domutils

Release 2.2.0

Dominik Jacques

Nov 09, 2023

LEGS

The package **domutils** provides 3 modules:

CHAPTER

ONE

LEGS TUTORIAL

A **leg** can be defined as:

- *a portion of a trip*
- *one section of a relay race*

Here the race happens in data space and goes from - infinity to + infinity.

The general idea is to assign a number of distinct color mappings, hereafter called legs, to contiguous portions of this (quite large) range.

There can be any number of legs and what happens beyond the range of the color mapping must be specified explicitly. A mechanism is also provided for assigning colors to any number of exception values.

By doing so, it becomes easy to create and modify continuous, semi-continuous, categorical and divergent color mappings.

This tutorial demonstrates how to construct custom color mappings. Elements covered include:

Table of Contents

1.1 Very basic color mapping

For this tutorial, lets start by making data and setting up figure information.

```
>>> import numpy as np
>>> import scipy.signal
>>> import matplotlib.pyplot as plt
>>> import matplotlib as mpl
>>> import domutils.legs as legs
>>>
>>> #Gaussian bell mock data
>>> npts = 1024
>>> half_npts = int(npts/2)
>>> x = np.linspace(-1., 1, half_npts+1)
>>> y = np.linspace(-1., 1, half_npts+1)
>>> xx, yy = np.meshgrid(x, y)
>>> gauss_bulge = np.exp(-(xx**2 + yy**2) / .6**2)
>>>
>>> #radar looking mock data
>>> sigma1 = .03
>>> sigma2 = .25
>>> np.random.seed(int(3.14159*100000))
>>> rand = np.random.normal(size=[npts,npts])
>>> xx, yy = np.meshgrid(np.linspace(-1.,1.,num=half_npts),np.linspace(1.,-1.,num=half_
↪npts))
>>> kernel1 = np.exp(-1.*np.sqrt(xx**2.+yy**2.)/.02)
>>> kernel2 = np.exp(-1.*np.sqrt(xx**2.+yy**2.)/.15)
>>> reflectivity_like = (   scipy.signal.fftconvolve(rand,kernel1,mode='valid')
...                       + scipy.signal.fftconvolve(rand,kernel2,mode='valid') )
>>> reflectivity_like = ( reflectivity_like
...                       / np.max(np.absolute(reflectivity_like.max()))
...                       * 62. )
>>>
>>> # figure properties
>>> mpl.rcParams.update({'font.size': 15})
>>> rec_w = 4.           #size of axes
>>> rec_h = 4.           #size of axes
>>> sp_w  = 2.           #horizontal space between axes
>>> sp_h  = 1.           #vertical space between axes
>>>
>>> # a function for formatting axes
>>> def format_axes(ax):
...     for axis in ['top','bottom','left','right']:
...         ax.spines[axis].set_linewidth(.3)
...     limits = (-1.,1.)           #data extent for axes
...     dum = ax.set_xlim(limits)   # "dum =" to avoid printing output
...     dum = ax.set_ylim(limits)
...     ticks  = [-1.,0.,1.]        #tick values
...     dum = ax.set_xticks(ticks)
...     dum = ax.set_yticks(ticks)
```

The default color mapping applies a black and wite gradient in the interval [0,1].

```
>>> fig_w, fig_h = 5.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> format_axes(ax)
>>>
>>> #instantiate default color mapping
>>> mapping = legs.PalObj()
>>>
>>> #plot data & palette
>>> mapping.plot_data(ax=ax,data=gauss_bulge,
...                   palette='right',
...                   pal_format='{:2.0f}')
>>>
>>> plt.savefig('_static/default.svg')
```

1.2 Data values above and below the color palette

The default behavior is to throw an error if data values are found beyond the range of of the color palette. The following code will fail and give you suggestions as to what to do.

```
>>>
>>>
>>> #extend range of data to plot beyond 1.0
>>> extended_gauss_bulge = 1.4 * gauss_bulge - 0.2 # data range is now [-.2, 1.2]
>>>
>>>
>>> mapping.plot_data(ax=ax,data=extended_gauss_bulge,
...                   palette='right', pal_format='{:2.0f}')
Traceback (most recent call last):
  File "/fs/site3/eccc/mrd/rpndat/dja001/python_miniconda3/envs/domutils_dev/lib/python3.
↪7/doctest.py", line 1330, in __run
    compileflags, 1), test.globs)
  File "<doctest legsTutorial.rst[35]>", line 2, in <module>
    palette='right', pal_format='{:2.0f}')
  File "/fs/homeu1/eccc/mrd/ords/rpndat/dja001/python/packages/domutils_package/domutils/
↪legs/legs.py", line 405, in plot_data
    out_rgb = self.to_rgb(rdata)
  File "/fs/homeu1/eccc/mrd/ords/rpndat/dja001/python/packages/domutils_package/domutils/
↪legs/legs.py", line 473, in to_rgb
    validate.no_unmapped(data_flat, action_record, self.lows, self.highs)
  File "/fs/homeu1/eccc/mrd/ords/rpndat/dja001/python/packages/domutils_package/domutils/
↪legs/validation_tools/no_unmapped.py", line 103, in no_unmapped
    raise RuntimeError(err_mess)
RuntimeError:

Found data point(s) smaller than the minimum of an exact palette:
  [-0.19458771 -0.19446921 -0.19434859 -0.19434811 -0.19422583]...
```

(continues on next page)

(continued from previous page)

```
Found data point(s) larger than the maximum of an exact palette:
  [1.00004429 1.00055305 1.00060393 1.0008584  1.00101111]...


One possibility is that the data value(s) exceed the palette
while they should not.
   For example: correlation coefficients greater than one.
   In this case, fix your data.

Another possibility is that data value(s) is (are) expected
above/below the palette.
In this case:
  1- Use the "over_under","over_high" or "under_low" keywords to explicitly
     assign a color to data values below/above the palette.
  2- Assign a color to exception values using the "excep_val" and "excep_col" keywords.
     For example: excep_val=-9999., excep_col="red".
```

Lets assume that we expected data values to exceed the [0,1] range where the color palette is defined. Then we should use the keywords **over_under** or **under_low** and **over_high** to avoid errors.

```
>>> fig_w, fig_h = 11.6, 10.
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #mapping which extends the end-point color beyond the palette range
>>> x0, y0 = (.5+rec_w/2.+sp_w/2.)/fig_w , (.5+rec_h+sp_h)/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('Extend', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
>>> mapping_ext = legs.PalObj(over_under='extend')
>>> mapping_ext.plot_data(ax=ax1,data=extended_gauss_bulge,
...                       palette='right', pal_units='[unitless]',
...                       pal_format='{:2.0f}')
>>>
>>>
>>> #mapping where end points are dealt with separately
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('Extend Named Color', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> mapping_ext_2 = legs.PalObj(over_high='dark_red', under_low='dark_blue')
>>> mapping_ext_2.plot_data(ax=ax2,data=extended_gauss_bulge,
...                         palette='right', pal_units='[unitless]',
...                         pal_format='{:2.0f}')
>>>
>>>
>>> #as for all color specification, RBG values also work
```

(continues on next page)

(continued from previous page)

```
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , .5/fig_h
>>> ax3 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax3.annotate('Extend using RGB', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax3)
>>>
>>> mapping_ext_3 = legs.PalObj(over_high=[255,198, 51], under_low=[ 13,171, 43])
>>> mapping_ext_3.plot_data(ax=ax3,data=extended_gauss_bulge,
...                         palette='right', pal_units='[unitless]',
...                         pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/default_extend.svg')
```

1.3 Exceptions

Exception values to a color mapping come in many forms:

- Missing values from a dataset
- Points outside of a simulation domain
- The zero value when showing the difference between two things
- Water in a topographical map / Land in vertical cross-sections
- etc.

Being able to easily assign colors values allows for all figures of a given manuscript/article to show missing data with the same color.

There can be any number of exceptions associated with a given color mapping. These exceptions have precedence over the regular color mapping and will show up in the associated color palette.

Data points that are outside of an exact color mapping but that are covered by an exception will not trigger an error.

```
>>> fig_w, fig_h = 11.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #Lets assume that data values in the range [0.2,0.3] are special
>>> #make a mapping where these values are assigned a special color
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('1 exceptions inside \npalette range', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
>>> #data values in the range 0.25+-0.05 are assigned the color blue
>>> mapping_1_except = legs.PalObj(excep_val=[.25],
...                                excep_tol=[.05],
```

(continues on next page)

(continued from previous page)

```
...                                 excep_col=[ 71,152,237])
>>> mapping_1_except.plot_data(ax=ax1,data=gauss_bulge,
...                            palette='right', pal_units='[unitless]',
...                            pal_format='{:2.0f}')
>>>
>>>
>>> #exceptions are usefull for NoData, missing values, etc
>>> #lets assing exception values to the Gaussian Bulge data
>>> gauss_bulge_with_exceptions = np.copy(gauss_bulge)
>>> gauss_bulge_with_exceptions[388:488, 25:125] = -9999.
>>> gauss_bulge_with_exceptions[388:488,150:250] = -6666.
>>> gauss_bulge_with_exceptions[388:488,275:375] = -3333.
>>>
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , .5/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('3 exceptions outside \npalette range', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> #a mapping where 3 exceptions are specified
>>> #the defalut tolerance around specified value is 1e-3
>>> mapping_3_except = legs.PalObj(excep_val=[-9999.,      -6666.,    -3333.],
...                                 excep_col=['dark_green','grey_80', 'light_pink'])
>>> mapping_3_except.plot_data(ax=ax2,data=gauss_bulge_with_exceptions,
...                            palette='right', pal_units='[unitless]',
...                            pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/default_exceptions.svg')
```

1.4 Specifying colors

Up to nine legs using linear gradient mapping are specified by default. They can be called by name.

```
>>> pal_w  = .25     #width of palette
>>> pal_sp = 1.2     #space between palettes
>>> fig_w, fig_h = 13.6, 5.  #size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>> supported_colors = ['brown','blue','green','orange',
...                     'red','pink','purple','yellow', 'b_w']
>>> for n, thisCol in enumerate(supported_colors):
...     x0, y0 = (.5+n*(pal_w+pal_sp))/fig_w , .5/fig_h
...     #color mapping with one color leg
...     this_map = legs.PalObj(color_arr=thisCol)
...     #plot palette only
...     this_map.plot_palette(pal_pos=[x0,y0,pal_w/fig_w,rec_h/fig_h],
...                           pal_units=thisCol,
...                           pal_format='{:1.0f}')
>>> plt.savefig('_static/default_linear_legs.svg')
```

1.4.1 Semi-continuous, semi-quantitative color mappings

The keyword **n_col** will create a palette from the default legs in the order in which they appear above.

```
>>> fig_w, fig_h = 5.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> format_axes(ax)
>>> #mapping with 6 color legs
>>> mapping_default_6_colors = legs.PalObj(range_arr=[0.,60], n_col=6,
...                                        over_high='extend',
...                                        under_low='white')
>>> mapping_default_6_colors.plot_data(ax=ax,data=reflectivity_like,
...                                    palette='right', pal_units='[dBZ]',
...                                    pal_format='{:2.0f}')
>>> plt.savefig('_static/default_6cols.svg')
```

The keyword **color_arr** can be used to make a mapping from the default color legs in whatever order. It can also be used to make custom color legs from RGB values. By default linear interpolation is used between the provided RGB.

```
>>> fig_w, fig_h = 11.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #mapping with 3 of the default color legs
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('3 of the default \ncolor legs', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
>>> mapping_3_colors = legs.PalObj(range_arr=[0.,60],
...                                color_arr=['brown','orange','red'],
...                                over_high='extend',
...                                under_low='white')
>>> mapping_3_colors.plot_data(ax=ax1,data=reflectivity_like,
...                            palette='right', pal_units='[dBZ]',
...                            pal_format='{:2.0f}')
>>>
>>>
>>> #mapping with custom pastel color legs
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , .5/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('Custom pastel \ncolor legs', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> #Custom pastel colors
```

(continues on next page)

(continued from previous page)

```
>>> pastel = [ [[255,190,187],[230,104, 96]],  #pale/dark red
...            [[255,185,255],[147, 78,172]],  #pale/dark purple
...            [[210,235,255],[ 58,134,237]],  #pale/dark blue
...            [[223,255,232],[ 61,189, 63]] ] #pale/dark green
>>> mapping_pastel = legs.PalObj(range_arr=[0.,60],
...                              color_arr=pastel,
...                              over_high='extend',
...                              under_low='white')
>>> #plot data & palette
>>> mapping_pastel.plot_data(ax=ax2,data=reflectivity_like,
...                          palette='right', pal_units='[dBZ]',
...                          pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/col_arr_demo.svg')
```

1.4.2 Categorical, quantitative color mappings

The keyword **solid** is used for generating categorical palettes.

```
>>> fig_w, fig_h = 11.6, 10.
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #mapping with solid dark colors
>>> x0, y0 = .5/fig_w , (.5+rec_h+sp_h)/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('Using default dark colors', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
>>> mapping_solid_dark = legs.PalObj(range_arr=[0.,60],
...                                  color_arr=['brown','orange','red'],
...                                  solid='col_dark',
...                                  over_high='extend',
...                                  under_low='white')
>>> mapping_solid_dark.plot_data(ax=ax1,data=reflectivity_like,
...                              palette='right', pal_units='[dBZ]',
...                              pal_format='{:2.0f}')
>>>
>>>
>>> #mapping with solid light colors
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , (.5+rec_h+sp_h)/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('Using default light colors', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> mapping_solid_light = legs.PalObj(range_arr=[0.,60],
```

(continues on next page)

(continued from previous page)

```
...                                  color_arr=['green','orange','purple'],
...                                  solid=    'col_light',
...                                  over_high='extend',
...                                  under_low='white')
>>> mapping_solid_light.plot_data(ax=ax2,data=reflectivity_like,
...                               palette='right', pal_units='[dBZ]',
...                               pal_format='{:2.0f}')
>>>
>>>
>>> #mapping with custom solid colors
>>> x0, y0 = (.5+rec_w/2.+sp_w/2.)/fig_w , .5/fig_h
>>> ax3 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax3.annotate('Using custom solid colors', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax3)
>>> #colors from www.colorbrewer2.org
>>> magenta =[ [253, 224, 239],  #pale magenta
...            [241, 182, 218],
...            [222, 119, 174],
...            [197,  27, 125],
...            [142,   1,  82] ] #dark magenta
>>> mapping_solid_custom = legs.PalObj(range_arr=[0.,60],
...                                    color_arr= magenta,
...                                    solid=    'supplied',
...                                    over_high='extend',
...                                    under_low='white')
>>> mapping_solid_custom.plot_data(ax=ax3,data=reflectivity_like,
...                                palette='right', pal_units='[dBZ]',
...                                pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/solid_demo.svg')
```

1.4.3 Divergent color mappings

The keyword **dark_pos** is useful for making divergent palettes.

```
>>> fig_w, fig_h = 11.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> # two color divergent mapping
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('Divergent mapping \nusing defaul colors', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
```

(continues on next page)

(continued from previous page)

```
>>> mapping_div_2_colors = legs.PalObj(range_arr=[-50.,50],
...                                    color_arr=['orange','blue'],
...                                    dark_pos =['low',   'high'],
...                                    over_under='extend')
>>> mapping_div_2_colors.plot_data(ax=ax1,data=reflectivity_like,
...                                palette='right', pal_units='[dBZ]',
...                                pal_format='{:2.0f}')
>>>
>>>
>>> #Custom pastel colors
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , .5/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('Divergent mapping with \ncustom color ', size=18,
...                    xy=(.17/rec_w, 3.35/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> pastel = [ [[255,255,255],[147, 78,172]],  # white, purple
...            [[255,255,255],[ 61,189, 63]] ] # white, green
>>> mapping_pastel = legs.PalObj(range_arr=[-50.,50],
...                              color_arr=pastel,
...                              dark_pos =['low','high'],
...                              over_under='extend')
>>> mapping_pastel.plot_data(ax=ax2,data=reflectivity_like,
...                          palette='right', pal_units='[dBZ]',
...                          pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/dark_pos_demo.svg')
```

Quantitative divergent color mappings can naturally be made using the **solid** keyword.

```
>>> fig_w, fig_h = 11.6, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #mapping with custom colors
>>> x0, y0 = (.5+rec_w/2.+sp_w/2.)/fig_w , .5/fig_h
>>> ax = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> format_axes(ax)
>>>
>>> # these colors were defined using inkscape.
>>> # www.colorbrewer2.org is also a great place for getting such color mappings
>>> green_purple =[ [114,  30, 179],  #dark purple
...                 [172,  61, 255],
...                 [210, 159, 255],
...                 [255, 215, 255],  #pale purple
...                 [255, 255, 255],  #white
...                 [218, 255, 207],  #pale green
...                 [162, 222, 134],
...                 [111, 184,   0],
...                 [  0, 129,   0] ] #dark green
```

(continues on next page)

(continued from previous page)

```
>>>
>>> mapping_divergent_solid = legs.PalObj(range_arr=[-60.,60],
...                                       color_arr= green_purple,
...                                       solid=    'supplied',
...                                       over_under='extend')
>>> mapping_divergent_solid.plot_data(ax=ax,data=reflectivity_like,
...                                   palette='right', pal_units='[dBZ]',
...                                   pal_format='{:2.0f}')
>>>
>>>
>>> plt.savefig('_static/divergent_solid.svg')
```

1.5 Colors legs covering unequal range intervals

So far, the keyword **range_arr** has been used to determine the range of the entire color mapping. It can also be used to define color legs with different extents.

```
>>>
>>> #ensure strictky +ve reflectivity values
>>> reflectivity_like_pve = np.where(reflectivity_like <= 0., 0., reflectivity_like)
>>> #convert reflectivity in dBZ to precipitation rates in mm/h (Marshall-Palmer, 1949)
>>> precip_rate =  10.**(reflectivity_like_pve/16.) / 27.424818
>>>
>>> fig_w, fig_h = 5.8, 5.#size of figure
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> #mapping with color legs spanning different ranges of values
>>> x0, y0 = .5/fig_w , .5/fig_h
>>> ax = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> format_axes(ax)
>>> mapping_diff_ranges = legs.PalObj(range_arr=[.1,3.,6.,12.,25.,50.,100.],
...                                   n_col=6,
...                                   over_high='extend',
...                                   under_low='white')
>>> # the keyword "equal_legs" makes the legs have equal space in the palette even
>>> # when they cover different value ranges
>>> mapping_diff_ranges.plot_data(ax=ax,data=precip_rate,
...                               palette='right', pal_units='[mm/h]',
...                               pal_format='{:2.0f}',
...                               equal_legs=True)
>>> plt.savefig('_static/different_ranges.svg')
```

1.6 Separate plotting of data and palette

When multiple pannels are plotted, it is often convenient to separate the display of data form that of the palette. In this example, two color mappings are used first separately and then together.

```
>>> fig_w, fig_h = 11.6, 10.
>>> fig = plt.figure(figsize=(fig_w, fig_h))
>>>
>>>
>>> # black and white mapping
>>> # without the **palette** keyword, **plot_data** only plots data.
>>> x0, y0 = .5/fig_w , (.5+rec_h+sp_h)/fig_h
>>> ax1 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax1.annotate('Black & white mapping', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax1)
>>> mapping_bw = legs.PalObj(range_arr=[0.,1.], color_arr='b_w')
>>> mapping_bw.plot_data(ax=ax1,data=gauss_bulge)
>>>
>>> #get RGB image using the to_rgb method
>>> gauss_rgb = mapping_bw.to_rgb(gauss_bulge)
>>>
>>>
>>> #color mapping using 6 default linear color segments
>>> # this time, data is plotted by hand
>>> x0, y0 = (.5+rec_w+sp_w)/fig_w , (.5+rec_h+sp_h)/fig_h
>>> ax2 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax2.annotate('6 Colors', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax2)
>>> mapping_ref = legs.PalObj(range_arr=[0.,60],n_col=6, over_under='extend')
>>> reflectivity_rgb = mapping_ref.to_rgb(reflectivity_like)
>>> x1, x2 = ax2.get_xlim()
>>> y1, y2 = ax2.get_ylim()
>>> dum = ax2.imshow(reflectivity_rgb, interpolation='nearest',
...                  origin='upper', extent=[x1,x2,y1,y2] )
>>> ax2.set_aspect('auto')  #force matplotlib to respect the axes that was defined
>>>
>>>
>>> #As a third panel, let's combine the two images
>>> x0, y0 = (.5+rec_w/2.)/fig_w , .5/fig_h
>>> ax3 = fig.add_axes([x0,y0,rec_w/fig_w,rec_h/fig_h])
>>> dum = ax3.annotate('combined image', size=18,
...                    xy=(.17/rec_w, 3.65/rec_h), xycoords='axes fraction',
...                    bbox=dict(boxstyle="round", fc='white', ec='white'))
>>> format_axes(ax3)
>>>
>>> #blend the two images by hand
>>> #image will be opaque where reflectivity > 0
>>> alpha = np.where(reflectivity_like >= 0., 1., 0.)
>>> alpha = np.where(np.logical_and(reflectivity_like >= 0., reflectivity_like < 10.), 0.
```

(continues on next page)

(continued from previous page)

```
↪1*reflectivity_like, alpha)
>>> combined_rgb = np.zeros(gauss_rgb.shape,dtype='uint8')
>>> for zz in np.arange(3):
...     combined_rgb[:,:,zz] = (1. - alpha)*gauss_rgb[:,:,zz] + alpha*reflectivity_rgb[:,
↪:,zz]
>>>
>>> #plot image w/ imshow
>>> x1, x2 = ax3.get_xlim()
>>> y1, y2 = ax3.get_ylim()
>>> dum = ax3.imshow(combined_rgb, interpolation='nearest',
...                  origin='upper', extent=[x1,x2,y1,y2])
>>> ax3.set_aspect('auto')
>>>
>>> #plot palettes with the plot_palette method
>>> pal_w  = .25/fig_w   #width of palette
>>> x0, y0 = x0+rec_w/fig_w+.25/fig_w  , .5/fig_h
>>> mapping_bw.plot_palette(pal_pos=[x0,y0,pal_w,rec_h/fig_h],
...                         pal_units='unitless',
...                         pal_format='{:2.0f}')
>>> x0, y0 = x0+1.2/fig_w  , .5/fig_h
>>> mapping_ref.plot_palette(pal_pos=[x0,y0,pal_w,rec_h/fig_h],
...                          pal_units='dBZ',
...                          pal_format='{:3.0f}')
>>>
>>>
>>> plt.savefig('_static/separate_data_palettes.svg')
```

CHAPTER

TWO

LEGS API

class domutils.legs.legs.**PalObj**(*range_arr=None*, *dark_pos=None*, *color_arr=None*, *n_col=None*, *map_f_arr=None*, *solid=None*, *over_high=None*, *under_low=None*, *over_under=None*, *excep_val=None*, *excep_tol=None*, *excep_col=None*)

A class for intuitive construction of color palettes

- Easy construction of qualitative, semi-quantitative, categorical and divergent color mappings
- Guarantees that a color is assigned for data values between -infinity and +infinity
- Explicit handling of colors at and/or exceeding end points
- Supports any number of exception values assigned to specific colors
- Encourages the user to specify units of data being depicted
- Consistency of the color palettes and colored representation of data is assured through the use of the same color mapping functions.

