returns A list of times in seconds since the experiment reference time.

sec_to_datetime(*exp_seconds*)

Convert a number of seconds since the experiment reference time to a datetime object.

Parameters **exp_seconds** (*int*) – The number of seconds since the experiment reference time.

Returns A datetime object representing that time.

Return type datetime

sec_to_epoch(*exp_seconds*)
Convert a time in seconds since experiment reference time to a time in seconds since the epoch.

Parameters **exp_seconds** (*int*) – The number of seconds since the experiment reference time.

Returns A number of seconds since the epoch.

Return type int

dt_to_epoch(*dt*)
Convert a Python datetime object to a number of seconds since the epoch (00 UTC 1 January 1970).

Parameters **dt** (*datetime*) – A Python datetime object

Returns Time since the epoch in seconds.

epoch_to_dt(*epoch*)
Convert a number of seconds since the epoch (00 UTC 1 January 1970) to a Python datetime object.

Parameters **epoch** (*float*) – Time since the epoch in seconds

Returns A Python datetime object.

temporal(*base_time*, **slices*)
Create an object to keep track of time in an experiment and do time-related conversions.

Parameters

- **base_time** (*datetime*) – The base time in the experiment (i.e. inittime in the ARPS input file) as a datetime object.
- ***slices** (*slice*) – One or more slice objects, each specifying start and end times (in seconds since the base time) and the time step length (in seconds) for a segment. Multiple segments are chained together under the hood to form a continuous timeline.

Returns A Temporal or PatchedTemporal object (both have the same interface).

6.7 pycaps.util.util module

class abstract(*func*, *custom_name=None*)
Bases: object

Use as a decorator to declare a method as abstract.

format_arps_time(*rawtime*)
Takes an integer number provided and converts it to the six-digit ARPS time format, which is then returned.

Parameters **rawtime** (*str*) – The number to be converted into the ARPS time format

Returns rawtime converted into the ARPS time format

run_concurrently(*target*, *placeholder_vals*, *args=[]*, *kwargs={}*, *max_concurrent=-1*, *zip_result=False*, *progress=None*)
Runs several instances of a function at the same time and returns all their outputs as a list.

Parameters

- **target** (*function*) –
- **placeholder_vals** (*list*) – Values of a placeholder parameter to run the function on.
- **args** (*list*) – Arguments to pass to the target function. One or more may have the special value "__placeholder__", which will be replaced with a value from placeholder_vals for each instance of the function.
- **kwargs** (*dict*) – Keyword arguments to pass to the target function. One or more may have the special value "__placeholder__", which will be replaced with a value from placeholder_vals for each instance of the function.
- **max_concurrent** (*int*) – Maximum number of function instances to run at the same time. The default is to run an instance for each value in placeholder_vals at the same time.
- **zip_result** (*bool*) – A boolean specifying whether to run the zip() function on the result, perhaps if the function returns a tuple of values. Default is False.
- **progress** (ProgressBar) – An optional progress bar instance to use in displaying progress on the tasks.

Returns A list of the return values from each instance of the function, sorted by the corresponding placeholder value.

6.8 pycaps.util.wsr88d module

class RadarMeta (*filename='data/radarinfo.dat'*)

Bases: object

Class for keeping track of metadata for US radars.

__call__ (*network*)

Overrides the function call syntax (e.g. obj(...))

Parameters **network** (*str*) – Name of the network from which to return the radar ids. The expected values are 'tdwr' for TDWRs, 'casa' for the CASA radar network, 'legacy' for legacy radars, 'research' for research radars, and 'wsr88d' for the WSR-88D network. The default value is None, which returns all radar ids.

Returns An iterator over radar metadata for the network.

__getitem__ (*key*)

Overloads the container access operator (e.g. obj[...])

Parameters **key** (*str*) – A 4-character radar ID

Returns A dictionary containing the metadata for the radar with the given 4-character ID

__iter__ ()

Overloads the iterator (e.g. for elem in obj)

Returns An iterator over all the radars

__len__ ()

Overrides the length operator (e.g. len(obj))

Returns The total number of radars.

get_radar_ids (*network=None*)

Get a list of radar ids from a given network

Parameters **network** (*str*) – Name of the network from which to return the radar ids. The expected values are 'tdwr' for TDWRs, 'casa' for the CASA radar network, 'legacy' for legacy radars, 'research' for research radars, and 'wsr88d' for the WSR-88D network. The default value is None, which returns all radar ids.

Returns A list of 4-character radar ids

get_radar_meta(*radar_id*)

Gets the metadata for the given 4-character radar id.

Parameters **radar_id** (*str*) – The 4-character id for a radar

Returns A dictionary containing the latitude, longitude, name, and ID of the radar.

6.9 Module contents

CHAPTER

SEVEN

PYCAPS.VERIFY PACKAGE

7.1 Submodules

7.2 pycaps.verify.ContingencyTable module

class `ContingencyTable`(*tp*, *fp*, *fn*, *tn*)

Bases: `object`

Initializes a contingency table of the following form:

Event Yes No

Forecast Yes a b No c d

`tp`

int

Number of true positives or hits.

fp (int): Number of false positives. fn (int): Number of false negatives or misses. tn (int): Number of true negatives.