Parameters

- **range_arr** (Optional[List[float]]) – End points of the different color legs.

  Must be:

  1. A two element array-like describing the maximum and minimum data values represented by the color palette

     eg: `[0.,1]`

  2. A *n* element array-like describing the data values delimiting the bounds of each color leg.

     eg: `[0.,1.,10.,100, ...]`

     In this case, the dimension of *range_arr* should be the number of legs +1

- **color_arr** (Optional[Any]) – Defines the different colors to be used in a palette.

  For categorical palettes, it is used in conjunction with the *solid* keyword.

  It can be:

  1. A list of named colors

     eg: [“red”,”green”]

     for two semi-continuous color legs, one red and one green

  2. An array-like of rgb colors describing dark and light values of each color leg.

     eg: `[[[158, 0, 13],[255,190,187]],[[ 0,134, 0],[134,222,134]]]`

for two semi-continuous color legs, one red and one green (same as above) In this context, the number of elements in color_arr must be a multiple of 6 (two rgp value per legs)

3. For categorical color palettes *color_arr* must be an array-like of rgb colors describing solid colors for each color leg. This requires that the "solid" keyword be specified.

   a) With solid="col_dark" or solid="col_light" *color_arr* must be a list of strings of named colors

      eg: `["red","green"]`

      The dark or light shade of this color will be used for each leg.

   b) With solid="supplied" "color_arr" must be an array-like of rgb colors

      eg: `[[255,000,000],[000,255,000]]`

      for two color legs, one solid red, the other solid green.

- **solid** (`Optional[str]`) – A string set to `"supplied"`, `"col_dark"` or `"col_light"`. See the documentation of the *color_arr* argument for usage.
- **dark_pos** (`Optional[List[str]]`) – Defines whether the dark color is to be associated with high or low data value. It is only meaningful when constructing a continuous or semi-continuous palettes.
  - The default is to assign dark colors to the larger data values
  - If set to "high" or "low" all legs will be set to the high or low data value
  - If set to a list of strings,

    eg: `["high","low"]`,

    it will determine the position of the dark color for the individual color legs. In this case, the length of the list should be the same as the number of legs in the palette.
- **n_col** (`Optional[int]`) – Specifies how many color to display when using the default color sequence.
  - It must be convertible to an integer number.
  - It must be <= 8.
- **over_high** (`Optional[str]`) – Determines what color gets assigned to data values larger than the palette range.

  Accepted values are:

  1. `"exact"`: no data values expected beyond the range of the color mapping.. At the moment of applying the color mapping, an error will be raised if such data values are found.
  2. `"extend"`: The end-point color will be used for data beyond the range of the color mapping
  3. A named color:

     eg: `"red"`
  4. A rgb color:

     eg: `[ 0, 0,255]`
- **under_low** (`Optional[str]`) – Determines what color gets assigned to data values smaller than the palette range.

  Accepted values are:

1. "exact": no data values expected beyond the range of the color mapping.. At the moment of applying the color mapping, an error will be raised if such data values are found.
2. "extend": The end-point color will be used for data beyond the range of the color mapping
3. A named color:

   eg: "red"
4. A rgb color:

   eg: [0, 0,255]

- **over_under** (Optional[str]) – Shortcut to specify both *over_high* and *under_low* at the same time
- **excep_val** (Optional[List[float]]) – Data values to which special colors will be assigned.
- **excep_tol** (Optional[List[float]]) – Tolerance within which a data value specified in *excep_val* is considered an exception

  exception_range = *excep_val* +- *except_tol* inclusively

  – *excep_tol* must have the same dimension as "excep_val"
- **excep_col** (Optional[Any]) – Color(s) to be assigned to data value specified in *excep_val*

  Must be

  1. A string describing a named colors

     eg: "dark_red"

     This color will be assigned to all exceptions.
  2. A 1D list of named colors

     eg: ["dark_red" ,"dark_blue" , ... , "dark_green"]
  3. A 2D array-like of rgb colors

     eg: [[000,000,255],[000,255,000], ... , [000,000,255]]

  For cases 2 and 3, the number of colors represented must be equal to the number of exceptions provided in "excep_val".
- **map_f_arr** (Optional[Callable]) – Defines the name of the function that performs the mapping between data values and rgb values.

  1. If a string, the function will be used for all color legs of a palette.
  2. If a list of strings, it defines the mapping function for each individual legs. In this case, the dimension of *map_f_arr* should be equal to the number of color legs.

  – The default behavior is to use the "linmap" function. For linear interpolation of rgb between two predefiuned colors.

  – When the *solid* keyword is set, the "within" function is used instead.

Callable

alias of Callable

np = <module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/domutils/conda/latest/lib/python3.7/site-packages/numpy/__init__.py'>

plot_data(*ax*, *data*, *palette=None*, *zorder=None*, *aspect='auto'*, *rot90=None*, ***kwargs*)

Applies the mapping from data space to color space and plots the result into an existing axes.

Parameters

- **ax** (Any) – The matplotlib Axes object where imshow will plot the data
- **data** (ndarray) – Data to be plotted
- **palette** (Optional[str]) – Flag to show a palette beside the data axes.
- **aspect** (Optional[Any]) – aspect ratio, see documentation for axes.set_aspect()
- **rot90** (Optional[int]) – Number of counter-clockwise rotations to be applied to data before plotting
- **zorder** (Optional[int]) – Matplotlib zorder for the imshow call

plot_palette(*data_ax=None*, *equal_legs=None*, *pal_pos=None*, *pal_sp=0.1*, *pal_w=0.25*, *pal_units='Undefined Units'*, *pal_linewidth=0.3*, *pal_format='{:3.3f}'*)

plot a color palette near an existing axes where data is displayed

Parameters

- **data_ax** (Optional[Any]) – The matplotlib axes where the data is plotted. The palette will be plotted in an ax besides this Has no effect if **pal_pos** is provided.
- **equal_legs** (Optional[bool]) – Flag indicating that all color legs should have the same space in the palette.

  The default is to set the space proportional to the data interval spanned by individual color legs.
- **pal_pos** (Optional[Any]) – Position [x0,y0,width,height] for plotting palette.
- **pal_sp** (Optional[float]) – Space [inches] between figure and palette
- **pal_w** (Optional[float]) – Width [inches] of color palette
- **pal_units** (Optional[str]) – The units of the palette being shown.
- **pal_linewidth** (Optional[float]) – Width of the lines
- **pal_format** (Optional[str]) – Format string for the data values displayed beside tick-marks

to_rgb(*data_in*)

Applies the mapping from data space to color space and returns resulting rgb array

Parameters
data – Data to be plotted

Return type
ndarray

CHAPTER

THREE

COLOR MAPPING RESSOURCES

A good place for building sequential color palettes http://colorbrewer2.org

Up to 8% of men are color-blind to some extent source. Use this page to test your color mappings. https://www.color-blindness.com/coblis-color-blindness-simulator

Motivation for using something other than rainbow. https://journals.ametsoc.org/doi/full/10.1175/BAMS-D-13-00155.1

The same people now provide a HCL palette maker http://hclwizard.org:64230/hclwizard/

Pick colors from a color wheel. There are plenty like this one on the web. https://www.sessions.edu/color-calculator/

For easy construction of custom color mappings & palettes:

CHAPTER

FOUR

GEO_TOOLS API

A class for interpolating any data to any grids with Cartopy.

In addition to interpolation, this class is usefull to display geographical data. Since projection mappings need to be defined only once for a given dataset/display combination, multi-pannels figures can be made efficiently.

Most of the action happens through the class *ProjInds*.

The following figure illustrates the convention for storring data in numpy arrays. It is assumed that the first index (rows) represents x-y direction (longitudes):

class domutils.geo_tools.projinds.**ProjInds**(*data_xx=None*, *data_yy=None*, *src_lon=None*, *src_lat=None*, *dest_lon=None*, *dest_lat=None*, *extent=None*, *dest_crs=None*, *source_crs=None*, *fig=None*, *image_res=(400, 400)*, *extend_x=True*, *extend_y=True*, *average=False*, *smooth_radius=None*, *min_hits=1*, *missing=-9999.0*)

A class for making, keeping record of, and applying projection indices.

This class handles the projection of gridded data with lat/lon coordinates to cartopy's own data space in geoAxes.

Parameters

- **src_lon** (Optional[Any]) – (array like) longitude of data. Has to be same dimension as data.
- **src_lat** (Optional[Any]) – (array like) latitude of data. Has to be same dimension as data.
- **dest_lon** (Optional[Any]) – Longitude of destination grid on which the data will be re-gridded
- **dest_lat** (Optional[Any]) – Latitude of destination grid on which the data will be regrid-ded
- **extent** (Optional[Any]) – To be used together with dest_crs, to get destination grid from a cartopy geoAxes (array like) Bounds of geoAxes domain [lon_min, lon_max, lat_min, lat_max]
- **dest_crs** (Optional[Any]) – If provided, cartopy crs instance for the destination grid (nec-essary for plotting maps)
- **source_crs** (Optional[Any]) – Cartopy crs of the data coordinates. If None, a Geodetic crs is used for using latitude/longitude coordinates.
- **fig** (Optional[Any]) – Instance of figure currently being used. If None (the default) a new figure will be instantiated but never displayed.

- **image_res** (Optional[tuple]) – Pixel density of the figure being generated. eg (300,200) means that the image displayed on the geoAxes will be 300 pixels wide (i.e. longitude in the case of geo_axes) by 200 pixels high (i.e. latitude in the case of geo_axes).
- **extend_x** (Optional[Any]) – Set to True (default) when the destination grid is larger than the source grid. This option allows to mark no data points with Missing rather than extrapolating. Set to False for global grids.
- **extend_y** (Optional[Any]) – Same as extend_x but in the Y direction.
- **average** (Optional[Any]) – (bool) When true, all pts of a source grid falling within a destination grid pt will be averaged. Usefull to display/interpolate hi resolution data to a coarser grid. Weighted averages can be obtained by providing weights to the *project_data* method.
- **smooth_radius** (Optional[Any]) – Boxcar averaging with a circular area of a given radius. This option allows to perform smoothing at the same time as interpolation. For each point in the destination grid, all source data points within a radius given by *smooth_radius* will be averaged together.
- **min_hits** (Optional[Any]) – For use in conjunction with average or smooth_radius. This keyword specifies the minimum number of data points for an average to be considered valid.
- **missing** (Optional[float]) – Value which will be assigned to points outside of the data domain.

Example

Simple example showing how to project and display data. Rotation of points on the globe is also demonstrated

```
>>> import numpy as np
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
>>> import cartopy.crs as ccrs
>>> import cartopy.feature as cfeature
>>> import domutils.legs as legs
>>> import domutils.geo_tools as geo_tools
>>>
>>> # make mock data and coordinates
>>> # note that there is some regularity to the grid
>>> # but that it does not conform to any particular projection
>>> regular_lons =     [ [-91. , -91  , -91   ],
...                       [-90. , -90  , -90   ],
...                       [-89. , -89  , -89   ] ]
>>> regular_lats =     [ [ 44  ,  45  ,  46   ],
...                       [ 44  ,  45  ,  46   ],
...                       [ 44  ,  45  ,  46   ] ]
>>> data_vals =        [ [  6.5,   3.5,    .5 ],
...                       [  7.5,   4.5,   1.5 ],
...                       [  8.5,   5.5,   2.5 ] ]
>>> missing = -9999.
>>>
>>> #pixel resolution of image that will be shown in the axes
>>> img_res = (800,600)
>>> #point density for entire figure
>>> mpl.rcParams['figure.dpi'] = 800
```

(continues on next page)

(continued from previous page)

```
>>>
>>> #projection and extent of map being displayed
>>> proj_aea = ccrs.AlbersEqualArea(central_longitude=-94.,
...                                 central_latitude=35.,
...                                 standard_parallels=(30.,40.))
>>> map_extent=[-94.8,-85.2,43,48.]
>>>
>>> #---------------------------------------------------------------------
>>> #regular lat/lons are boring, lets rotate the coordinate system about
>>> # the central data point
>>>
>>> #use cartopy transforms to get xyz coordinates
>>> proj_ll = ccrs.Geodetic()
>>> geo_cent = proj_ll.as_geocentric()
>>> xyz = geo_cent.transform_points(proj_ll, np.asarray(regular_lons),
...                                         np.asarray(regular_lats))
>>>
>>> #lets rotate points by 45 degrees counter clockwise
>>> theta = np.pi/4
>>> rotation_matrix = geo_tools.rotation_matrix([xyz[1,1,0], #x
...                                              xyz[1,1,1], #y
...                                              xyz[1,1,2]],#z
...                                              theta)
>>> rotated_xyz = np.zeros((3,3,3))
>>> for ii, (lat_arr, lon_arr) in enumerate(zip(regular_lats, regular_lons)):
...     for jj, (this_lat, this_lon) in enumerate(zip(lat_arr, lat_arr)):
...         rotated_xyz[ii,jj,:] = np.matmul(rotation_matrix,[xyz[ii,jj,0], #x
...                                                           xyz[ii,jj,1], #y
...                                                           xyz[ii,jj,2]])#z
>>>
>>> #from xyz to lat/lon
>>> rotated_latlon = proj_ll.transform_points(geo_cent, rotated_xyz[:,:,0],
...                                                     rotated_xyz[:,:,1],
...                                                     rotated_xyz[:,:,2])
>>> rotatedLons = rotated_latlon[:,:,0]
>>> rotatedLats = rotated_latlon[:,:,1]
>>> # done rotating
>>> #---------------------------------------------------------------------
>>>
>>> #larger characters
>>> mpl.rcParams.update({'font.size': 15})
>>>
>>> #instantiate figure
>>> fig = plt.figure(figsize=(7.5,6.))
>>>
>>> #instantiate object to handle geographical projection of data
>>> # onto geoAxes with this specific crs and extent
>>> ProjInds = geo_tools.ProjInds(rotatedLons, rotatedLats,extent=map_extent, dest_
↪crs=proj_aea,
...                               image_res=img_res)
>>>
>>> #axes for this plot
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_axes([.01,.1,.8,.8], projection=proj_aea)
>>> ax.set_extent(map_extent)
>>>
>>> # Set up colormapping object
>>> color_mapping = legs.PalObj(range_arr=[0.,9.],
...                             color_arr=['brown','blue','green','orange',
...                                        'red','pink','purple','yellow','b_w'],
...                             solid='col_dark',
...                             excep_val=missing, excep_col='grey_220')
>>>
>>> #geographical projection of data into axes space
>>> proj_data = ProjInds.project_data(data_vals)
>>>
>>> #plot data & palette
>>> color_mapping.plot_data(ax=ax,data=proj_data,
...                         palette='right', pal_units='[unitless]',
...                         pal_format='{:4.0f}')   #palette options
>>>
>>> #add political boundaries
>>> dum = ax.add_feature(cfeature.STATES.with_scale('50m'),
...                      linewidth=0.5, edgecolor='0.2',zorder=1)
>>>
>>> #plot border and mask everything outside model domain
>>> ProjInds.plot_border(ax, mask_outside=False, linewidth=2.)
>>>
>>> #uncomment to save figure
>>> plt.savefig('_static/projection_demo.svg')
```

Example showing how ProjInds can also be used for nearest neighbor interpolation

```
>>> import numpy as np
>>>
>>> # Source data on a very simple grid
>>> src_lon =     [ [-90.1 , -90.1  ],
...                 [-89.1 , -89.1  ] ]
>>>
>>> src_lat =     [ [ 44.1  , 45.1  ],
...                 [ 44.1  , 45.1  ] ]
>>>
>>> data    =     [ [  3    ,  1    ],
...                 [  4    ,  2    ] ]
>>>
>>> # destination grid where we want data
>>> # Its larger than the source grid and slightly offset
>>> dest_lon =     [ [-91. , -91  , -91 , -91  ],
...                   [-90. , -90  , -90 , -90  ],
...                   [-89. , -89  , -89 , -89  ],
...                   [-88. , -88  , -88 , -88  ] ]
>>>
>>> dest_lat =     [ [ 43  ,  44  ,  45 ,  46 ],
...                   [ 43  ,  44  ,  45 ,  46 ],
```

(continues on next page)

(continued from previous page)

```
...                 [ 43  ,  44  ,  45 ,  46 ],
...                 [ 43  ,  44  ,  45 ,  46 ] ]
>>>
>>> #instantiate object to handle interpolation
>>> ProjInds = geo_tools.ProjInds(data_xx=src_lon,   data_yy=src_lat,
...                               dest_lon=dest_lon, dest_lat=dest_lat,
...                               missing=-99.)
>>> #interpolate data with "project_data"
>>> interpolated = ProjInds.project_data(data)
>>> #nearest neighbor output, pts outside the domain are set to missing
>>> #Interpolation with border detection in all directions
>>> print(interpolated)
[[-99. -99. -99. -99.]
 [-99.   3.   1. -99.]
 [-99.   4.   2. -99.]
 [-99. -99. -99. -99.]]
>>>
>>>
>>> #on some domain, border detection is not desirable, it can be turned off
>>> #
>>> # extend_x here refers to the dimension in data space (longitudes) that are
>>> # represented along rows of python array.
>>>
>>> # for example:
>>>
>>> # Border detection in Y direction (latitudes) only
>>> proj_inds_ext_y = geo_tools.ProjInds(data_xx=src_lon,   data_yy=src_lat,
...                                      dest_lon=dest_lon, dest_lat=dest_lat,
...                                      missing=-99.,
...                                      extend_x=False)
>>> interpolated_ext_y = proj_inds_ext_y.project_data(data)
>>> print(interpolated_ext_y)
[[-99.   3.   1. -99.]
 [-99.   3.   1. -99.]
 [-99.   4.   2. -99.]
 [-99.   4.   2. -99.]]
>>> #
>>> # Border detection in X direction (longitudes) only
>>> proj_inds_ext_x = geo_tools.ProjInds(data_xx=src_lon,   data_yy=src_lat,
...                                      dest_lon=dest_lon, dest_lat=dest_lat,
...                                      missing=-99.,
...                                      extend_y=False)
>>> interpolated_ext_x = proj_inds_ext_x.project_data(data)
>>> print(interpolated_ext_x)
[[-99. -99. -99. -99.]
 [  3.   3.   1.   1.]
 [  4.   4.   2.   2.]
 [-99. -99. -99. -99.]]
>>> #
>>> # no border detection
>>> proj_inds_no_b = geo_tools.ProjInds(data_xx=src_lon,   data_yy=src_lat,
...                                     dest_lon=dest_lon, dest_lat=dest_lat,
```

(continues on next page)

(continued from previous page)

```
...                                     missing=-99.,
...                                     extend_x=False, extend_y=False)
>>> interpolated_no_b = proj_inds_no_b.project_data(data)
>>> print(interpolated_no_b)
[[3. 3. 1. 1.]
 [3. 3. 1. 1.]
 [4. 4. 2. 2.]
 [4. 4. 2. 2.]]
```

Interpolation to coarser grids can be done with the *nearest* keyword. With this flag, all high-resoution data falling within a tile of the destination grid will be averaged together. Border detection works as in the example above.

```
>>> import numpy as np
>>> # source data on a very simple grid
>>> src_lon =     [ [-88.2 , -88.2  ],
...                 [-87.5 , -87.5  ] ]
>>>
>>> src_lat =     [ [ 43.5  , 44.1  ],
...                 [ 43.5  , 44.1  ] ]
>>>
>>> data    =     [ [  3    ,  1    ],
...                 [  4    ,  2    ] ]
>>>
>>> # coarser destination grid where we want data
>>> dest_lon =     [ [-92. , -92  , -92 , -92  ],
...                   [-90. , -90  , -90 , -90  ],
...                   [-88. , -88  , -88 , -88  ],
...                   [-86. , -86  , -86 , -86  ] ]
>>>
>>> dest_lat =     [ [ 42  ,  44  ,  46 ,  48 ],
...                   [ 42  ,  44  ,  46 ,  48 ],
...                   [ 42  ,  44  ,  46 ,  48 ],
...                   [ 42  ,  44  ,  46 ,  48 ] ]
>>>
>>> #instantiate object to handle interpolation
>>> #Note the average keyword set to true
>>> ProjInds = geo_tools.ProjInds(data_xx=src_lon,   data_yy=src_lat,
...                               dest_lon=dest_lon, dest_lat=dest_lat,
...                               average=True, missing=-99.)
>>>
>>> #interpolate data with "project_data"
>>> interpolated = ProjInds.project_data(data)
>>>
>>> #Since all high resolution data falls into one of the output
>>> #grid tile, they are all aaveraged togetherL:  (1+2+3+4)/4 = 2.5
>>> print(interpolated)
[[-99.  -99.  -99.  -99. ]
 [-99.  -99.  -99.  -99. ]
 [-99.    2.5 -99.  -99. ]
 [-99.  -99.  -99.  -99. ]]
>>>
>>> #weighted average can be obtained by providing weights for each data pt
```

(continues on next page)

(continued from previous page)

```
>>> #being averaged
>>> weights   =     [ [  0.5   ,  1.    ],
...                   [  1.    ,  0.25  ] ]
>>>
>>> weighted_avg = ProjInds.project_data(data, weights=weights)
>>> #result is a weighted average:
>>> # (1.*1 + 0.25*2 + 0.5*3 + 1.*4) / (1.+0.25+0.5+1.) = 7.0/2.75 = 2.5454
>>> print(weighted_avg)
[[-99.         -99.         -99.         -99.        ]
 [-99.         -99.         -99.         -99.        ]
 [-99.           2.54545455 -99.         -99.        ]
 [-99.         -99.         -99.         -99.        ]]
```

Sometimes, it is useful to smooth data during the interpolation process. For example when comparing radar measurement against model output, smoothing can be used to remove the small scales present in the observations but that the model cannot represent.

Use the *smoooth_radius* to average all source data point within a certain radius (in km) of the destination grid tiles.

```
>>> # source data on a very simple grid
>>> src_lon =     [ [-88.2 , -88.2  ],
...                 [-87.5 , -87.5  ] ]
>>>
>>> src_lat =     [ [ 43.5  , 44.1  ],
...                 [ 43.5  , 44.1  ] ]
>>>
>>> data    =     [ [  3    ,  1    ],
...                 [  4    ,  2    ] ]
>>>
>>> # coarser destination grid where we want data
>>> dest_lon =     [ [-92. , -92  , -92 , -92  ],
...                 [-90. , -90  , -90 , -90  ],
...                 [-88. , -88  , -88 , -88  ],
...                 [-86. , -86  , -86 , -86  ] ]
>>>
>>> dest_lat =     [ [ 42  ,  44  ,  46 ,  48 ],
...                 [ 42  ,  44  ,  46 ,  48 ],
...                 [ 42  ,  44  ,  46 ,  48 ],
...                 [ 42  ,  44  ,  46 ,  48 ] ]
>>>
>>> #instantiate object to handle interpolation
>>> #All source data points found within 300km of each destination
>>> #grid tiles will be averaged together
>>> ProjInds = geo_tools.ProjInds(data_xx=src_lon,    data_yy=src_lat,
...                               dest_lat=dest_lat,  dest_lon=dest_lon,
...                               smooth_radius=300., missing=-99.)
>>>
>>> #interpolate and smooth data with "project_data"
>>> interpolated = ProjInds.project_data(data)
>>>
>>> #output is smoother than data source
>>> print(interpolated)
```

(continues on next page)

(continued from previous page)

```
[[-99.         -99.         -99.         -99.        ]
 [  2.66666667   2.5          1.5        -99.        ]
 [  2.5          2.5          2.5        -99.        ]
 [  2.5          2.5          1.5        -99.        ]]
```

plot_border(*ax=None*, *crs=None*, *linewidth=0.5*, *zorder=3*, *mask_outside=False*)

Plot border of a more or less regularly gridded domain

Optionally, mask everything that is outside the domain for simpler figures.

Parameters

- **ax** (Optional[Any]) – Instance of a cartopy geoAxes where border is to be plotted
- **crs** (Optional[Any]) – Cartopy crs of the data coordinates. If None, a Geodetic crs is used for using latitude/longitude coordinates.
- **linewidth** (Optional[float]) – Width of border being plotted
- **mask_outside** (Optional[bool]) – If True, everything outside the domain will be hidden
- **zorder** (Optional[int]) – zorder for the mask being applied with maskOutside=True. This number should be larger than any other zorder in the axes.

Returns

Nothing

project_data(*data*, *weights=None*, *output_avg_weights=False*, *missing_v=-9999.0*)

Project data into geoAxes space

Parameters

- **data** (Any) – (array like), data values to be projected. Must have the same dimension as the lat/lon coordinates used to instantiate the class.
- **weights** (Optional[Any]) – (array like), weights to be applied during averaging (see average and smooth_radius keyword in ProjInds)
- **output_avg_weights** (Optional[bool]) – If True, projected_weights will be outputed along with projected_data "Averaged" and "smooth_radius" interpolation can take a long time on large grids. This option is useful to get radar data + quality index interpolated using one call to project_data instead of two halving computation time. Not available for nearest-neighbor interpolation which is already quite fast anyway.
- **missing_v** (Optional[float]) – Value in input data that will be neglected during the averaging process Has no impact unless *average* or *smooth_radius* are used in class instantiation.

Returns

projected_data A numpy array of the same shape as destination grid used to instantiate the class

if output_avg_weights=True, returns:

projected_data, projected_weights

domutils.geo_tools.lat_lon_extend.**lat_lon_extend**(*lon1_in*, *lat1_in*, *lon2_in*, *lat2_in*, *crs=None*, *half_dist=False*, *predefined_theta=None*)

Extends latitude and longitudes on a sphere

Given two points (pt1 and pt2) on a sphere, this function returns a third point (pt3) at the same distance and in the same direction as between pt1 and pt2. All points are defined with latitudes and longitudes, same altitude is assumed.

The diagram below hopefully make the purpose of this function clearer:

```
#   pt3              output                              #
#     ^                                                  #
#     \                                                 #
#      \               -> output with half_dist=True     #
#       \                                                #
#        pt2           input 2                           #
#         ^                                              #
#          \                                             #
#           \                                            #
#            \                                           #
#             pt1       input 1                          #
```

Parameters

- **lon1_in** (Any) – (array like) longitude of point 1
- **lat1_in** (Any) – (array like) latitude of point 1
- **lon2_in** (Any) – (array like) longitude of point 2
- **lat2_in** (Any) – (array like) latitude of point 2
- **crs** (Optional[Any]) – Instance of Cartopy geoAxes in which pt coordinates are defined (default is PlateCarree)
- **half_dist** (bool) – If true, pt 3 will be located at half the distance between pt1 and pt2
- **predefined_theta** – (array like) radians. If set, will determine distance between pt2 and pt3

Returns

(longitude, latitude) (array like) of extended point 3

Examples

Extend one point.

```
>>> import numpy as np
>>> import domutils.geo_tools as geo_tools
>>> # coordinates of pt1
>>> lat1 = 0.; lon1 = 0.
>>> # coordinates of pt2
>>> lat2 = 0.; lon2 = 90.
>>> # coordinates of extended point
>>> lon3, lat3 = geo_tools.lat_lon_extend(lon1,lat1,lon2,lat2)
>>> print(lon3, lat3)
[180.] [0.]
```

Extend a number of points all at once

```
>>> # coordinates for four pt1
>>> lat1 = [ 0., 0., 0., 0. ]
>>> lon1 = [ 0., 0., 0., 0. ]
>>> # coordinates for four pt2
>>> lat2 = [ 0.,90., 0., 30.]
>>> lon2 = [90., 0.,45., 0. ]
>>> # coordinates of extended points
>>> lon3, lat3 = geo_tools.lat_lon_extend(lon1,lat1,lon2,lat2)
>>> with np.printoptions(precision=1,suppress=True):
...       print(lon3)
...       print(lat3)
[180. 180.  90.   0.]
[ 0.   0.   0.  59.8]
>>> #           ^
>>> #           Presumably not =60. because cartopy uses a spheroid
```

domutils.geo_tools.lat_lon_range_az.**lat_lon_range_az**(*lon1_in*, *lat1_in*, *range_in*, *azimuth_in*, *crs=None*, *earth_radius=None*)

Computes lat/lon of a point at a given range and azimuth (meteorological angles)

Given a point (pt1) on a sphere, this function returns the lat/lon of a second point (pt2) at a given range and azimuth from pt1. Same altitude is assumed. Range is a distance following curvature of the earth

The diagram below hopefully make the purpose of this function clearer:

```
#                                                 #
#       N                                         #
#                                                 #
#       ^  -                                      #
#       |      -                                  #
#       |        -    azimuth (deg)               #
#       |          -                              #
#       |           -                             #
#      pt1           -  E                         #
#        \          -                             #
#  range  \        -                              #
#          \      -                                #
#           \  -                                  #
#           pt2    output is lat lon of pt2        #
```

Internally, two rotations on the sphere are conducted

- 1st rotation shifts pt1 toward the North and is dictated by range
- 2nd rotation is around point pt1 and is determined by azimuth

Input arrays will be broadcasted together.

Parameters

- **lon1_in** (Any) – (array like) longitude of point 1
- **lat1_in** (Any) – (array like) latitude of point 1
- **range_in** (Any) – (array like) [km] range (distance) at which pt2 is located from pt1

- **azimuth_in** (Any) – (array like) [degrees] direction (meteorological angle) in thich pt2 is located
- **crs** (Optional[Any]) – Instance of Cartopy geoAxes in which pt coordinates are defined (default is Geodesic for lat/lon specifications)
- **earth_radius** (Optional[Any]) – (km) radius of the earth (default 6371 km)

Returns

longitude, latitude (array like) of pt2

Examples

Extend one point.

```
>>> import numpy as np
>>> import domutils.geo_tools as geo_tools
>>> # coordinates of pt1
>>> lat1 = 0.
>>> lon1 = 0.
>>> # range and azimuth
>>> C = 2.*np.pi*6371.  #circumference of the earth
>>> range_km = C/8.
>>> azimuth_deg = 0.
>>> # coordinates of extended point
>>> lon2, lat2 = geo_tools.lat_lon_range_az(lon1,lat1,range_km,azimuth_deg)
>>> #   print(lon2, lat2) should give approximately
>>> #   0.0 45.192099833854435
>>> print(np.allclose((lon2, lat2), (0.0, 45.192099833854435)))
True
>>>  #Also works for inputs arrays
>>> lat1        = [[0.,     0],
...                [0.,     0]]
>>> lon1        = [[0.,     0],
...                [0.,     0]]
>>> range_km    = [[C/8.,  C/8.],
...                [C/8.,  C/8.]]
>>> azimuth_deg = [[0.,     90.],
...                [-90.,   180]]
>>> lon2, lat2 = geo_tools.lat_lon_range_az(lon1,lat1,range_km,azimuth_deg)
>>> print(np.allclose(lon2, np.array([[ 0.0000000e+00,  4.5000000e+01],
...                                   [-4.5000000e+01,  7.0167093e-15]])))
True
```

Since arrays are broadcasted together, inputs that consists of repeated values can be passed as floats. Here, we get the lat/lon along a circle 50km from a reference point

```
>>> #longitudes and latitudes
>>> #of points along a circle of 50 km radius around a point at 37 deg N, 88 deg␣
↪Ouest
>>> #note how only azimuths is an array
>>> lat0 =  37.
>>> lon0 = -88.
>>> ranges    = 50.
```

(continues on next page)

(continued from previous page)

```
>>> azimuths  = np.arange(0, 360, 10, dtype='float')
>>> circle_lons, circle_lats = geo_tools.lat_lon_range_az(lat0, lon0, ranges,␣
↪azimuths)
>>> print(circle_lons)
[37.         38.82129197 40.61352908 42.34696182 43.99045278 45.51079424
 46.8720648  48.03508603 48.9570966  49.59184727 49.89044518 49.80343238
 49.28472885 48.29808711 46.82635859 44.88285245 42.52223827 39.8463912
 37.         34.1536088  31.47776173 29.11714755 27.17364141 25.70191289
 24.71527115 24.19656762 24.10955482 24.40815273 25.0429034  25.96491397
 27.1279352  28.48920576 30.00954722 31.65303818 33.38647092 35.17870803]
>>> print(circle_lats)
[-87.55334031 -87.55889412 -87.57546173 -87.60276046 -87.64031502
 -87.68745105 -87.74328587 -87.80671606 -87.87640222 -87.95075157
 -88.02790057 -88.10570197 -88.18172472 -88.25328003 -88.31749246
 -88.37143711 -88.4123562  -88.43794478 -88.44665642 -88.43794478
 -88.4123562  -88.37143711 -88.31749246 -88.25328003 -88.18172472
 -88.10570197 -88.02790057 -87.95075157 -87.87640222 -87.80671606
 -87.74328587 -87.68745105 -87.64031502 -87.60276046 -87.57546173
 -87.55889412]
```

domutils.geo_tools.rotation_matrix.**rotation_matrix**(*axis*, *theta*)

Rotation matrix for counterclockwise rotation on a sphere

This formulation was found on SO https://stackoverflow.com/questions/6802577/rotation-of-3d-vector Thanks, "unutbu"

Parameters

- **axis** (Any) – Array-Like containing i,j,k coordinates that defines an axis of rotation starting from the center of the sphere (0,0,0)
- **theta** (float) – The amount of rotation (radians) desired.

Returns

A 3x3 matrix (numpy array) for performing the rotation

Example

```
>>> import numpy as np
>>> import domutils.geo_tools as geo_tools
>>> #a rotation axis [x,y,z] pointing straight up
>>> axis = [0.,0.,1.]
>>> # rotation of 45 degrees = pi/4 ratians
>>> theta = np.pi/4.
>>> #make rotation matrix
>>> mm = geo_tools.rotation_matrix(axis, theta)
>>> print(mm.shape)
(3, 3)
>>> # a point on the sphere [x,y,z] which will be rotated
>>> origin = [1.,0.,0.]
>>> #apply rotation
>>> rotated = np.matmul(mm,origin)
>>> #rotated coordinates
```

(continues on next page)

(continued from previous page)

```
>>> print(rotated)
[0.70710678 0.70710678 0.        ]
```

domutils.geo_tools.rotation_matrix.**rotation_matrix_components**(*axis*, *theta*)

Rotation matrix for counterclockwise rotation on a sphere

This version returns only the components of the rotation matrix(ces) This allows to use numpy array operations to simultaneously perform many application of these rotation matrices

Parameters

- **axis** (Any) – Array-Like containing i,j,k coordinates that defines an axes of rotation starting from the center of the sphere (0,0,0) dimension of array must be (m,3) for m points each defined by i,j,k components
- **theta** (float) – The amount of rotation (radians) desired. shape = (m,)

Returns

A, B, C, D, E, F, G, H, I

```
| A B C |
| D E F |
| G H I |
```

Each of the components above will be of shape (m,)

Example

```
>>> import numpy as np
>>> import domutils.geo_tools as geo_tools
>>> #four rotation axes [x,y,z]; two pointing up, two pointing along y axis
>>> axes = [[0.,0.,1.], [0.,0.,1.], [0.,1.,0.], [0.,1.,0.]]
>>> # first and third points will be rotated by 45 degrees (pi/4) second and fourth␣
↪by 90 degrees (pi/2)
>>> thetas = [np.pi/4., np.pi/2., np.pi/4., np.pi/2.]
>>> #make rotation matrices
>>> a, b, c, d, e, f, g, h, i = geo_tools.rotation_matrix_components(axes, thetas)
>>> #Four rotation matrices are here generated
>>> print(a.shape)
(4,)
>>> # points on the sphere [x,y,z] which will be rotated
>>> # here we use the same point for simplicity but any points can be used
>>> oi,oj,ok = np.array([[1.,0.,0.],[1.,0.,0.],[1.,0.,0.],[1.,0.,0.]]).T #transpose␣
↪to unpack columns
>>> #i, j and k components of each position vector
>>> print(oi.shape)
(4,)
>>> #apply rotation by hand (multiplication of each point (i,j,k vector) by␣
↪rotation matrix)
>>> rotated = np.array([a*oi+b*oj+c*ok, d*oi+e*oj+f*ok, g*oi+h*oj+i*ok]).T
>>> print(rotated.shape)
(4, 3)
>>> print(np.allclose(rotated, np.array([[ 7.07106781e-01,  7.07106781e-01,  0.
```

(continues on next page)

(continued from previous page)

```
↪00000000e+00],
...                                 [ 2.22044605e-16,  1.00000000e+00,  0.
↪00000000e+00],
...                                 [ 7.07106781e-01,  0.00000000e+00, -7.
↪07106781e-01],
...                                 [ 2.22044605e-16,  0.00000000e+00, -1.
↪00000000e+00]])))
True
```

For handling geographical projections of data with arbitrary lat/lon coordinates.

CHAPTER

FIVE

RADAR_TOOLS DEMO

The module radar_tools allows to access radar mosaics in various formats and process the data in different ways. Data from different sources is obtained transparently and various interpolation and filtering options are provided.

The radar mosaics are accessed via two functions:

- *get_instantaneous* for instantaneous reflectivity or precipitation rates
- *get_accumulations* for precipitation accumulations

These two functions are very similar, *get_accumulations* itself calling *get_instantaeous* repeatedly in order to construct accumulations on the fly.

This tutorial demonstrates the different operations that can be performed.

5.1 Tutorial setup

This section defines plotting functions and objects that will be used throughout this tutorial. You can skip to the next section for the radar related stuff.