`__add__`(*other*)

Add two contingency tables together and return a combined one.

Parameters `other` –

Returns

`accuracy`()

Finley's measure, fraction correct, accuracy (a+d)/N

`bias`()

Returns Frequency Bias. Formula: (a+b)/(a+c)

`csi`()

Gilbert's Score or Threat Score or Critical Success Index a/(a+b+c)

`css`()

Clayton Skill Score (ad - bc)/((a+b)(c+d))

`dfr`()

Returns Detection Failure Ratio (DFR). Formula: c/(c+d)

`ets`()

Equitable Threat Score, Gilbert Skill Score, v, (a - R)/(a + b + c - R), R=(a+b)(a+c)/N

far()
Returns False Alarm Ratio (FAR). Formula: b/(a+b)

focn()
Returns Frequency of Correct Null (FOCN). Formula: d/(c+d)

foh()
Returns Frequency of Hits (FOH) or Success Ratio. Formula: a/(a+b)

fom()
Returns Frequency of Misses (FOM). Formula: c/(a+c).

hss()
Doolittle (Heidke) Skill Score. 2(ad-bc)/((a+b)(b+d) + (a+c)(c+d))

pod()
Returns Probability of Detection (POD) or Hit Rate. Formula: a/(a+c)

pofd()
Returns Probability of False Detection (POFD). b/(b+d)

pon()
Returns Probability of Null (PON). Formula: d/(b+d)

pss()
Peirce (Hansen-Kuipers, True) Skill Score (ad - bc)/((a+c)(b+d))

update(*tp*, *fp*, *fn*, *tn*)
Update contingency table with new values without creating a new object.

7.3 pycaps.verify.MulticlassContingencyTable module

class **MulticlassContingencyTable**(*table=None*, *n_classes=2*, *class_names=('1', '0')*)
Bases: `object`

This class is a container for a contingency table containing more than 2 classes. The contingency table is stored in table as a numpy array with the rows corresponding to forecast categories, and the columns corresponding to observation categories.

gerrity_score()
Gerrity Score, which weights each cell in the contingency table by its observed relative frequency. :return:

heidke_skill_score()

peirce_skill_score()
Multiclass Peirce Skill Score (also Hanssen and Kuipers score, True Skill Score)

mct_main()

7.4 pycaps.verify.ProbabilityMetrics module

class **DistributedCRPS**(*thresholds=None*, *input_str=None*)
Bases: `object`

A container for the statistics used to calculate the Continuous Ranked Probability Score

Parameters

- **thresholds** (*numpy.ndarray*) – Array of the intensity threshold bins

- **input_str** (*str*) – String containing the information for initializing the object from a storable text format.

crps()
Calculates the continuous ranked probability score.

Returns

from_str(*in_str*)

merge(*other_crps*)

update(*forecasts*, *observations*)
Update the statistics with forecasts and observations.

Parameters

- **forecasts** –
- **observations** –

Returns

class DistributedROC(*thresholds=None*, *obs_threshold=None*, *input_str=None*)
Bases: `object`

Store statistics for calculating receiver operating characteristic (ROC) curves and performance diagrams and permit easy aggregation of ROC curves from many small datasets.

Parameters

- **thresholds** (*numpy.ndarray of floats*) – List of probability thresholds in increasing order.
- **obs_threshold** (*float*) – Observation value used as the split point for determining positives.
- **input_str** (*str*) – String in the format output by the __str__ method so that initialization of the object can be done from items in a text file.

__add__(*other*)
Add two DistributedROC objects together and combine their contingency table values.

Parameters **other** – Another DistributedROC object.

Returns

__str__()
Output the information within the DistributedROC object to a string.

Returns

auc()
Calculate the Area Under the ROC Curve (AUC).

Returns

from_str(*in_str*)
Read the object string and parse the contingency table values from it. :param in_str: :return:

merge(*other_roc*)
Ingest the values of another DistributedROC object into this one and update the statistics inplace.

Parameters **other_roc** – another DistributedROC object.

Returns

performance_curve()
Calculate the Probability of Detection and False Alarm Ratio in order to output a performance diagram.

Returns pandas.DataFrame containing POD, FAR, and probability thresholds.

roc_curve()
Generate a ROC curve from the contingency table by calculating the probability of detection (TP/(TP+FN)) and the probability of false detection (FP/(FP+TN)).

Returns A pandas.DataFrame containing the POD, POFD, and the corresponding probability thresholds.

update(*forecasts*, *observations*)
Update the ROC curve with a set of forecasts and observations

Parameters

- **forecasts** – 1D array of forecast values
- **observations** – 1D array of observation values.

Returns

class **DistributedReliability**(*thresholds=None*, *obs_threshold=None*, *input_str=None*)
Bases: `object`

A container for the statistics required to generate reliability diagrams and calculate the Brier Score.