Here we setup a function to display images from the data that we will be reading:

```
>>> def plot_img(fig_name, title, units, data, latitudes, longitudes,
...              colormap, equal_legs=False ):
...
...     import matplotlib.pyplot as plt
...     import cartopy.crs as ccrs
...     import cartopy.feature as cfeature
...     import domutils.geo_tools as geo_tools
...
...     #pixel density of image to plot
...     ratio = .8
...     hpix = 600.       #number of horizontal pixels
...     vpix = ratio*hpix #number of vertical pixels
...     img_res = (int(hpix),int(vpix))
...
...     #size of image to plot
...     fig_w = 7.                     #size of figure
...     fig_h = 5.5                     #size of figure
...     rec_w = 5.8/fig_w               #size of axes
...     rec_h = ratio*(rec_w*fig_w)/fig_h #size of axes
...
...     #setup cartopy projection
```

(continues on next page)

(continued from previous page)

```
...     central_longitude=-94.
...     central_latitude=35.
...     standard_parallels=(30.,40.)
...     proj_aea = ccrs.AlbersEqualArea(central_longitude=central_longitude,
...                                     central_latitude=central_latitude,
...                                     standard_parallels=standard_parallels)
...     map_extent=[-104.8,-75.2,27.8,48.5]
...
...     #instantiate projection object
...     proj_inds = geo_tools.ProjInds(src_lon=longitudes, src_lat=latitudes,
...                                    extent=map_extent, dest_crs=proj_aea, image_
↪res=img_res)
...     #use it to project data to image space
...     projected_data = proj_inds.project_data(data)
...
...     #instantiate figure
...     fig = plt.figure(figsize=(fig_w,fig_h))
...
...     #axes for data
...     x0 = 0.01
...     y0 = .1
...     ax1 = fig.add_axes([x0,y0,rec_w,rec_h], projection=proj_aea)
...     ax1.set_extent(map_extent)
...
...     #add title
...     dum = ax1.annotate(title, size=20,
...                        xy=(.02, .9), xycoords='axes fraction',
...                        bbox=dict(boxstyle="round", fc='white', ec='white'))
...
...     #plot data & palette
...     colormap.plot_data(ax=ax1,data=projected_data,
...                        palette='right', pal_units=units, pal_format='{:2.0f}',
...                        equal_legs=equal_legs)
...
...     #add political boundaries
...     ax1.add_feature(cfeature.STATES.with_scale('50m'), linewidth=0.5, edgecolor='0.2
↪')
...
...     #plot border
...     #proj_inds.plot_border(ax1, linewidth=.5)
...
...     #save output
...     plt.savefig(fig_name,dpi=400)
...     plt.close(fig)
```

Let's also initialize some color mapping objects for the different quantities that will be displayed (see *Legs Tutorial* for details).

```
>>> import domutils.legs as legs
>>>
>>> #flags
>>> undetect = -3333.
```

(continues on next page)

(continued from previous page)

```
>>> missing  = -9999.
>>>
>>> #Color mapping object for reflectivity
>>> ref_color_map = legs.PalObj(range_arr=[0.,60.],
...                             n_col=6,
...                             over_high='extend', under_low='white',
...                             excep_val=[undetect,missing], excep_col=['grey_200',
↪'grey_120'])
>>>
>>> #Color mapping object for quality index
>>> pastel = [ [[255,190,187],[230,104, 96]],  #pale/dark red
...            [[255,185,255],[147, 78,172]],  #pale/dark purple
...            [[255,227,215],[205,144, 73]],  #pale/dark brown
...            [[210,235,255],[ 58,134,237]],  #pale/dark blue
...            [[223,255,232],[ 61,189, 63]] ] #pale/dark green
>>> #precip Rate
>>> ranges = [.1,1.,2.,4.,8.,16.,32.]
>>> pr_color_map = legs.PalObj(range_arr=ranges,
...                            n_col=6,
...                            over_high='extend', under_low='white',
...                            excep_val=[undetect,missing], excep_col=['grey_200','grey_
↪120'])
>>> #accumulations
>>> ranges = [1.,2.,5.,10., 20., 50., 100.]
>>> acc_color_map = legs.PalObj(range_arr=ranges,
...                             n_col=6,
...                             over_high='extend', under_low='white',
...                             excep_val=[undetect,missing], excep_col=['grey_200',
↪'grey_120'])
```

5.2 Get radar mosaics from different sources and file formats

5.2.1 Baltrad ODIM H5

Let's read reflectivity fields from an ODIM H5 composite file using the *get_instantaneous* method.

```
>>> import os, inspect
>>> import datetime
>>> import domutils.radar_tools as radar_tools
>>>
>>> #when we want data
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>>
>>> #where is the data
>>> currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
>>> parentdir  = os.path.dirname(currentdir) #directory where this package␣
↪lives
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>>
```

(continues on next page)

(continued from previous page)

```
>>> #how to construct filename.
>>> #   See documentation for the *strftime* method in the datetime module
>>> #   Note the *.h5* extention, this is where we specify that we want ODIM␣
↪H5 data
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>>
>>> #get reflectivity on native grid
>>> #with latlon=True, we will also get the data coordinates
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True)
>>> #show what we just got
>>> for key in dat_dict.keys():
...     if key == 'valid_date':
...         print(key,dat_dict[key])
...     else:
...         print(key,dat_dict[key].shape)
reflectivity (2882, 2032)
total_quality_index (2882, 2032)
valid_date 2019-10-31 16:30:00+00:00
latitudes (2882, 2032)
longitudes (2882, 2032)
>>>
>>> #show data
>>> fig_name ='_static/original_reflectivity.svg'
>>> title = 'Odim H5 reflectivity on original grid'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

Dark grey represents missing values, light grey represent the *undetect* value.

5.2.2 MRMS precipitation rates in grib2 format

Reading precipitation rates from MRMS is done in a very similar way with the *get_instantaneous* method.

```
>>> import os, inspect
>>> import datetime
>>> import domutils.radar_tools as radar_tools
>>>
>>> #when we want data
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>>
>>> #where is the data
>>> currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
```

(continues on next page)

(continued from previous page)

```
↪currentframe())))
>>> parentdir  = os.path.dirname(currentdir) #directory where this package␣
↪lives
>>> data_path = parentdir + '/test_data/mrms_grib2/'
>>>
>>> #how to construct filename.
>>> #   See documentation for the *strftime* method in the datetime module
>>> #   Note the *.grib2* extention, this is where we specify that we wants␣
↪mrms data
>>> data_recipe = 'PrecipRate_00.00_%Y%m%d-%H%M%S.grib2'
>>>
>>> #Notre that the file RadarQualityIndex_00.00_20191031-163000.grib2.gz must␣
↪be present in the
>>> #same directory for the quality index to be defined.
>>>
>>> #get precipitation on native grid
>>> #with latlon=True, we will also get the data coordinates
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          desired_quantity='precip_rate',
...                                          latlon=True)
>>> #show what we just got
>>> for key in dat_dict.keys():
...     if key == 'valid_date':
...         print(key,dat_dict[key])
...     else:
...         print(key,dat_dict[key].shape)
precip_rate (3500, 7000)
total_quality_index (3500, 7000)
valid_date 2019-10-31 16:30:00
latitudes (3500, 7000)
longitudes (3500, 7000)
>>>
>>> #show data
>>> fig_name ='_static/mrms_precip_rate.svg'
>>> title = 'MRMS precip rates on original grid'
>>> units = '[mm/h]'
>>> data       = dat_dict['precip_rate']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

5.2.3 4-km mosaics from URP

Reading URP reflectivity mosaics is only a matter of changing the file extension to:

- .fst
- .std
- .stnd
- or no extension at all.

The script searches for *RDBZ* and *L1* entries in the standard file. The first variable that is found is returned. The quality index is set to 1 wherever the data is not flagged as missing in the standard file.

```
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> #URP 4km reflectivity mosaics
>>> data_path = parentdir + '/test_data/std_radar_mosaics/'
>>> #note the *.stnd* extension specifying that a standard file will be read
>>> data_recipe = '%Y%m%d%H_%Mref_4.0km.stnd'
>>>
>>> #exactly the same command as before
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True)
>>> for key in dat_dict.keys():
...     if key == 'valid_date':
...         print(key,dat_dict[key])
...     else:
...         print(key,dat_dict[key].shape)
reflectivity (1650, 1500)
total_quality_index (1650, 1500)
valid_date 2019-10-31 16:30:00+00:00
latitudes (1650, 1500)
longitudes (1650, 1500)
>>>
>>> #show data
>>> fig_name ='_static/URP4km_reflectivity.svg'
>>> title = 'URP 4km reflectivity on original grid'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

5.3 Get the nearest radar data to a given date and time

Getting the nearest radar data to an arbitrary validity time is convenient for comparison with model outputs at higher temporal resolutions.

By default, *get_instantaneous* returns None if the file does not exist at the specified time.

```
>>> #set time at 16h35 where no mosaic file exists
>>> this_date = datetime.datetime(2019, 10, 31, 16, 35, 0)
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe)
>>> print(dat_dict)
None
```

Set the *nearest_time* keyword to the temporal resolution of the data to rewind time to the closest available mosaic.

```
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          nearest_time=10)
>>> #note how the valid_date is different from the one that was requested
>>> #in the function call
>>> print(dat_dict['valid_date'])
2019-10-31 16:30:00+00:00
```

5.4 Get precipitation rates (in mm/h) from reflectivity (in dBZ)

By default, exponential drop size distributions are assumed with

$$Z = aR^b$$

in linear units. The default is to use WDSSR's relation with a=300 and b=1.4.

```
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>>
>>> #require precipitation rate in the output
>>> dat_dict = radar_tools.get_instantaneous(desired_quantity='precip_rate',
...                                          valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True)
>>>
>>> #show data
>>> fig_name ='_static/odimh5_reflectivity_300_1p4.svg'
>>> title = 'precip rate with a=300, b=1.4 '
>>> units = '[mm/h]'
```

(continues on next page)

(continued from previous page)

```
>>> data       = dat_dict['precip_rate']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

Different Z-R relationships can be used by specifying the a and b coefficients explicitly (for example, for the Marshall-Palmer DSD, a=200 and b=1.6):

```
>>> #custom coefficients a and b
>>> dat_dict = radar_tools.get_instantaneous(desired_quantity='precip_rate',
...                                          coef_a=200, coef_b=1.6,
...                                          valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True)
>>>
>>> #show data
>>> fig_name ='_static/odimh5_reflectivity_200_1p6.svg'
>>> title = 'precip rate with a=200, b=1.6 '
>>> units = '[mm/h]'
>>> data       = dat_dict['precip_rate']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

5.5 Apply a median filter to reduce speckle (noise)

Baltrad composites are quite noisy. For some applications, it may be desirable to apply a median filter to reduce speckle. This is done using the *median_filt* keyword. The filtering is applied both to the reflectivity or rain rate data and to its accompanying quality index.

```
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>>
>>> #Apply median filter by setting *median_filt=3* meaning that a 3x3 boxcar
>>> #will be used for the filtering
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True,
...                                          median_filt=3)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> #show data
>>> fig_name ='_static/speckle_filtered_reflectivity.svg'
>>> title = 'Skpeckle filtered Odim H5 reflectivity'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>>
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

5.6 Interpolation to a different grid

Interpolation to a different output grid can be done using the *dest_lat* and *dest_lon* keywords.

Three interpolation methods are supported:

- Nearest neighbor (default)
- Average all input data points falling within the output grid tile. This option tends to be slow.
- Average all input within a certain radius of the center of the output grid tile. This allows to perform smoothing at the same time as interpolation.

```
>>> import pickle
>>> #let our destination grid be at 10 km resolution in the middle of the US
>>> #this is a grid where I often perform integration with the GEM atmospheric␣
↪model
>>> #recover previously prepared data
>>> with open(parentdir + '/test_data/pal_demo_data.pickle', 'rb') as f:
...     data_dict = pickle.load(f)
>>> gem_lon = data_dict['longitudes']    #2D longitudes [deg]
>>> gem_lat = data_dict['latitudes']     #2D latitudes  [deg]
```

5.6.1 Nearest neighbor

```
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> #get data on destination grid using nearest neighbor
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True,
...                                          dest_lon=gem_lon,
...                                          dest_lat=gem_lat)
>>>
>>> #show data
>>> fig_name ='_static/nearest_interpolation_reflectivity.svg'
```

(continues on next page)

(continued from previous page)

```
>>> title = 'Nearest Neighbor to 10 km grid'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

5.6.2 Average all inputs falling within a destination grid tile

```
>>> #get data on destination grid using averaging
>>> this_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>> dat_dict = radar_tools.get_instantaneous(valid_date=this_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True,
...                                          dest_lon=gem_lon,
...                                          dest_lat=gem_lat,
...                                          average=True)
>>>
>>> #show data
>>> fig_name ='_static/average_interpolation_reflectivity.svg'
>>> title = 'Average to 10 km grid'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

5.6.3 Average all inputs within a radius

This method allows to smooth the data at the same time as it is interpolated.

```
>>> #get data on destination grid averaging all points
>>> #within a circle of a given radius
>>> #also apply the median filter on input data
>>> end_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>> dat_dict = radar_tools.get_instantaneous(valid_date=end_date,
...                                          data_path=data_path,
...                                          data_recipe=data_recipe,
...                                          latlon=True,
```

(continues on next page)

(continued from previous page)

```
...                                             dest_lon=gem_lon,
...                                             dest_lat=gem_lat,
...                                             median_filt=3,
...                                             smooth_radius=12.)
>>>
>>> #show data
>>> fig_name ='_static/smooth_radius_interpolation_reflectivity.svg'
>>> title = 'Average input within a radius of 12 km'
>>> units = '[dBZ]'
>>> data       = dat_dict['reflectivity']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          ref_color_map)
```

5.7 On-the-fly computation of precipitation accumulations

Use the *get_accumulation* method to get accumulations of precipitation. Three quantities can be outputted:

- *accumulation* The default option; returns the amount of water (in mm);
- *avg_precip_rate* For average precipitation rate (in mm/h) during the accumulation period;
- *reflectivity* For the reflectivity (in dBZ) associated with the average precipitation rate during the accumulation period;

For this example, let's get the accumulated amount of water in mm during a period of one hour.

```
>>> #1h accumulations of precipitation
>>> end_date = datetime.datetime(2019, 10, 31, 16, 30, 0)
>>> duration = 60.  #duration of accumulation in minutes
>>> data_path = parentdir + '/test_data/odimh5_radar_composites/'
>>> data_recipe = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         data_path=data_path,
...                                         data_recipe=data_recipe,
...                                         latlon=True)
>>>
>>> #show data
>>> fig_name ='_static/one_hour_accum_orig_grid.svg'
>>> title = '1h accumulation original grid'
>>> units = '[mm]'
>>> data       = dat_dict['accumulation']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

The *get_accumulation* method is very similar to *get_instantaneous*. All the features presented above also work with this method.

For this last example, we apply a median filter on the original data, we get the total amount of water during a period of one hour and interpolate the result to a different grid using the *smooth_radius* keyword.

```
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         data_path=data_path,
...                                         data_recipe=data_recipe,
...                                         dest_lon=gem_lon,
...                                         dest_lat=gem_lat,
...                                         median_filt=3,
...                                         smooth_radius=12.,
...                                         latlon=True)
>>>
>>> #if you were to look a "INFO" level logs, you would see what is going on␣
↪under the hood:
>>>
>>> #get_accumulation starting
>>> #get_instantaneous, getting data for:  2019-10-31 16:30:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311630.h5
>>> #get_instantaneous, applying median filter
>>> #get_instantaneous, getting data for:  2019-10-31 16:20:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311620.h5
>>> #get_instantaneous, applying median filter
>>> #get_instantaneous, getting data for:  2019-10-31 16:10:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311610.h5
>>> #get_instantaneous, applying median filter
>>> #get_instantaneous, getting data for:  2019-10-31 16:00:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311600.h5
>>> #get_instantaneous, applying median filter
>>> #get_instantaneous, getting data for:  2019-10-31 15:50:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311550.h5
>>> #get_instantaneous, applying median filter
>>> #get_instantaneous, getting data for:  2019-10-31 15:40:00
>>> #read_h5_composite: reading: b'DBZH' from: /fs/homeu1/eccc/mrd/ords/rpndat/
↪dja001/python/packages/domutils_package/test_data/odimh5_radar_composites/
↪2019/10/31/qcomp_201910311540.h5
>>> #get_instantaneous, applying median filter
>>> #get_accumulation, computing average precip rate in accumulation period
>>> #get_accumulation, interpolating to destination grid
>>> #get_accumulation computing accumulation from avg precip rate
>>> #get_accumulation done
```

(continues on next page)

(continued from previous page)

```
>>>
>>> #show data
>>> fig_name ='_static/one_hour_accum_interpolated.svg'
>>> title = '1h accum, filtered and interpolated'
>>> units = '[mm]'
>>> data       = dat_dict['accumulation']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

5.8 Reading in stage IV accumulations

The *get_accumulation* method can also be used to read and manipulate "stage IV" Quantitative Precipitation Estimates.

The data can be obtained from:

https://data.eol.ucar.edu/dataset/21.093

- Files of the form: *ST4.YYYYMMDDHH.??.h* are for the North American domain
- Files of the form: *st4_pr.YYYYMMDDHH.??.h* are for the Caribean domain

5.8.1 Read one file and get lat/lon of the data grid

For this example, we get a 6-h accumulation of precipitation in mm from a 6-hour accumulation file. This is basically just reading one file.

```
>>> #6h accumulations of precipitation
>>> end_date = datetime.datetime(2019, 10, 31, 18, 0)
>>> duration = 360.  #duration of accumulation in minutes here 6h
>>> data_path = parentdir + '/test_data/stage4_composites/'
>>> data_recipe = 'ST4.%Y%m%d%H.06h' #note the '06h' for a 6h accumulation file
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         data_path=data_path,
...                                         data_recipe=data_recipe,
...                                         latlon=True)
>>>
>>> #show data
>>> fig_name ='_static/stageIV_six_hour_accum_orig_grid.svg'
>>> title = '6h accumulation original grid'
>>> units = '[mm]'
>>> data       = dat_dict['accumulation']
>>> latitudes  = dat_dict['latitudes']
>>> longitudes = dat_dict['longitudes']
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          acc_color_map, equal_legs=True)
```

5.8.2 Read one file, interpolate to a different grid, convert to average precipitation rate

The *get_accumulation* method becomes more usefull if you want to interpolate the data at the same time that it is read and/or if you want to compute derived quantities.

In this next example, we read the same 6h accumulation file but this time, we will get the average precipitation rate over the 10km grid used in the previous example.

```
>>> #6h average precipitation rate on 10km grid
>>> end_date = datetime.datetime(2019, 10, 31, 18, 0)
>>> duration = 360.  #duration of accumulation in minutes here 6h
>>> data_path = parentdir + '/test_data/stage4_composites/'
>>> data_recipe = 'ST4.%Y%m%d%H.06h' #note the '06h' for a 6h accumulation file
>>> dat_dict = radar_tools.get_accumulation(desired_quantity='avg_precip_rate',
↪ #what quantity want
...                                         end_date=end_date,
...                                         duration=duration,
...                                         data_path=data_path,
...                                         data_recipe=data_recipe,
...                                         dest_lon=gem_lon,       #lat/lon␣
↪of 10km grid
...                                         dest_lat=gem_lat,
...                                         smooth_radius=12.)  #use smoothing␣
↪radius of 12km for the interpolation
>>>
>>> #show data
>>> fig_name ='_static/stageIV_six_hour_pr_10km_grid.svg'
>>> title = '6h average precip rate on 10km grid'
>>> units = '[mm/h]'
>>> data       = dat_dict['avg_precip_rate']
>>> latitudes  = gem_lat
>>> longitudes = gem_lon
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          pr_color_map, equal_legs=True)
```

5.8.3 Construct accumulation from several files

Finally, use the *get_accumulation* method to construct accumulations of arbitrary length from stage IV accumulations.

Here, we construct a 3h accumulation from three consecutive 1h accumulations. As before, the data is interpolated to a 10 km grid.

```
>>> #3h accumulation from three 1h accumulations file
>>> end_date = datetime.datetime(2019, 10, 31, 23, 0)
>>> duration = 180.  #duration of accumulation in minutes here 3h
>>> data_path = parentdir + '/test_data/stage4_composites/'
>>> data_recipe = 'ST4.%Y%m%d%H.01h' #note the '01h' for a 1h accumulation file
```

(continues on next page)

(continued from previous page)

```
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         data_path=data_path,
...                                         data_recipe=data_recipe,
...                                         dest_lon=gem_lon,    #lat/lon of␣
↪10km grid
...                                         dest_lat=gem_lat,
...                                         smooth_radius=5.)  #use smoothing␣
↪radius of 5km for the interpolation
>>>
>>> #show data
>>> fig_name ='_static/stageIV_3h_accum_10km_grid.svg'
>>> title = '3h accumulation on 10km grid'
>>> units = '[mm]'
>>> data       = dat_dict['accumulation']
>>> latitudes  = gem_lat
>>> longitudes = gem_lon
>>> plot_img(fig_name, title, units, data, latitudes, longitudes,
...          acc_color_map, equal_legs=True)
```

This tutorial demonstrates how to process batches of observations and perform nowcasting-based time interpolation.

5.9 Time interpolation using nowcasting

Sometimes, you need reflectivity or precipitation-rates at moments in time that do not match exactly with the time at which observational data is available. Simple linear interpolation between two fields at different times will not do a goot job because of precipitation displacement.

As a solution to this, the module *obs_process* provides a nowcast interpolation functionality. The basic idea is that data at intermediate timesteps, data is estimated as a weighted average of the forward advection of the observations before and backward advection of the observations after.

5.9.1 Interpolate a batch of observations

This section demonstrates the processing of a batch of observations in one call to the **obs_process** function. The results are then displayed in the form of a short animation.

Lets start with the required imports and directory setup:

```
>>> import os
>>> import datetime
>>> import subprocess
>>> import glob
>>> import shutil
>>> import numpy as np
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
>>> import cartopy.crs as ccrs
```

(continues on next page)

(continued from previous page)

```
>>> import cartopy.feature as cfeature
>>> import domutils
>>> import domutils.legs as legs
>>> import domutils.geo_tools as geo_tools
>>> import domutils.radar_tools as radar_tools
>>> import domcmc.fst_tools as fst_tools
>>>
>>> #setting up directories
>>> domutils_dir = os.path.dirname(domutils.__file__)
>>> package_dir  = os.path.dirname(domutils_dir)
>>> test_data_dir = package_dir+'/test_data/'
>>> test_results_dir = package_dir+'/test_results/'
>>> figure_dir = test_results_dir+'/t_interp_demo/'
>>> if not os.path.isdir(test_results_dir):
...     os.makedirs(test_results_dir)
>>> if not os.path.isdir(figure_dir):
...     os.makedirs(figure_dir)
>>> log_dir = './logs'
>>> if not os.path.isdir(log_dir):
...     os.makedirs(log_dir)
>>>
>>> # matplotlib global settings
>>> mpl.rcParams.update({'font.size': 28})
>>> mpl.rcParams['font.family'] = 'Latin Modern Roman'
>>> mpl.rcParams['figure.dpi'] = 200
```

obs_process is a python script callable from the shell such as:

```
#process data with time interpolation
python -m domutils.radar_tools.obs_process    \
          --input_t0         202206150800     \
          --input_tf         202206160000     \
          --input_dt         10               \
          --output_t0        202206150900     \
          --output_tf        202206160000     \
          --output_dt        1                \
          --t_interp_method  'nowcast'        \
          ...
```

However, for this example we will be running directly from Python with arguments provided by the attributes of a simple object.

```
>>> # a class that mimics the output of argparse
>>> class ArgsClass():
...     input_t0                = '202205212050'
...     input_tf                = '202205212140'
...     input_dt                = 10
...     output_t0               = '202205212110'
...     output_tf               = '202205212140'
...     output_dt               = 1
...     output_file_format      = 'fst'
...     complete_dataset        = 'False'
```

(continues on next page)

(continued from previous page)

```
...     t_interp_method          = 'nowcast'
...     input_data_dir           = test_data_dir+'/odimh5_radar_composites/'
...     input_file_struc         = '%Y/%m/%d/qcomp_%Y%m%d%H%M.h5'
...     h5_latlon_file           = test_data_dir+'radar_continental_2.5km_2882x2032.
↪pickle'
...     sample_pr_file           = test_data_dir+'hrdps_5p1_prp0.fst'
...     ncores                   = 1    # use as many cpus as you have on your system
...     median_filt              = 3
...     output_dir               = test_results_dir+'/obs_process_t_interp/'
...     processed_file_struc     = '%Y%m%d%H%M.fst'
...     tinterpolated_file_struc = '%Y%m%d%H.fst'
...     log_level                = 'WARNING'
```

The processing of observations and time interpolation is done in one simple function call.

```
>>> # all the arguments are attributes of the args object
>>> args = ArgsClass()
>>>
>>> # observations are processed here
>>> radar_tools.obs_process(args)
```

To make an animation showing the time-interpolated dat, we first define a function for plotting each individual panels.

```
>>> def plot_panel(data,
...                fig, ax_pos, title,
...                proj_aea, map_extent,
...                proj_obj, colormap,
...                plot_palette=None,
...                pal_units=None,
...                show_artefacts=False):
...
...     ax = fig.add_axes(ax_pos, projection=proj_aea)
...     ax.set_extent(map_extent)
...     dum = ax.annotate(title, size=32,
...                       xy=(.022, .85), xycoords='axes fraction',
...                       bbox=dict(boxstyle="round", fc='white', ec='white'))
...
...     # projection from data space to image space
...     projected_data = proj_obj.project_data(data)
...
...     # plot data & palette
...     colormap.plot_data(ax=ax, data=projected_data,
...                        palette=plot_palette,
...                        pal_units=pal_units, pal_format='{:5.1f}',
...                        equal_legs=True)
...
...     # add political boundaries
...     ax.add_feature(cfeature.STATES.with_scale('10m'), linewidth=0.5, edgecolor='0.2')
...
...     # show artefacts in accumulation plots
...     if show_artefacts:
...         ax2 = fig.add_axes(ax_pos)
```

(continues on next page)

(continued from previous page)

```
...         ax2.set_xlim((0.,1.))
...         ax2.set_ylim((0.,1.))
...         ax2.patch.set_alpha(0.0)
...         ax2.set_axis_off()
...         for x0, y0, dx in [(.17,.75,.1), (.26,.79,.1), (.36,.83,.1)]:
...             ax2.arrow(x0, y0, dx, -.03,
...                       width=0.015, facecolor='red', edgecolor='black',
...                       head_width=3*0.01, linewidth=2.)
...
```

We now setup the general characteristics of the figure being generated. See *Legs Tutorial* for information on the definition of color mapping objects.

```
>>> #pixel density of each panel
>>> ratio = 1.
>>> hpix = 600.       #number of horizontal pixels
>>> vpix = ratio*hpix #number of vertical pixels
>>> img_res = (int(hpix),int(vpix))
>>>
>>> #size of image to plot
>>> fig_w = 19.                    #size of figure
>>> fig_h = 15.7                   #size of figure
>>> rec_w = 7./fig_w               #size of axes
>>> rec_h = ratio*(rec_w*fig_w)/fig_h #size of axes
>>> sp_w = .5/fig_w                #space between panel and border
>>> sp_m = 2.2/fig_w               #space between panels
>>> sp_h = .5/fig_h                #space between panels
>>>
>>> # color mapping object
>>> range_arr = [.1,1.,5.,10.,25.,50.,100.]
>>> missing = -9999.
>>> # colormap object for precip rates
>>> pr_colormap = legs.PalObj(range_arr=range_arr,
...                           n_col=6,
...                           over_high='extend', under_low='white',
...                           excep_val=missing,
...                           excep_col='grey_200')
>>> # colormap for QI index
>>> pastel = [ [[255,190,187],[230,104, 96]],  #pale/dark red
...            [[255,185,255],[147, 78,172]],  #pale/dark purple
...            [[255,227,215],[205,144, 73]],  #pale/dark brown
...            [[210,235,255],[ 58,134,237]],  #pale/dark blue
...            [[223,255,232],[ 61,189, 63]] ] #pale/dark green
>>> qi_colormap = legs.PalObj(range_arr=[0., 1.],
...                           dark_pos='high',
...                           color_arr=pastel,
...                           excep_val=[missing,0.],
...                           excep_col=['grey_220','white'])
>>>
>>> #setup cartopy projection
>>> ##250km around Blainville radar
>>> pole_latitude=90.
```

(continues on next page)

(continued from previous page)

```
>>> pole_longitude=0.
>>> lat_0 = 46.
>>> delta_lat = 2.18/2.
>>> lon_0 = -73.75
>>> delta_lon = 3.12/2.
>>> map_extent=[lon_0-delta_lon, lon_0+delta_lon, lat_0-delta_lat, lat_0+delta_
↪lat]
>>> proj_aea = ccrs.RotatedPole(pole_latitude=pole_latitude, pole_
↪longitude=pole_longitude)
>>>
>>> # get lat/lon of input data from one of the h5 files
>>> dum_h5_file = test_data_dir+'/odimh5_radar_composites/2022/05/21/qcomp_
↪202205212100.h5'
>>> input_ll    = radar_tools.read_h5_composite(dum_h5_file, latlon=True)
>>> input_lats  = input_ll['latitudes']
>>> input_lons  = input_ll['longitudes']
>>>
>>> # get lat/lon of output data
>>> output_ll = fst_tools.get_data(args.sample_pr_file, var_name='PR',␣
↪latlon=True)
>>> output_lats = output_ll['lat']
>>> output_lons = output_ll['lon']
>>>
>>> # instantiate projection object for input data
>>> input_proj_obj = geo_tools.ProjInds(src_lon=input_lons, src_lat=input_lats,
...                                     extent=map_extent, dest_crs=proj_aea,␣
↪image_res=img_res)
>>>
>>> # instantiate projection object for output data
>>> output_proj_obj = geo_tools.ProjInds(src_lon=output_lons, src_lat=output_
↪lats,
...                                      extent=map_extent, dest_crs=proj_aea,␣
↪image_res=img_res)
```

Now, we make individual frames of the animation.

```
>>> this_frame = 1
>>> t0 = datetime.datetime(2022,5,21,21,10)
>>> source_deltat = [0, 10, 20]          # minutes
>>> interpolated_deltat = np.arange(10) # minutes
>>> for src_dt in source_deltat:
...     source_t_offset = datetime.timedelta(seconds=src_dt*60.)
...     source_valid_time = t0 + source_t_offset
...
...     for interpolated_dt in interpolated_deltat:
...         interpolated_t_offset = datetime.timedelta(seconds=interpolated_
↪dt*60.)
...         interpolated_valid_time = (t0 + source_t_offset) + interpolated_t_
↪offset
...
...         # instantiate figure
...         fig = plt.figure(figsize=(fig_w,fig_h))
```

(continues on next page)

(continued from previous page)

```
...
...         # source data on original grid
...         dat_dict = radar_tools.get_instantaneous(desired_quantity='precip_
↪rate',
...                                                  valid_date=source_valid_
↪time,
...                                                  data_path=args.input_data_
↪dir,
...                                                  data_recipe=args.input_
↪file_struc)
...         x0 = sp_w + rec_w + sp_m
...         y0 = 2.*sp_h + rec_h
...         ax_pos = [x0, y0, rec_w, rec_h]
...         title = f'Source data \n @ t0+{src_dt}minutes'
...         plot_panel(dat_dict['precip_rate'],
...                    fig, ax_pos, title,
...                    proj_aea, map_extent,
...                    input_proj_obj, pr_colormap,
...                    plot_palette='right',
...                    pal_units='mm/h')
...
...         # processed data on destination grid
...         dat_dict = radar_tools.get_instantaneous(desired_quantity='precip_
↪rate',
...                                                  valid_date=source_valid_
↪time,
...                                                  data_path=args.output_dir+
↪'processed/',
...                                                  data_recipe=args.
↪processed_file_struc)
...         x0 = sp_w + rec_w + sp_m
...         y0 = sp_h
...         ax_pos = [x0, y0, rec_w, rec_h]
...         title = f'Processed data \n @ t0+{src_dt}minutes'
...         plot_panel(dat_dict['precip_rate'],
...                    fig, ax_pos, title,
...                    proj_aea, map_extent,
...                    output_proj_obj, pr_colormap,
...                    plot_palette='right',
...                    pal_units='mm/h')
...
...         # Time interpolated data
...         dat_dict = radar_tools.get_instantaneous(desired_quantity='precip_
↪rate',
...                                                  valid_date=interpolated_
↪valid_time,
...                                                  data_path=args.output_dir,
...                                                  data_recipe=args.
↪tinterpolated_file_struc)
...         x0 = sp_w
...         y0 = sp_h
...         ax_pos = [x0, y0, rec_w, rec_h]
```

(continues on next page)

(continued from previous page)

```
...         title = f'Interpolated \n @ t0+{src_dt+interpolated_dt}minutes'
...         plot_panel(dat_dict['precip_rate'],
...                    fig, ax_pos, title,
...                    proj_aea, map_extent,
...                    output_proj_obj, pr_colormap)
...
...         # quality index is also interpolated using nowcasting
...         x0 = sp_w
...         y0 = 2.*sp_h + rec_h
...         ax_pos = [x0, y0, rec_w, rec_h]
...         title = f'Quality Ind Interpolated \n @ t0+{src_dt+interpolated_dt}
↪minutes'
...         plot_panel(dat_dict['total_quality_index'],
...                    fig, ax_pos, title,
...                    proj_aea, map_extent,
...                    output_proj_obj, qi_colormap,
...                    plot_palette='right',
...                    pal_units='[unitless]')
...
...         # save output
...         fig_name = figure_dir+f'{this_frame:02}_time_interpol_demo_plain.
↪png'
...         plt.savefig(fig_name,dpi=400)
...         plt.close(fig)
...
...         # use "convert" to make a gif out of the png
...         cmd = ['convert', '-geometry', '15%', fig_name, fig_name.replace(
↪'png', 'gif')]
...         process = subprocess.Popen(cmd, stdout=subprocess.PIPE)
...         output, error = process.communicate()
...
...         # we don't need the original png anymore
...         os.remove(fig_name)
...
...         this_frame += 1
```

Finally, an animated gif is constructed from the frames we just made,

```
>>> movie_name = '_static/time_interpol_plain_movie.gif'
>>> if os.path.isfile(movie_name):
...     os.remove(movie_name)
>>> gif_list = sorted(glob.glob(figure_dir+'*.gif'))   #*
>>> cmd = ['convert', '-loop', '0', '-delay', '30']+gif_list+[movie_name]
>>> process = subprocess.Popen(cmd, stdout=subprocess.PIPE)
>>> output, error = process.communicate()
```

5.9.2 Accumulations from time interpolated data

Using nowcasting for time interpolation can be advantageous when computing accumulations from source data available at discrete times. In the example below, we compare accumulations obtained from the source data to accumulations obtained from the time interpolated data.

```
>>> end_date = datetime.datetime(2022,5,21,21,40)
>>> duration = 30 # minutes
>>>
>>> # instantiate figure
>>> fig = plt.figure(figsize=(fig_w,fig_h))
>>>
>>> # make accumulation from source data
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         input_dt=10., # minutes
...                                         data_path=args.input_data_dir,
...                                         data_recipe=args.input_file_struc)
>>> x0 = 2.*sp_w + rec_w
>>> y0 = 2.*sp_h + rec_h
>>> ax_pos = [x0, y0, rec_w, rec_h]
>>> title = 'Accumulation from \n source data'
>>> plot_panel(dat_dict['accumulation'],
...            fig, ax_pos, title,
...            proj_aea, map_extent,
...            input_proj_obj, pr_colormap,
...            plot_palette='right',
...            pal_units='mm',
...            show_artefacts=True)
>>>
>>> # make accumulation from processed data
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
...                                         input_dt=10., # minutes
...                                         data_path=args.output_dir+'/
↪processed/',
...                                         data_recipe=args.processed_file_
↪struc)
>>> x0 = 2.*sp_w + rec_w
>>> y0 = sp_h
>>> ax_pos = [x0, y0, rec_w, rec_h]
>>> title = 'Accumulation from \n processed data'
>>> plot_panel(dat_dict['accumulation'],
...            fig, ax_pos, title,
...            proj_aea, map_extent,
...            output_proj_obj, pr_colormap,
...            plot_palette='right',
...            pal_units='mm',
...            show_artefacts=True)
>>>
>>> # make accumulation from time interpolated data
>>> dat_dict = radar_tools.get_accumulation(end_date=end_date,
...                                         duration=duration,
```

(continues on next page)

(continued from previous page)

```
...                                             input_dt=1., # minutes
...                                             data_path=args.output_dir,
...                                             data_recipe=args.tinterpolated_
↪file_struc)
>>> x0 = sp_w
>>> y0 = sp_h
>>> ax_pos = [x0, y0, rec_w, rec_h]
>>> title = 'Accumulation from \n time interpolated data'
>>> plot_panel(dat_dict['accumulation'],
...            fig, ax_pos, title,
...            proj_aea, map_extent,
...            output_proj_obj, pr_colormap)
>>>
>>> # save output
>>> fig_name = '_static/time_interpol_demo_accum_plain.svg'
>>> plt.savefig(fig_name,dpi=400)
>>> plt.close(fig)
>>>
>>> # we are done, remove log s
>>> if os.path.isdir(log_dir):
...     shutil.rmtree(log_dir)
```

The figure below shows 30 minutes precipitation accumulation computed from:

- The source data every 10 minutes
- The filtered data every 10 minutes
- The time interpolated data every 1 minute

In the two panels on the right, the red arrows indicate artefacts that originate from the poor time resolution of the source data compared to the speed at which the bow echo propagates.

The accumulation on the left is constructed from the time-interpolated values every minute and does not display the displacement artefacts.

CHAPTER

SIX

RADAR_TOOLS API

domutils.radar_tools.get_instantaneous.**get_instantaneous**(*valid_date=None*, *data_path=None*, *data_recipe=None*, *desired_quantity=None*, *median_filt=None*, *coef_a=None*, *coef_b=None*, *qced=True*, *missing=-9999.0*, *latlon=False*, *dest_lon=None*, *dest_lat=None*, *average=False*, *nearest_time=None*, *smooth_radius=None*, *odim_latlon_file=None*, *verbose=0*)

Get instantaneous precipitation from various sources

Provides one interface for:

- reading the following input formats:
  - Odim h5 composites
  - Stage IV files
  - MRMS data
  - CMC "standard files"
- output to an arbitrary output grid
- Consistent filtering (median and/or circular boxcar) on precip observations and the accompanying quality index
- Various types of averaging
- Find the nearest time where observations are available

The file extension present in *data_recipe* determines the file type of the source data Files having no extension are assumed to be in the 'standard' file format

Parameters

- **valid_date** (Optional[Any]) – datetime object with the validity date of the precip field
- **data_path** (Optional[str]) – path to directory where data is expected
- **data_recipe** (Optional[str]) – datetime code for constructing the file name eg: /%Y/%m/%d/qcomp_%Y%m%d%H%M.h5' the filename will be obtained with data_path + valid_date.strftime(data_recipe)
- **desired_quantity** (Optional[str]) – What quantity is desired in output dictionary *precip_rate* in [mm/h] and *reflectivity* in [dBZ] are supported

- **median_filt** (Optional[int]) – If specified, a median filter will be applied on the data being read and the associated quality index. eg. *medialFilter=3* will apply a median filter over a 3x3 boxcar If unspecified, no filtering is applied
- **coef_a** (Optional[float]) – Coefficient *a* in Z = aR^b
- **coef_b** (Optional[float]) – Coefficient *b* in Z = aR^b
- **qced** (Optional[bool]) – Only for Odim H5 composites When True (default), Quality Controlled reflectivity (DBZH) will be returned. When False, raw reflectivity (TH) will be returned.
- **missing** (Optional[float]) – Value that will be assigned to missing data
- **latlon** (Optional[bool]) – Return *latitudes* and *longitudes* grid of the data
- **dest_lon** (Optional[Any]) – Longitudes of destination grid. If not provided data is returned on its original grid
- **dest_lat** (Optional[Any]) – Latitudes of destination grid. If not provided data is returned on its original grid
- **average** (Optional[bool]) – Use the averaging method to interpolate data (see geo_tools documentation), this can be slow
- **nearest_time** (Optional[float]) – If set, rewind time until a match is found to an integer number of *nearestTime* For example, with nearestTime=10, time will be rewinded to the nearest integer of 10 minutes
- **smooth_radius** (Optional[float]) – Use the smoothing radius method to interpolate data, faster (see geo_tools documentation)
- **odim_latlon_file** (Optional[str]) – file containing the latitude and longitudes of Baltrad mosaics in Odim H5 format
- **verbose** (Optional[int]) – – Deprecated – Set >=1 to print info on execution steps

Returns

None If no file matching the desired time is found

or

{

'reflectivity' 2D reflectivity on destination grid (if requested)

'precip_rate' 2D precipitation rate on destination grid (if requested)

'total_quality_index' Quality index of data with 1 = best and 0 = worse

'valid_date' Actual validity date of data

'latitudes' If latlon=True

'longitudes' If latlon=True

}

domutils.radar_tools.get_accumulation.**get_accumulation**(*end_date=None*, *duration=None*, *input_dt=None*, *desired_quantity=None*, *data_path=None*, *data_recipe=None*, *median_filt=None*, *coef_a=None*, *coef_b=None*, *qced=True*, *missing=-9999.0*, *latlon=False*, *dest_lon=None*, *dest_lat=None*, *average=False*, *nearest=None*, *smooth_radius=None*, *odim_latlon_file=None*, *verbose=0*)

Get accumulated precipitation from instantaneous observations

This is essentially a wrapper around get_instantaneous. Data is read during the accumulation period and accumulated (in linear units of precipitation rates) taking the quality index into account. If interpolation to a different grid is desired, it is performed after the accumulation procedure.

If the desired quantity *reflectivity* or *precip_rate* is desired, then the returned quantity will reflect the average precipitation rate during the accumulation period.

With an endTime set to 16:00h and duration to 60 (minutes), data from:

- 15:10h, 15:20h, 15:30h, 15:40h, 15:50h and 16:00h

will be accumulated

Parameters

- **end_date** (Optional[Any]) – datetime object representing the time (inclusively) at the end of the accumulation period
- **duration** (Optional[Any]) – amount of time (minutes) during which precipitation should be accumulated
- **input_dt** (Optional[int]) – time separation (minutes) of input data used for constructing accumulations
- **data_path** (Optional[str]) – path to directory where data is expected
- **data_recipe** (Optional[str]) – datetime code for constructing the file name eg: /%Y/%m/%d/qcomp_%Y%m%d%H%M.h5' the filename will be obtained with data_path + valid_date.strftime(data_recipe)
- **desired_quantity** (Optional[str]) – What quantity is desired in output dictionary. Supported values are:
  - *accumulation* Default option, the amount of water (in mm)
  - *avg_precip_rate* For average precipitation rate (in mm/h) during the accumulation period
  - *reflectivity* For the reflectivity (in dBZ) associated with the average precipitation rate
- **median_filt** (Optional[int]) – If specified, a median filter will be applied on the data being read and the associated quality index. eg. *medialFilter=3* will apply a median filter over a 3x3 boxcar If unspecified, no filtering is applied
- **coef_a** (Optional[float]) – Coefficient *a* in Z = aR^b
- **coef_b** (Optional[float]) – Coefficient *b* in Z = aR^b
- **qced** (Optional[bool]) – Only for Odim H5 composites When True (default), Quality Controlled reflectivity (DBZH) will be returned. When False, raw reflectivity (TH) will be returned.
- **missing** (Optional[float]) – Value that will be assigned to missing data

- **latlon** (Optional[bool]) – Return *latitudes* and *longitudes* grid of the data
- **dest_lon** (Optional[Any]) – Longitudes of destination grid. If not provided data is returned on its original grid
- **dest_lat** (Optional[Any]) – Latitudes of destination grid. If not provided data is returned on its original grid
- **average** (Optional[bool]) – Use the averaging method to interpolate data (see geo_tools documentation), this can be slow
- **nearest** (Optional[float]) – If set, rewind time until a match is found to an integer number of *nearest* For example, with nearest=10, time will be rewinded to the nearest integer of 10 minutes
- **smooth_radius** (Optional[float]) – Use the smoothing radius method to interpolate data, faster (see geo_tools documentation)
- **odim_latlon_file** (Optional[str]) – file containing the latitude and longitudes of Baltrad mosaics in Odim H5 format
- **verbose** (Optional[int]) – – Deprecated – Set >=1 to print info on execution steps

Returns

None If no file matching the desired time is found

or

{

'end_date' End time for accumulation period

'duration' Accumulation length (minutes)

'accumulation' 2D accumulation (mm) on destination grid

'reflectivity' 2D reflectivity on destination grid (if requested)

'precip_rate' precipitation rate on destination grid (if requested)

'total_quality_index' Quality index of data with 1 = best and 0 = worse

'latitudes' If latlon=True

'longitudes' If latlon=True

}

domutils.radar_tools.obs_process.**obs_process**(*args=None*)

Batch processing of precipitation observation

This function provides time and space interpolation of precipitiaton data. If desired, processing such as median filtering and boxcar-type filtering can be applied at the same time.