Parameters

- **thresholds** (*numpy.ndarray*) – Array of probability thresholds
- **obs_threshold** (*float*) – Split value for the observations
- **input_str** (*str*) – String containing information to initialize the object from a text representation.

__add__(*other*)
Add two DistributedReliability objects together and combine their values.

Parameters **other** – a DistributedReliability object

Returns a DistributedReliability Object

brier_score()
Calculate the Brier Score

Returns

brier_score_components()
Calculate the components of the Brier score decomposition: reliability, resolution, and uncertainty.

Returns

brier_skill_score()
Calculate the Brier Skill Score

Returns

climatology()

from_str(*in_str*)

merge(*other_rel*)

reliability_curve()
Calculate the reliability diagram statistics.

Returns

update (*forecasts*, *observations*)
Update the statistics with a set of forecasts and observations.

Parameters

- **forecasts** –
- **observations** –

Returns

class ROC (*forecasts*, *observations*, *thresholds*, *obs_threshold*)
Bases: `object`

auc()

calc_roc()

class Reliability (*forecasts*, *observations*, *thresholds*, *obs_threshold*)
Bases: `object`

brier_score()

brier_score_components()

brier_skill_score()

calc_reliability_curve()

bootstrap (*score_objs*, *n_boot=1000*)

7.5 pycaps.verify.verif_modules module

NEP_2D (*var2D_ens*, *threshold*, *kernel*)
A function for performing neighborhood ensemble probability (NEP)

Given a ensemble of 2D slices (var2D_ens), this function will perform the NEP of P[var2D > threshold] and return a corresponding 2D array with the NEP values. The NEP will use a neighborhood defined by kernel. The kernel is a 2D array and is defined using the define_kernel function (also contained in this library).

Parameters

- **var2D_ens** – The 2D ensemble of x-y slices to calculate NEP for [n_ens, ny, nx]
- **threshold** – NEP will be calculated for P[var2D_ens > threshold] within the neighborhood where threshold can be any real number.
- **kernel** – a 2D array of some size < (ny, nx) with values between 0 and 1 to serve as the kernel for the 2D convolution.

Returns a 2D [ny, nx] field containing NEP for P[var2D_ens > threshold] for the given neighborhood (defined by kernel)

Return type var_NEP

define_kernel (*radius*, *dropoff=0.0*)
Defines a 2D kernel suitable for use in neighborhood ensemble probability calculations (or other verification methods requiring the use of a 2D convolution).

Parameters

- **radius** – The radius of the kernel, in gridpoints (can be any whole number)

- **dropoff** – Set this to a value between 0.0 and 1.0 to have points near the edge of the radius recieve a lower weight. By default, dropoff is set to 0.0, giving all points within the radius equal weight.

Returns a 2D array containing the kernel needed for performing convolutions

Return type Kernel

define_kernel_ellipse (*xradius*, *yradius*, *dropoff=0.0*)

Defines an elliptical 2D kernel suitable for use in neighborhood ensemble probability calculations (or other verification methods requiring the use of a 2D convolution).

Parameters

- **xradius** – The semi-major (or semi-minor) axis in the east-west direction, in gridpoints (can be any whole number)
- **yradius** – The semi-major (or semi-minor) axis in the north-south direction, in gridpoints (can be any whole number)
- **dropoff** – Set this to a value between 0.0 and 1.0 to have points near the edge of the radius recieve a lower weight. By default, dropoff is set to 0.0, giving all points within the radius equal weight.

Returns a 2D array containing the kernel needed for performing convolutions

Return type Kernel

ens_prob_within_dist (*var2D_ens*, *threshold*, *kernel*)

A function for calculating ensemble probability of the form "probability of X within Y km of a point". Similar to, but distinct from, NEP_2D.

Given a ensemble of 2D slices (var2D_ens), this function will compute the ensemble probability of P[var2D > threshold (anywhere within kernel)] and return a corresponding 2D array with the probability values. The kernel is a 2D array and is defined using the define_kernel function (also contained in this library).

Parameters

- **var2D_ens** – The 2D ensemble of x-y slices to calculate NEP for [n_ens, ny, nx]
- **threshold** – NEP will be calculated for P[var2D_ens > threshold] within the neighborhood where threshold can be any real number.
- **kernel** – a 2D array of some size < (ny, nx) with values of 0 or 1 to serve as the kernel for the 2D convolution.

Returns a 2D [ny, nx] field containing NEP for P[var2D_ens > threshold] for the given neighborhood (defined by kernel)

Return type ens_prob

pm_mean_2D (*var2D_ens*)

A function for computing probability-matched mean (PM mean)

Given an ensemble of 2D slices (var2D_ens), this function will perform the PM mean calculation and return the PM mean as a 2D array over the same slice.

var2D_ens should have dimensions of (n_ens, ny, nx).

Parameters **var2D_ens** – an ensemble of 2D slices of a variable field [nens, ny, nx]

Returns the probability-matched mean of var2D_ens

Return type pm_mean

7.6 Module contents

p

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

X

Z