See for an example.

It is intended to be used as a command-line program, that would be called like:

```
python -m domutils.radar_tools.obs_process
          --input_data_dir   /space/hall4/sitestore/eccc/mrd/rpndat/dja001/data/
↪radar_h5_composites/v8/
          --output_dir       /home/dja001/python/obs_process/outdir/
          --figure_dir       /home/dja001/python/obs_process/figdir/
          --fst_file_struc   %Y/%m/%d/%Y%m%d%H%M_mosaic.fst
```

(continues on next page)

(continued from previous page)

```
          --input_file_struc %Y/%m/%d/qcomp_%Y%m%d%H%M.h5
          --h5_latlon_file   /home/dja001/shared_stuff/files/radar_continental_2.
↪5km_2882x2032.pickle
          --t0               ${t_start}
          --tf               ${t_stop}
          --input_dt         10
          --sample_pr_file   /space/hall4/sitestore/eccc/mrd/rpndat/dja001/domains/
↪hrdps_5p1_prp0.fst
          --ncores           40
          --complete_dataset True
          --median_filt      3
          --smooth_radius    4
```

Alternatively, is is possible to call this function directly from Python by defining a simple object whose attributes are the arguments. Such use is demonstrated in *Interpolate a batch of observations*.

Argument description:

read radar H5 files, interpolate/smooth and write to FST

```
usage: obs_process [-h] --input_data_dir INPUT_DATA_DIR --output_dir
                   OUTPUT_DIR --input_t0 INPUT_T0 --processed_file_struc
                   PROCESSED_FILE_STRUC --input_file_struc INPUT_FILE_STRUC
                   --input_dt INPUT_DT
                   [--tinterpolated_file_struc TINTERPOLATED_FILE_STRUC]
                   [--h5_latlon_file H5_LATLON_FILE] [--input_tf INPUT_TF]
                   [--fcst_len FCST_LEN] [--accum_len ACCUM_LEN]
                   [--output_t0 OUTPUT_T0] [--output_tf OUTPUT_TF]
                   [--output_dt OUTPUT_DT] [--t_interp_method T_INTERP_METHOD]
                   [--sample_pr_file SAMPLE_PR_FILE]
                   [--output_file_format OUTPUT_FILE_FORMAT] [--ncores NCORES]
                   [--complete_dataset COMPLETE_DATASET]
                   [--median_filt MEDIAN_FILT] [--smooth_radius SMOOTH_RADIUS]
                   [--figure_dir FIGURE_DIR] [--cartopy_dir CARTOPY_DIR]
                   [--figure_format FIGURE_FORMAT] [--log_level LOG_LEVEL]
```

6.1 Required named arguments

--input_data_dir path of source radar mosaics files

--output_dir directory for output fst files

--input_t0 yyyymmsshhmmss begining time; datestring

--processed_file_struc strftime syntax for constructing fst filenames for output of obsprocess

--input_file_struc strftime syntax for constructing H5 filenames

--input_dt interval (minutes) between input radar mosaics

6.2 Optional named arguments

--tinterpolated_file_struc strftime syntax for constructing time interpolated filenames
Default: "None"

--h5_latlon_file Pickle file containing the lat/lons of the Baltrad grid
Default: "None"

--input_tf yyyymmsshhmmss end time; datestring
Default: "None"

--fcst_len duration of forecast (hours)
Default: None

--accum_len duration of accumulation (minutes)
Default: "None"

--output_t0 yyyymmsshhmmss begining time; datestring
Default: "None"

--output_tf yyyymmsshhmmss end time; datestring
Default: "None"

--output_dt interval (minutes) between output radar mosaics
Default: "None"

--t_interp_method time interpolation method
Default: "None"

--sample_pr_file File containing PR to establish the domain
Default: "None"

--output_file_format File format of processed files
Default: "npz"

--ncores number of cores for parallel execution
Default: 1

--complete_dataset Skip existing files, default is to clobber them
Default: "False"

--median_filt box size (pixels) for median filter
Default: "None"

--smooth_radius radius (km) where radar data be smoothed
Default: "None"

--figure_dir If provided, a figure will be created for each std file created
Default: "no_figures"

--cartopy_dir Directory for cartopy shape files
Default: "None"

--figure_format	File format of figure Default: "gif"
--log_level	minimum level of messages printed to stdout and in log files Default: "INFO"

domutils.radar_tools.exponential_zr.**exponential_zr**(*data*, *coef_a=None*, *coef_b=None*, *dbz_to_r=False*, *r_to_dbz=False*, *missing=-9999.0*, *undetect=-3333.0*)

Conversion between dBZ and precip rate (R) using exponential Z-R

The relation being used is Z = aR^b

When no coefficients are provided, use WSR-88D Z-R coefficients.

- a=300, b=1.4

Use

- a=200, b=1.6

for the original Marshall-Palmer.

To get dBZ from R we have:

$$\begin{aligned} dBZ &= 10\log_{10}(Z) \\ &= 10\log_{10}(a) + 10b\log_{10}(R) \\ &= U + V\log_{10}(R) \end{aligned}$$

To get R from dBZ:

$$R = \frac{10^{\frac{dBZ}{(10b)}}}{10^{\frac{\log_{10}(a)}{b}}} = \frac{10^{\frac{dBZ}{W}}}{X}$$

Parameters

- **data** (Any) – (array-Like) data to be converted
- **coef_a** (Optional[float]) – Coefficient *a* in Z = aR^b
- **coef_b** (Optional[float]) – Coefficient *b* in Z = aR^b
- **dbz_to_r** (Optional[bool]) – When True input data is assumed in dBZ and output is R in mm/h
- **r_to_dbz** (Optional[bool]) – When True, input data is assumed in mm/h and is converted to dBZ One of dBZtoR or RtodBZ must be set to True

Returns

A numpy array of the size of input data containing the converted data.

Example

```
>>> import domutils.radar_tools as radar_tools
>>> import numpy as np
>>> #reflectivity values
>>> ref_dbz  = np.array([-10., -5., 0., 5, 10, 20, 30, 40, 50, 60])
>>> #convert to precip rate im mm/h
>>> rate_mmh = radar_tools.exponential_zr(ref_dbz, dbz_to_r=True)
```

(continues on next page)

(continued from previous page)

```
>>> with np.printoptions(precision=3, suppress=True):
...     print(rate_mmh)
[  0.003   0.007   0.017   0.039   0.088   0.456   2.363  12.24   63.395
 328.354]
>>>
>>> #convert back to dBZ
>>> recovered_dbz= radar_tools.exponential_zr(rate_mmh, r_to_dbz=True)
>>> print(recovered_dbz)
[-10.  -5.   0.   5.  10.  20.  30.  40.  50.  60.]
```

domutils.radar_tools.read_h5_composite.**read_h5_composite**(*odim_file=None*, *latlon=False*, *qced=True*, *no_data=-9999.0*, *undetect=-3333.0*, *latlon_file=None*, *verbose=0*)

Read reflectivity and quality index from OdimH5 composite

Odim H5 files are a bit annoying in that datasets and quality index are just numbered (data1, data2, ...) and not named following what they contain. Also, one cannot expect that a dataset named 'quality3' will always contain the same quantity. The result is that one has to loop over each item and check that the contents match what we are looking for.

This routine does that as well as converting the Byte data found in H5 files to more tangible quantities such as reflectivity in dBZ and quality index between 0. and 1. Special values are also assigned to no_data and undetect values. UTC time zone is assumed for datestamp in H5 file

Parameters

- **odim_file** (Optional[str]) – /path/to/odim/composite.h5
- **latlon** (Optional[bool]) – When true, will output latitudes and longitudes of the odim grid
- **qced** (Optional[bool]) – When true (default), Quality Controlled reflectivity (DBZH) will be returned. When false, raw reflectivity (TH) will be returned.
- **no_data** (Optional[float]) – Value that will be assigned to missing values
- **undetect** (Optional[float]) – Value that will be assigned to valid measurement of no precipitation
- **latlon_file** (Optional[str]) – pickle file containing latlon of domain (only used if latlon=True)

Returns

If no or invalid file present

or a dictionary containing:

'reflectivity': (ndarray) 2D reflectivity

'total_quality_index': (ndarray) 2D quality index

'valid_date': (python datetime object) date of validity

'latitudes': (ndarray) 2d latitudes of data (conditional on latlon == True)

'longitudes': (ndarray) 2d longitudes of data (conditional on latlon == True)

Return type

None

Example

```
>>> #read odim H5 file
>>> import os, inspect
>>> import domutils.radar_tools as radar_tools
>>> currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
>>> parentdir = os.path.dirname(currentdir) #directory where this package lives
>>> out_dict = radar_tools.read_h5_composite(parentdir + '/test_data/odimh5_radar_
↪composites/2019/10/31/qcomp_201910311630.h5')
>>> h5_reflectivity        = out_dict['reflectivity']
>>> h5_total_quality_index = out_dict['total_quality_index']
>>> h5_valid_date          = out_dict['valid_date']
>>> print(h5_reflectivity.shape)
(2882, 2032)
>>> print(h5_valid_date)
2019-10-31 16:30:00+00:00
```

domutils.radar_tools.read_h5_vol.**read_h5_vol**(*odim_file=None*, *latlon=False*, *elevations='all'*, *quantities='all'*, *include_quality=False*, *no_data=-9999.0*, *undetect=-3333.0*)

Read reflectivity and quality index from OdimH5 composite

Odim H5 files are a bit annoying in that datasets and quality index are just numbered (data1, data2, ...) and not named following what they contain. Also, one cannot expect that a dataset named 'quality3' will always contain the same quantity. The result is that one has to loop over each item and check that the contents match what we are looking for. This routine does that as well as converting the Byte data found in H5 files to more tangible quantities such as reflectivity in dBZ and quality index between 0. and 1. Special values are also assigned to no_data and undetect values. UTC time zone is assumed for datestamp in H5 file

This code is for Volume scans in ODIM format. Reading these files is a bit different from reading mosaics so a different reader is necessary

For more verbose outputs, set logging level in calling handler

Parameters

- **odim_file** (Optional[str]) – /path/to/odim/composite.h5
- **latlon** (Optional[bool]) – When true, will output latitudes and longitudes of the returned PPIs
- **elevations** (Optional[Any]) – float or list of float for the desired nominal elevation. Set to 'all' to return all elevations in a file
- **quantities** (*include_quality Quality field will be included along side*) – desired quantities ['dbzh', 'th', 'rhohv', 'uphidp', 'wradh', 'phidp', 'zdr', 'kdp', 'sqih', 'vradh', 'dr', etc] set to 'all' to return everything
- **quantities** –
- **no_data** (Optional[float]) – Value that will be assigned to missing values
- **undetect** (Optional[float]) – Value that will be assigned to valid measurement of no precipitation

Returns

If no or invalid file present or desired elevation or quantities not found.

or a dictionary which mimics the structure of odim h5 files:

```
# dict['elevation1']['quantity1'](2D numpy array, PPI)
#                   ['quantity2'](2D numpy array, PPI)
#                      ...
#                   ['quality_name1'](2D numpy array, PPI)
#                   ['quality_name2'](2D numpy array, PPI)
#                      ...
#                   [nominal_elevation] float
#                   [elevations] (1D numpy array, rows of values)
#                   [azimuths]   (1D numpy array, rows of values)
#                   [ranges]     (1D numpy array, columns of values)
#                   [latitudes]  (2D numpy array)
#                   [longitudes] (2D numpy array)
# dict['elevation2']['quantity1'](2D numpy array)
#                      ...
```

Return type
None

Example

```
>>> #read odim H5 file
>>> import os, inspect
>>> import domutils.radar_tools as radar_tools
>>> currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
>>> parentdir = os.path.dirname(currentdir) #directory where this package lives
>>> odim_file = parentdir+'/test_data//odimh5_radar_volume_scans/2019071120_24_
↪ODIMH5_PVOL6S_VOL.qc.casbv.h5'
>>> res = radar_tools.read_h5_vol(odim_file=odim_file,
...                               elevations=[0.4],
...                               quantities=['all'],
...                               include_quality=True,
...                               latlon=True)
>>> #check returned radar dictionary
>>> print(res.keys())
dict_keys(['radar_height', 'radar_lat', 'radar_lon', 'date_str', '0.4'])
>>> #
>>> #check returned PPI dictionary
>>> print(res['0.4'].keys())
dict_keys(['dbzh', 'vradh', 'th', 'rhohv', 'uphidp', 'wradh', 'phidp', 'zdr', 'dr',
↪'kdp', 'sqih', 'quality_beamblockage', 'quality_att', 'quality_broad', 'quality_
↪qi_total', 'nominal_elevation', 'azimuths', 'elevations', 'ranges', 'latitudes',
↪'longitudes', 'm43_heights'])
```

domutils.radar_tools.read_fst_composite.**read_fst_composite**(*fst_file=None*, *valid_date=None*, *latlon=False*, *no_data=-9999.0*, *undetect=-3333.0*, *verbose=0*)

Read reflectivity or precip_rate from CMC *standard* files

Validity date is obtained via the *datev* attribute of the entry in the standard file being read. Quality index is set to 1 wherever data is not missing

Parameters

- **fst_file** (Optional[str]) – /path/to/fst/composite.std .stnd .fst or no 'extention'
- **valid_date** (Optional[datetime]) – Datetime object for the time where observations are desired
- **latlon** (Optional[bool]) – When true, will output latitudes and longitudes
- **no_data** (Optional[float]) – Value that will be assigned to missing values
- **undetect** (Optional[float]) – Value that will be assigned to valid measurement of no precipitation

Returns

If no or invalid file present

or a dictionary containing:

'reflectivity': (ndarray) 2D reflectivity

'total_quality_index': (ndarray) 2D quality index

'valid_date': (python datetime object) date of validity

'latitudes': (ndarray) 2d latitudes of data (conditional on latlon = True)

'longitudes': (ndarray) 2d longitudes of data (conditional on latlon = True)

Return type
None

Example

```
>>> #read fst file
>>> import os, inspect
>>> import domutils.radar_tools as radar_tools
>>> currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
>>> parentdir = os.path.dirname(currentdir) #directory where this package lives
>>> out_dict = radar_tools.read_fst_composite(parentdir + '/test_data/std_radar_
↪mosaics/2019103116_30ref_4.0km.stnd')
>>> reflectivity        = out_dict['reflectivity']
>>> total_quality_index = out_dict['total_quality_index']
>>> valid_date          = out_dict['valid_date']
>>> print(reflectivity.shape)
(1650, 1500)
>>> print(valid_date)
2019-10-31 16:30:00+00:00
```

domutils.radar_tools.median_filter.**apply_inds**(*data*, *inds*)

Apply inds computed from speckle_inds

This is just a shortcut to a 1 line piece of code:

data.ravel()[indices] = filtered bla

domutils.radar_tools.median_filter.**get_inds**(*data*, *window=3*)

Indices for speckle filtering through median filter

Applying filter is just a matter of "data.ravel()[indices] = filtered" which is performed by the function *apply_inds()* below.

Data points at the edge are repeated for filtering near the domain border

Returning indices allows to apply the same median filtering on the quality index as on the precipitation data itself.

Parameters

- **data** – source data, should be 2D ndarray
- **window** – Size of boxcar window default is 3x3 Should be an odd integer

Returns

indices for applying speckle filter data[indices] = filtered

Example

```
>>> import numpy as np
>>> import domutils.radar_tools as radar_tools
>>> data = np.array([[1,2,3,0],
...                  [8,9,4,0],
...                  [7,6,5,0]])
>>> inds = radar_tools.median_filter.get_inds(data)
>>> print(inds)
[[ 1  2  1  7]
 [ 8 10  2 11]
 [ 8  9 10 11]]
>>> #median filtered data on a 3x3 window
>>> print(data.ravel()[inds])
[[2 3 2 0]
 [7 5 3 0]
 [7 6 5 0]]
```

domutils.radar_tools.median_filter.**remap_data**(*data*, *window=3*, *mode='clip'*, *return_flat_inds=False*)

Puts 2D data into 3D array for any type of window (boxcar) operations

In compiled code, median filtering and other types of boxcar filtering is performed with nested loops which are very inefficient in Python.

This function remaps original data of shape M x N to a 3D array of shape M x N x W with W being the size of the filtering window.

At each point i,j, the values along the 3rd dimension are all the data values participating in the window averaging.

Numpy operations can then be leveraged for fast computation of these filters.

Parameters

- **data** – source data, should be 2D ndarray
- **window** – Size of boxcar window default is 3x3 Should be an odd integer
- **mode** – mode argument to *np.ravel_multi_index* Use 'clip' (default) to repeat values found at the edge use 'wrap' to wrap around

Returns

a M x N x W remapping of the original data

if return_flat_inds == True, we return the following:

remapped_data: a M x N x W remapping of the original data row_inds: a M x N array containg i index of output array col_inds: a M x N array containg j index of output array flat_inds: a M x N x W array containg flat indices toinput data

Return type

remapped_data

Example

```
>>> import numpy as np
>>> import domutils.radar_tools as radar_tools
>>> data = np.array([[1,2,3,0],
...                  [8,9,4,0],
...                  [7,6,5,0]])
>>> remapped_data = radar_tools.median_filter.remap_data(data, window=3, mode='clip
↪')
>>> #
>>> #data points in window for point i=0, j=0
>>> print(remapped_data[0,0,:])
[1. 1. 2. 1. 1. 2. 8. 8. 9.]
>>> #
>>> print(remapped_data[1,0,:])
[1. 1. 2. 8. 8. 9. 7. 7. 6.]
>>> #
>>> #data points in window for point i=1, j=1
>>> print(remapped_data[1,1,:])
[1. 2. 3. 8. 9. 4. 7. 6. 5.]
>>> #
>>> #sum of data points in boxcar
>>> box_sum = np.sum(remapped_data, axis=2)
>>> print(box_sum)
[[33. 33. 23. 10.]
 [49. 45. 29. 12.]
 [65. 57. 35. 14.]]
>>> #
>>> #boxcar average
>>> box_avg = np.mean(remapped_data, axis=2)
>>> print(box_avg)
[[3.66666667 3.66666667 2.55555556 1.11111111]
 [5.44444444 5.         3.22222222 1.33333333]
 [7.22222222 6.33333333 3.88888889 1.55555556]]
```

To get radar moisaics in various formats and construct accumulations on the fly.

CHAPTER

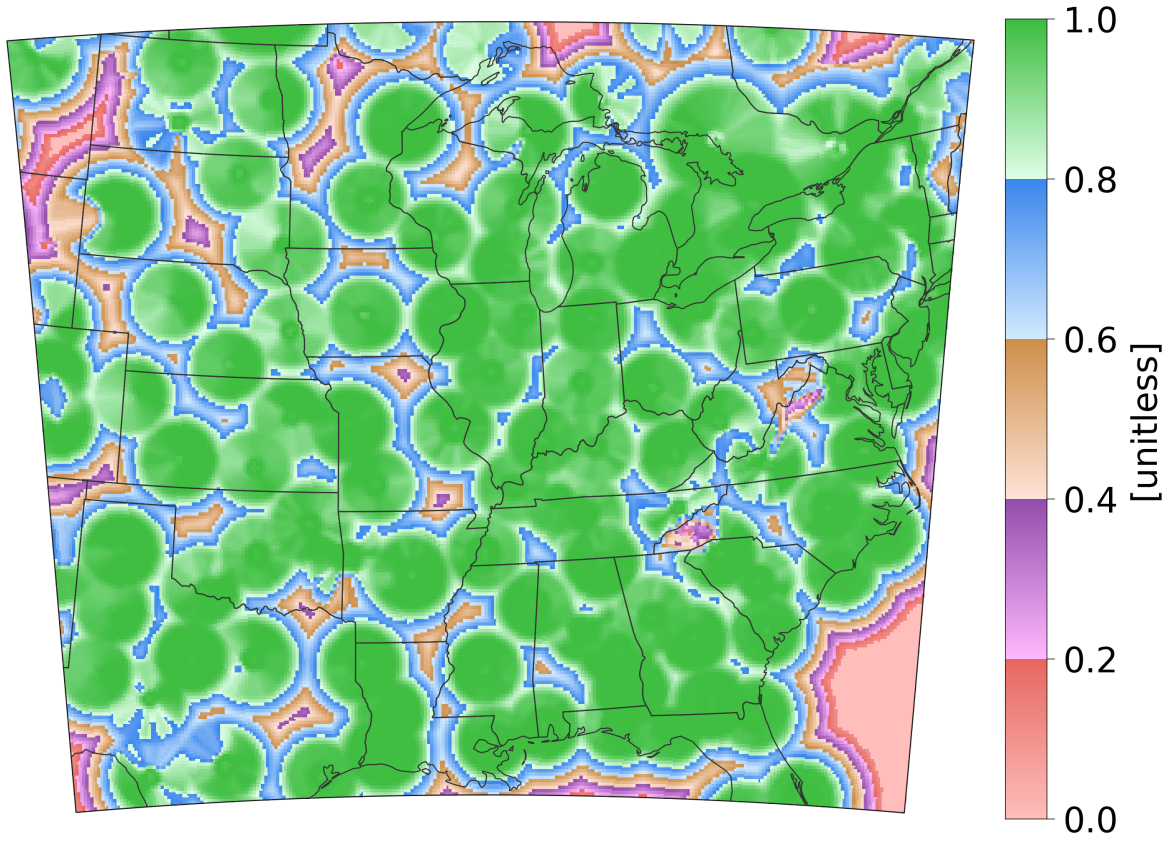
SEVEN

EXAMPLES LEGS + GEO_TOOLS

Real life examples of legs color palettes + geo_tools + cartopy

7.1 Semi-Continuous Semi-Quantitative

Semi-Continuous color mapping for depiction of a 0-1 quality index



```
import os, inspect
import pickle
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# In your scripts use something like :
import domutils.geo_tools as geo_tools
import domutils.legs      as legs

def main():
    #recover previously prepared data
    currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
    parentdir  = os.path.dirname(currentdir) #directory where this package lives
    source_file = parentdir + '/test_data/pal_demo_data.pickle'
    with open(source_file, 'rb') as f:
        data_dict = pickle.load(f)
    longitudes     = data_dict['longitudes']    #2D longitudes [deg]
    latitudes      = data_dict['latitudes']     #2D latitudes  [deg]
    quality_index  = data_dict['qualityIndex']  #2D quality index of a radar mosaic [0-
↪1]; 1 = best quality

    #missing value
    missing = -9999.

    #pixel density of image to plot
    ratio = 0.8
    hpix = 600.       #number of horizontal pixels  E-W
    vpix = ratio*hpix #number of vertical pixels    S-N
    img_res = (int(hpix),int(vpix))

    ##define Albers projection and extend of map
    #Obtained through trial and error for good fit of the mdel grid being plotted
    proj_aea = ccrs.AlbersEqualArea(central_longitude=-94.,
                                    central_latitude=35.,
                                    standard_parallels=(30.,40.))
    map_extent=[-104.8,-75.2,27.8,48.5]

    #point density for figure
    mpl.rcParams['figure.dpi'] = 400
    #larger characters
    mpl.rcParams.update({'font.size': 15})

    #instantiate figure
    fig = plt.figure(figsize=(7.5,6.))

    #instantiate object to handle geographical projection of data
    proj_inds = geo_tools.ProjInds(src_lon=longitudes, src_lat=latitudes,
                                   extent=map_extent,  dest_crs=proj_aea,
                                   image_res=img_res,  missing=missing)
```

(continues on next page)

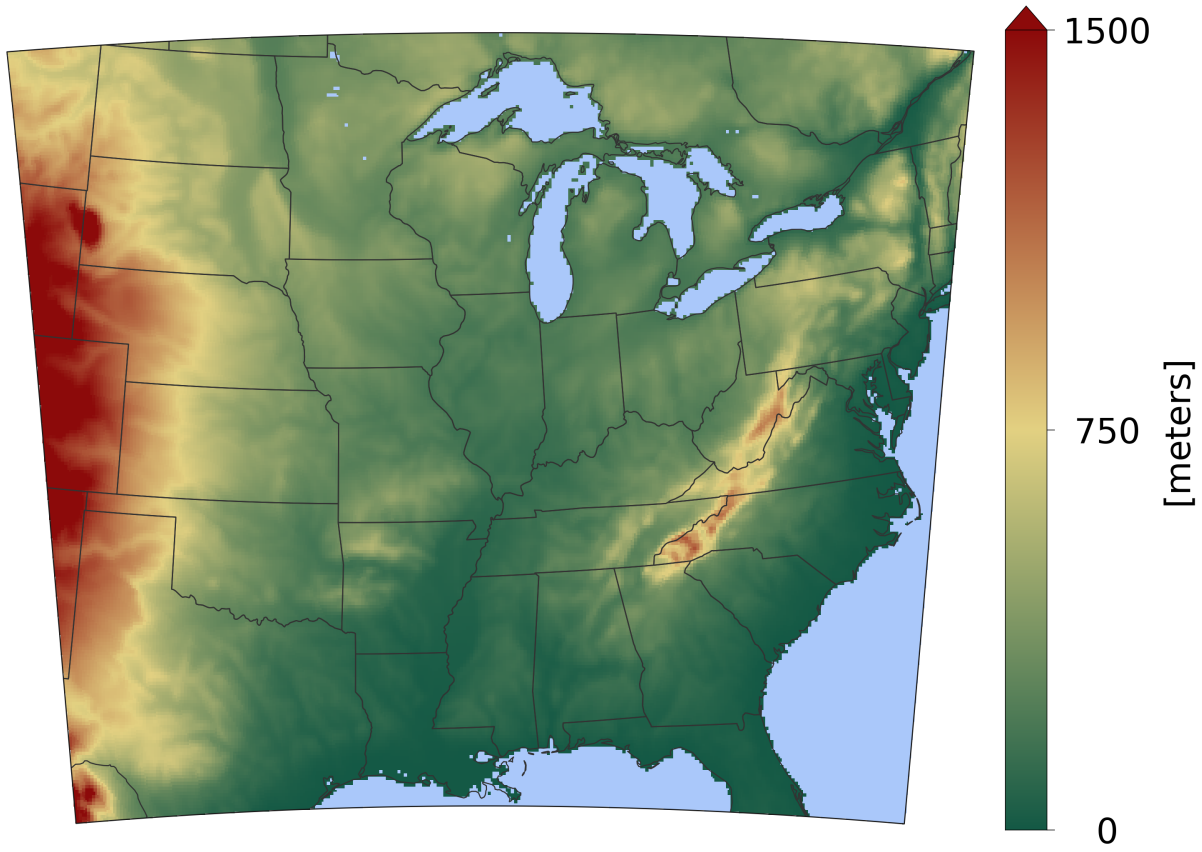
(continued from previous page)

```
    #axes for this plot
    ax = fig.add_axes([.01,.1,.8,.8], projection=proj_aea)
    ax.set_extent(map_extent)

    # Set up colormapping object
    #
    #custom pastel color segments
    pastel = [ [[255,190,187],[230,104, 96]],  #pale/dark red
               [[255,185,255],[147, 78,172]],  #pale/dark purple
               [[255,227,215],[205,144, 73]],  #pale/dark brown
               [[210,235,255],[ 58,134,237]],  #pale/dark blue
               [[223,255,232],[ 61,189, 63]] ] #pale/dark green

    #init color mapping object
    map_qi = legs.PalObj(range_arr=[0., 1.],
                         dark_pos='high',
                         color_arr=pastel,
                         excep_val=[missing],
                         excep_col='grey_120')

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(quality_index)

    #plot data & palette
    map_qi.plot_data(ax=ax,data=proj_data, zorder=0,
                     palette='right', pal_units='[unitless]', pal_format='{:2.1f}')
↪#palette options

    #add political boundaries
    ax.add_feature(cfeature.STATES.with_scale('50m'), linewidth=0.5, edgecolor='0.2',
↪zorder=1)

    #plot border and mask everything outside model domain
    proj_inds.plot_border(ax, mask_outside=True, linewidth=.5)

    #plt.show()
    #plt.savefig('example_custom_semi_continuous.svg')

if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 2.860 seconds)

7.2 Continuous Qualitative

Continuous and qualitative color mapping for depiction of terrain height in a 10 km version of the Canadian GEM atmospheric model.




```
import os, inspect
import pickle
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# In your scripts use something like :
import domutils.legs as legs
import domutils.geo_tools as geo_tools



def main():
    #recover previously prepared data
```

(continues on next page)

(continued from previous page)

```
    currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
    parentdir  = os.path.dirname(currentdir) #directory where this package lives
    source_file = parentdir + '/test_data/pal_demo_data.pickle'
    with open(source_file, 'rb') as f:
        data_dict = pickle.load(f)
    longitudes     = data_dict['longitudes']    #2D longitudes [deg]
    latitudes      = data_dict['latitudes']     #2D latitudes  [deg]
    ground_mask    = data_dict['groundMask']    #2D land fraction [0-1]; 1 = all land
    terrain_height = data_dict['terrainHeight'] #2D terrain height of model [m ASL]

    #flag non-terrain (ocean and lakes) as -3333.
    inds = np.asarray( (ground_mask.ravel() <= .01) ).nonzero()
    if inds[0].size != 0:
        terrain_height.flat[inds] = -3333.

    #missing value
    missing = -9999.

    #pixel density of image to plot
    ratio = 0.8
    hpix = 600.       #number of horizontal pixels E-W
    vpix = ratio*hpix #number of vertical pixels   S-N
    img_res = (int(hpix),int(vpix))

    ##define Albers projection and extend of map
    #Obtained through trial and error for good fit of the mdel grid being plotted
    proj_aea = ccrs.AlbersEqualArea(central_longitude=-94.,
                                    central_latitude=35.,
                                    standard_parallels=(30.,40.))
    map_extent=[-104.8,-75.2,27.8,48.5]

    #point density for figure
    mpl.rcParams['figure.dpi'] = 400
    #larger characters
    mpl.rcParams.update({'font.size': 15})

    #instantiate figure
    fig = plt.figure(figsize=(7.5,6.))

    #instantiate object to handle geographical projection of data
    proj_inds = geo_tools.ProjInds(src_lon=longitudes, src_lat=latitudes,
                                   extent=map_extent,  dest_crs=proj_aea,
                                   image_res=img_res,  missing=missing)

    #axes for this plot
    ax = fig.add_axes([.01,.1,.8,.8], projection=proj_aea)
    ax.set_extent(map_extent)

    # Set up colormapping object
    #
    # Two color segments for this palette
```

(continues on next page)

(continued from previous page)

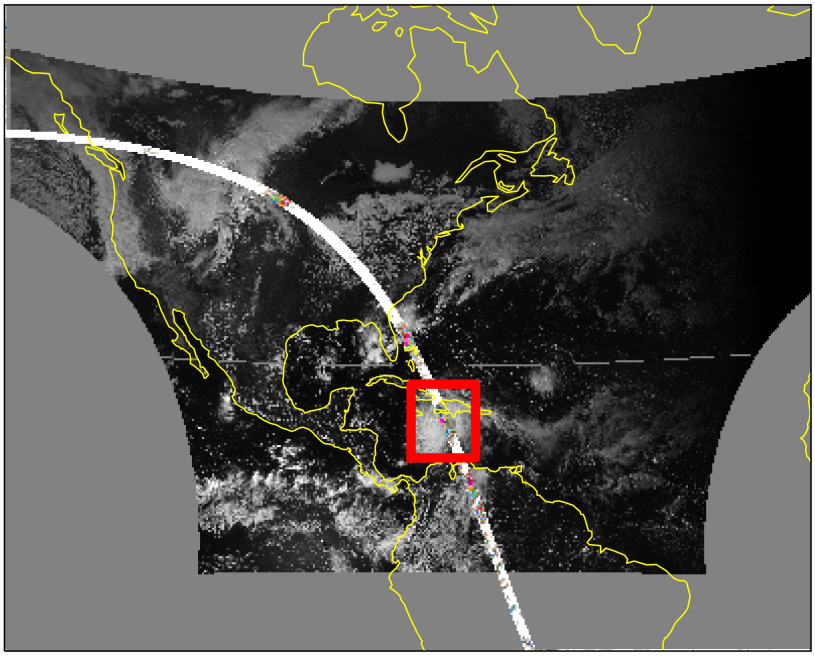
```
    red_green = [[[227,209,130],[ 20, 89, 69]],    # bottom color leg : yellow , dark␣
↪green
                 [[227,209,130],[140, 10, 10]]]    #    top color leg : yellow , dark red

    map_terrain = legs.PalObj(range_arr=[0., 750, 1500.],
                              color_arr=red_green, dark_pos=['low','high'],
                              excep_val=[-3333.       ,missing],
                              excep_col=[[170,200,250],[120,120,120]],  #blue , grey_120
                              over_high='extend')

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(terrain_height)

    #plot data & palette
    map_terrain.plot_data(ax=ax,data=proj_data, zorder=0,
                          palette='right', pal_units='[meters]', pal_format='{:4.0f}')
↪#palette options

    #add political boundaries
    ax.add_feature(cfeature.STATES.with_scale('50m'), linewidth=0.5, edgecolor='0.2',
↪zorder=1)

    #plot border and mask everything outside model domain
    proj_inds.plot_border(ax, mask_outside=True, linewidth=.5)

    #uncomment to save figure
    #plt.savefig('continuous_topo.svg')

if __name__ == '__main__':
    main()
```

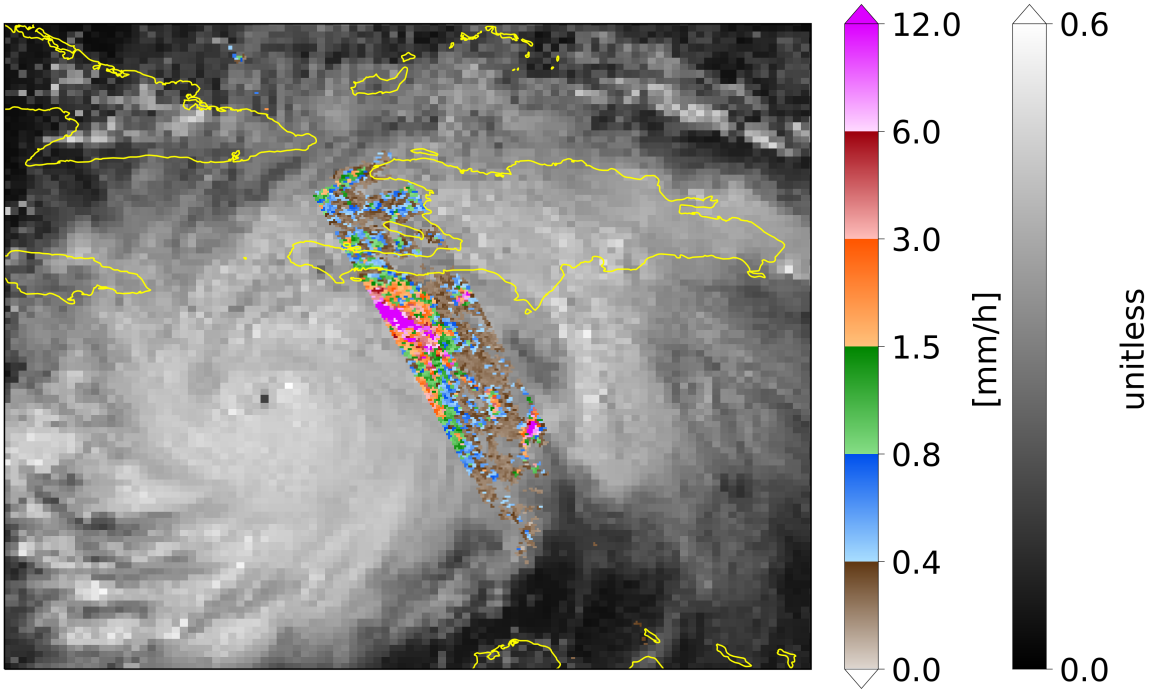
Total running time of the script: (0 minutes 1.665 seconds)

7.3 GPM precip rate measurements over Goes Albedo

Many things are demonstrated in this example:

- The superposition of two data types on one figure
- Plotting the same data on different domains
- The use of the *average* keyword for displaying high resolution data onto a coarser grid






```
import numpy as np
import os, inspect
import pickle
import matplotlib as mpl
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
```

(continues on next page)

(continued from previous page)

```
import cartopy.feature as cfeature

# In your scripts use something like :
import domutils.geo_tools as geo_tools
import domutils.legs      as legs

def make_panel(fig, pos, img_res, map_extent, missing,
               dpr_lats, dpr_lons, dpr_pr,
               goes_lats, goes_lons, goes_albedo,
               map_pr, map_goes,
               map_extent_small=None, include_zero=True,
               average_dpr=False):
    ''' Generic function for plotting data on an ax

        Data is displayed with specific projection settings

    '''

    #cartopy crs for lat/lon (ll) and the image (Miller)
    proj_ll  = ccrs.Geodetic()
    proj_mil = ccrs.Miller()

    #global
    #instantiate object to handle geographical projection of data
    #
    #Note the average=True for GPM data, high resolution DPR data
    # will be averaged within coarser images pixel tiles
    proj_inds_dpr = geo_tools.ProjInds(src_lon=dpr_lons, src_lat=dpr_lats,
                                       extent=map_extent, dest_crs=proj_mil,
                                       average=average_dpr, missing=missing,
                                       image_res=img_res)
    proj_inds_goes = geo_tools.ProjInds(src_lon=goes_lons, src_lat=goes_lats,
                                        extent=map_extent, dest_crs=proj_mil,
                                        image_res=img_res, missing=missing)

    ax = fig.add_axes(pos, projection=proj_mil)
    ax.set_extent(map_extent)

    #geographical projection of data into axes space
    proj_data_pr   = proj_inds_dpr.project_data(dpr_pr)
    proj_data_goes = proj_inds_goes.project_data(goes_albedo)

    #get RGB values for each data types
    precip_rgb =   map_pr.to_rgb(proj_data_pr)
    albedo_rgb = map_goes.to_rgb(proj_data_goes)

    #blend the two images by hand
    #image will be opaque where reflectivity > 0
    if include_zero:
        alpha = np.where(proj_data_pr >= 0., 1., 0.)  #include zero
    else:
        alpha = np.where(proj_data_pr >  0., 1., 0.)  #exclude zero
```

(continues on next page)

(continued from previous page)

```
    combined_rgb = np.zeros(albedo_rgb.shape,dtype='uint8')
    for zz in np.arange(3):
        combined_rgb[:,:,zz] = (1. - alpha)*albedo_rgb[:,:,zz] + alpha*precip_rgb[:,:,zz]

    #plot image w/ imshow
    x11, x22 = ax.get_xlim()    #get image limits in Cartopy data coordinates
    y11, y22 = ax.get_ylim()
    dum = ax.imshow(combined_rgb, interpolation='nearest',
                    origin='upper', extent=[x11,x22,y11,y22])
    ax.set_aspect('auto')

    #add political boundaries
    ax.add_feature(cfeature.COASTLINE, linewidth=0.8, edgecolor='yellow',zorder=1)

    #plot extent of the small domain that will be displayed in next panel
    if map_extent_small is not None :
        bright_red = np.array([255.,0.,0.])/255.
        w = map_extent_small[0]
        e = map_extent_small[1]
        s = map_extent_small[2]
        n = map_extent_small[3]
        ax.plot([w, e, e, w, w], [s,s,n,n,s],transform=proj_ll, color=bright_red,␣
↪linewidth=5 )


def main():


    #recover previously prepared data
    currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
    parentdir  = os.path.dirname(currentdir) #directory where this package lives
    source_file = parentdir + '/test_data/goes_gpm_data.pickle'
    with open(source_file, 'rb') as f:
        data_dict = pickle.load(f)
    dpr_lats    = data_dict['dprLats']
    dpr_lons    = data_dict['dprLons']
    dpr_pr      = data_dict['dprPrecipRate']
    goes_lats   = data_dict['goesLats']
    goes_lons   = data_dict['goesLons']
    goes_albedo = data_dict['goesAlbedo']

    #missing value
    missing = -9999.

    #Figure position stuff
    pic_h = 5.4
    pic_w = 8.8
    pal_sp = .1/pic_w
    pal_w = .25/pic_w
    ratio = .8
    sq_sz = 6.
```

(continues on next page)

(continued from previous page)

```
    rec_w =       sq_sz/pic_w
    rec_h = ratio*sq_sz/pic_h
    sp_w = .1/pic_w
    sp_h = 1.0/pic_h
    x1 = .1/pic_w
    y1 = .2/pic_h

    #number of pixels of the image that will be shown
    hpix = 400.       #number of horizontal pixels E-W
    vpix = ratio*hpix #number of vertical pixels   S-N
    img_res   = (int(hpix),int(vpix))

    #point density for figure
    mpl.rcParams.update({'font.size': 17})
    #Use this to make text editable in svg files
    mpl.rcParams['text.usetex'] = False
    mpl.rcParams['svg.fonttype'] = 'none'
    #Hi def figure
    mpl.rcParams['figure.dpi'] = 400

    #instantiate figure
    fig = plt.figure(figsize=(pic_w,pic_h))

    # Set up colormapping objects

    #For precip rates
    ranges = [0.,.4,.8,1.5,3.,6.,12.]
    map_pr = legs.PalObj(range_arr=ranges,
                         n_col=6,
                         over_high='extend',
                         under_low='white',
                         excep_val=[missing,0.], excep_col=['grey_230','white'])

    #For Goes albedo
    map_goes = legs.PalObj(range_arr=[0., .6],
                           over_high = 'extend',
                           color_arr='b_w', dark_pos='low',
                           excep_val=[-1, missing], excep_col=['grey_130','grey_130'])

    #Plot data on a domain covering North-America
    #
    #
    map_extent = [-141.0, -16., -7.0, 44.0]
    #position
    x0 = x1
    y0 = y1
    pos = [x0,y0,rec_w,rec_h]
    #border of smaller domain to plot on large figure
    map_extent_small = [-78., -68., 12.,22.]
    #
    make_panel(fig, pos, img_res, map_extent, missing,
               dpr_lats, dpr_lons, dpr_pr,
```

(continues on next page)

(continued from previous page)

```
                goes_lats, goes_lons, goes_albedo,
                map_pr, map_goes, map_extent_small,
                average_dpr=True)



    #instantiate 2nd figure
    fig2 = plt.figure(figsize=(pic_w,pic_h))
    # sphinx_gallery_thumbnail_number = 2

    #Closeup on a domain in the viscinity of Haiti
    #
    #
    map_extent = map_extent_small
    #position
    x0 = x1
    y0 = y1
    pos = [x0,y0,rec_w,rec_h]
    #
    make_panel(fig2, pos, img_res, map_extent, missing,
                dpr_lats, dpr_lons, dpr_pr,
                goes_lats, goes_lons, goes_albedo,
                map_pr, map_goes, include_zero=False)


    #plot palettes
    x0 = x0 + rec_w + pal_w
    y0 = y0
    map_pr.plot_palette(pal_pos=[x0,y0,pal_w,rec_h],
                        pal_units='[mm/h]',
                        pal_format='{:2.1f}',
                        equal_legs=True)

    x0 = x0 + 5.*pal_w
    y0 = y0
    map_goes.plot_palette(pal_pos=[x0,y0,pal_w,rec_h],
                          pal_units='unitless',
                          pal_format='{:2.1f}')

    #uncomment to save figure
    #pic_name = 'goes_plus_gpm.svg'
    #plt.savefig(pic_name,dpi=400)


if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 52.420 seconds)

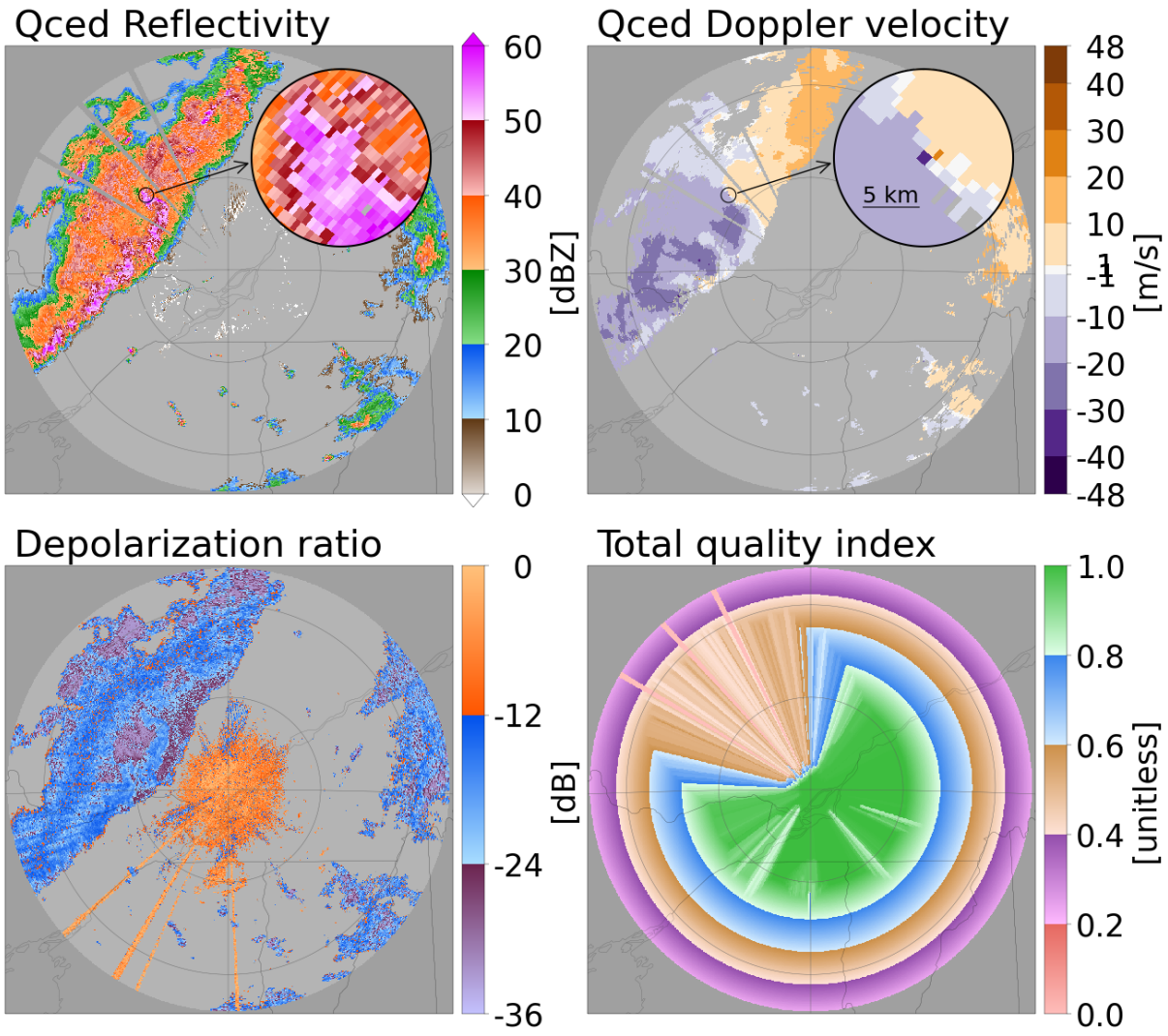
7.4 Radar PPI

Display radar Plan Position Indicators (PPIs) using a combination of *radar_tools*, *geo_tools* and *legs*.

- Data consists of a radar Volume scan in the ODIM HDF5 format that are read with *domutils.radar_tools*.
- Data is projected on Cartopy geoaxes using *domutils.geo_tools*
- Various color mappings are applied with *domutils.legs*

Some advanced concepts are demonstrated in this example.

- Quality controls are applied to Doppler velocity based on the Depolarization Ratio (DR) and the total quality index.
- Matplotlib/cartopy geoaxes are clipped and superposed to show a closeup of a possible meso-cyclone.



```
def add_feature(ax):
    """plot geographical and political boundaries in matplotlib axes
    """
    import cartopy.feature as cfeature
    ax.add_feature(cfeature.STATES.with_scale('10m'), linewidth=0.1, edgecolor='0.1',
↪zorder=1)

def radar_ax_circ(ax, radar_lat, radar_lon):
    '''plot azimuth lines and range circles around a given radar
    '''
    import numpy as np
    import cartopy.crs as ccrs
    import domutils.geo_tools as geo_tools

    #cartopy transform for latlons
    proj_pc = ccrs.PlateCarree()

    color=(100./256.,100./256.,100./256.)

    #add a few azimuths lines
    az_lines = np.arange(0,360.,90.)
    ranges   = np.arange(250.)
    for this_azimuth in az_lines:
        lons, lats = geo_tools.lat_lon_range_az(radar_lon, radar_lat, ranges, this_
↪azimuth)
        ax.plot(lons, lats, transform=proj_pc, c=color, zorder=300, linewidth=.3)

    #add a few range circles
    ranges   = np.arange(0,250.,100.)
    azimuths = np.arange(0,361.)#360 degree will be included for full circles
    for this_range in ranges:
        lons, lats = geo_tools.lat_lon_range_az(radar_lon, radar_lat, this_range,␣
↪azimuths)
        ax.plot(lons, lats, transform=proj_pc, c=color, zorder=300, linewidth=.3)

def main():

    import os, inspect
    import copy
    import numpy as np
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    import cartopy.crs as ccrs

    # imports from domutils
    import domutils.legs as legs
    import domutils.geo_tools   as geo_tools
    import domutils.radar_tools as radar_tools

    #missing values ane the associated color
    missing  = -9999.
    missing_color = 'grey_160'
    undetect = -3333.
```

(continues on next page)

(continued from previous page)

```
    undetect_color = 'grey_180'

    #file to read
    currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
    parentdir  = os.path.dirname(currentdir) #directory where this package lives
    odim_file = parentdir + '/test_data/odimh5_radar_volume_scans/20190711120_24_ODIMH5_
↪PVOL6S_VOL.qc.casbv.h5'

    #Read a PPI from a volume scan in the ODIM H5 format
    #we want the PPI with a nominal elevation of 0.4 degree
    #we retrieve reflectivity (dbzh) and Doppler velocity (vradh) and the Depolarization␣
↪ratio (dr)
    #the reader computes the lat/lon of data points in the PPI for you
    out_dict = radar_tools.read_h5_vol(odim_file=odim_file,
                                       elevations=[0.4],
                                       quantities=['dbzh','vradh','dr'],
                                       no_data=missing, undetect=undetect,
                                       include_quality=True,
                                       latlon=True)
    #radar properties
    radar_lat    = out_dict['radar_lat']
    radar_lon    = out_dict['radar_lon']
    #PPIs
    # You can see the quantities available in the dict with
    # print(out_dict['0.4'].keys())
    reflectivity = out_dict['0.4']['dbzh']
    doppvelocity = out_dict['0.4']['vradh']
    dr           = out_dict['0.4']['dr']
    tot_qi       = out_dict['0.4']['quality_qi_total']
    latitudes    = out_dict['0.4']['latitudes']
    longitudes   = out_dict['0.4']['longitudes']


    #
    #Quality controls on Doppler velocity
    #pixel is considered weather is DR < 12 dB
    weather_target = np.where( dr < -12., 1, 0)
    #not a weather target when DR is missing
    weather_target = np.where( np.isclose(dr, missing),  0, weather_target)
    #not a weather target when DR is undetect
    weather_target = np.where( np.isclose(dr, undetect), 0, weather_target)
    #a 3D representation of a boxcar filter see radar_tools API for docs on this function
    #5x5 search area in polar coordinates
    remapped_data_5 = radar_tools.median_filter.remap_data(weather_target, window=5,␣
↪mode='wrap')
    #if less than 12 points (half the points in a 5x5 window) have good dr, pixel is␣
↪marked as clutter
    is_clutter = np.where(np.sum(remapped_data_5, axis=2) <= 12, True, False)
    #3x3 search area in polar coordinates
    remapped_data_3 = radar_tools.median_filter.remap_data(weather_target, window=3,␣
↪mode='wrap')
```

(continues on next page)

(continued from previous page)

```
    #if less than 9 points (all the points in a 3x3 window) have good dr, pixel is␣
↪marked as clutter
    is_clutter = np.where(np.sum(remapped_data_3, axis=2) < 9, True, is_clutter)
    #
    doppvelocity_qc = copy.deepcopy(doppvelocity)
    #DR filtering only applied to points with reflectivities < 35 dB
    doppvelocity_qc = np.where( np.logical_and(is_clutter , reflectivity < 35.) ,␣
↪undetect, doppvelocity_qc)
    #add filtering using total quality index
    doppvelocity_qc = np.where( tot_qi <  0.2 ,                                  ␣
↪undetect, doppvelocity_qc)


    #pixel density of panels in figure
    ratio = 1.
    hpix = 800.       #number of horizontal pixels E-W
    vpix = ratio*hpix #number of vertical pixels   S-N
    img_res = (int(hpix),int(vpix))

    #cartopy Rotated Pole projection
    pole_latitude=90.
    pole_longitude=0.
    proj_rp = ccrs.RotatedPole(pole_latitude=pole_latitude, pole_longitude=pole_
↪longitude)
    #plate carree
    proj_pc = ccrs.PlateCarree()

    #a domain ~250km around the Montreal area where the radar is
    lat_0 = 45.7063
    delta_lat = 2.18
    lon_0 = -73.85
    delta_lon = 3.12
    map_extent=[lon_0-delta_lon, lon_0+delta_lon, lat_0-delta_lat, lat_0+delta_lat]

    #a smaller domain for a closeup that will be inlaid in figure
    lat_0 = 46.4329666
    delta_lat = 0.072666
    lon_0 = -75.00555
    delta_lon = 0.104
    map_extent_close=[lon_0-delta_lon, lon_0+delta_lon, lat_0-delta_lat, lat_0+delta_lat]
    #a circular clipping mask for the closeup axes
    x = np.linspace(-1., 1, int(hpix))
    y = np.linspace(-1., 1, int(vpix))
    xx, yy = np.meshgrid(x, y, indexing='ij')
    clip_alpha = np.where( np.sqrt(xx**2.+yy**2.) < 1., 1., 0. )
    #a circular line around the center of the closeup window
    radius=8. #km
    azimuths = np.arange(0,361.)#360 degree will be included for full circles
    closeup_lons, closeup_lats = geo_tools.lat_lon_range_az(lon_0, lat_0, radius,␣
↪azimuths)
    #a line 5km long for showing scale in closeup insert
    azimuth = 90 #meteorological angle
```

(continues on next page)

(continued from previous page)

```
    distance = np.linspace(0,5.,50)#360 degree will be included for full circles
    scale_lons, scale_lats = geo_tools.lat_lon_range_az(lon_0-0.07, lat_0-0.04, distance,
↪ azimuth)


    #point density for figure
    mpl.rcParams['figure.dpi'] = 100.   #crank this up for high def images
    # Use this for editable text in svg (eg with Inkscape)
    mpl.rcParams['text.usetex']  = False
    mpl.rcParams['svg.fonttype'] = 'none'
    #larger characters
    mpl.rcParams.update({'font.size': 25})


    # dimensions for figure panels and spaces
    # all sizes are inches for consistency with matplotlib
    fig_w = 13.5           # Width of figure
    fig_h = 12.5           # Height of figure
    rec_w = 5.             # Horizontal size of a panel
    rec_h = ratio * rec_w  # Vertical size of a panel
    sp_w = .5              # horizontal space between panels
    sp_h = .8              # vertical space between panels
    fig = plt.figure(figsize=(fig_w,fig_h))
    xp = .02               #coords of title (axes normalized coordinates)
    yp = 1.02
    #coords for the closeup  that is overlayed
    x0 = .55               #x-coord of bottom left position of closeup (axes coords)
    y0 = .55               #y-coord of bottom left position of closeup (axes coords)
    dx = .4                #x-size of closeup axes (fraction of a "regular" panel)
    dy = .4                #y-size of closeup axes (fraction of a "regular" panel)
    #normalize sizes to obtain figure coordinates (0-1 both horizontally and vertically)
    rec_w = rec_w / fig_w
    rec_h = rec_h / fig_h
    sp_w  = sp_w  / fig_w
    sp_h  = sp_h  / fig_h

    #instantiate objects to handle geographical projection of data
    proj_inds = geo_tools.ProjInds(src_lon=longitudes, src_lat=latitudes,
                                   extent=map_extent,  dest_crs=proj_rp,
                                   extend_x=False, extend_y=True,
                                   image_res=img_res,  missing=missing)
    #for closeup image
    proj_inds_close = geo_tools.ProjInds(src_lon=longitudes, src_lat=latitudes,
                                         extent=map_extent_close,  dest_crs=proj_rp,
                                         extend_x=False, extend_y=True,
                                         image_res=img_res,  missing=missing)


    #
    #Reflectivity
    #
    #axes for this plot
```

(continues on next page)

(continued from previous page)

```
    pos = [sp_w, sp_h+(sp_h+rec_h), rec_w, rec_h]
    ax = fig.add_axes(pos, projection=proj_rp)
    ax.spines['geo'].set_linewidth(0.3)
    ax.set_extent(map_extent)
    ax.set_aspect('auto')

    ax.annotate('Qced Reflectivity', size=30,
                xy=(xp, yp), xycoords='axes fraction')

    # colormapping object for reflectivity
    map_reflectivity = legs.PalObj(range_arr=[0.,60],
                                   n_col=6,
                                   over_high='extend',
                                   under_low='white',
                                   excep_val=[missing,       undetect],
                                   excep_col=[missing_color, undetect_color])

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(reflectivity)

    #plot data & palette
    map_reflectivity.plot_data(ax=ax, data=proj_data, zorder=0,
                               palette='right',
                               pal_units='[dBZ]', pal_format='{:3.0f}')

    #add political boundaries
    add_feature(ax)

    #radar circles and azimuths
    radar_ax_circ(ax, radar_lat, radar_lon)

    #circle indicating closeup area
    ax.plot(closeup_lons, closeup_lats, transform=proj_pc, c=(0.,0.,0.), zorder=300,␣
↪linewidth=.8)

    #arrow pointing to closeup
    ax.annotate("", xy=(0.33, 0.67), xytext=(.55, .74),  xycoords='axes fraction',
                arrowprops=dict(arrowstyle="<-"))

    #
    #Closeup of reflectivity
    #
    pos = [sp_w+x0*rec_w, sp_h+(sp_h+rec_h)+y0*rec_h, dx*rec_w, dy*rec_h]
    ax2 = fig.add_axes(pos, projection=proj_rp, label='reflectivity overlay')
    ax2.set_extent(map_extent_close)
    ax2.spines['geo'].set_linewidth(0.0)  #no border line
    ax2.set_facecolor((1.,1.,1.,0.))      #transparent background

    #geographical projection of data into axes space
    proj_data = proj_inds_close.project_data(reflectivity)

    #RGB values for data to plot
```

(continues on next page)

(continued from previous page)

```
    closeup_rgb = map_reflectivity.to_rgb(proj_data)

    #get corners of image in data space
    extent_data_space = ax2.get_extent()

    ## another way of doing the same thing is to get an object that convers axes coords␣
↪to data coords
    ## this method is more powerfull as it will return data coords of any points in axes␣
↪space
    #transform_data_to_axes = ax2.transData + ax2.transAxes.inverted()
    #transform_axes_to_data = transform_data_to_axes.inverted()
    #pts = ((0.,0.),(1.,1.)) #axes space coords
    #pt1, pt2 = transform_axes_to_data.transform(pts)
    #extent_data_space = [pt1[0],pt2[0],pt1[1],pt2[1]]

    #add alpha channel (transparency) to closeup image
    rgba = np.concatenate([closeup_rgb/255.,clip_alpha[...,np.newaxis]], axis=2)

    #plot image
    ax2.imshow(rgba, interpolation='nearest',
               origin='upper', extent=extent_data_space, zorder=100)
    ax2.set_aspect('auto')

    #circle indicating closeup area
    circle = ax2.plot(closeup_lons, closeup_lats, transform=proj_pc, c=(0.,0.,0.),␣
↪zorder=300, linewidth=1.5)
    #prevent clipping of the circle we just drawn
    for line in ax2.lines:
        line.set_clip_on(False)


    #
    #Quality Controlled Doppler velocity
    #
    #axes for this plot
    pos = [sp_w+(sp_w+rec_w+1./fig_w), sp_h+(sp_h+rec_h), rec_w, rec_h]
    ax = fig.add_axes(pos, projection=proj_rp)
    ax.set_extent(map_extent)
    ax.set_aspect('auto')

    ax.annotate('Qced Doppler velocity', size=30,
                xy=(xp, yp), xycoords='axes fraction')

    #from https://colorbrewer2.org
    brown_purple=[[ 45,  0, 75],
                  [ 84, 39,136],
                  [128,115,172],
                  [178,171,210],
                  [216,218,235],
                  [247,247,247],
                  [254,224,182],
                  [253,184, 99],
```

(continues on next page)

(continued from previous page)

```
                  [224,130, 20],
                  [179, 88,  6],
                  [127, 59,  8]]
    range_arr = [-48.,-40.,-30.,-20,-10.,-1.,1.,10.,20.,30.,40.,48.]

    map_dvel = legs.PalObj(range_arr=range_arr,
                           color_arr = brown_purple,
                           solid='supplied',
                           excep_val=[missing, undetect],
                           excep_col=[missing_color, undetect_color])

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(doppvelocity_qc)

    #plot data & palette
    map_dvel.plot_data(ax=ax,data=proj_data, zorder=0,
                       palette='right', pal_units='[m/s]', pal_format='{:3.0f}')
↪#palette options

    #add political boundaries
    add_feature(ax)

    #radar circles and azimuths
    radar_ax_circ(ax, radar_lat, radar_lon)

    #circle indicating closeup area
    ax.plot(closeup_lons, closeup_lats, transform=proj_pc, c=(0.,0.,0.), zorder=300,␣
↪linewidth=.8)

    #arrow pointing to closeup
    ax.annotate("", xy=(0.33, 0.67), xytext=(.55, .74),  xycoords='axes fraction',
                arrowprops=dict(arrowstyle="<-"))

    #
    #Closeup of Doppler velocity
    #
    pos = [sp_w+1.*(sp_w+rec_w+1./fig_w)+x0*rec_w, sp_h+(sp_h+rec_h)+y0*rec_h, dx*rec_w,␣
↪dy*rec_h]
    ax2 = fig.add_axes(pos, projection=proj_rp, label='overlay')
    ax2.set_extent(map_extent_close)
    ax2.spines['geo'].set_linewidth(0.0) #no border line
    ax2.set_facecolor((1.,1.,1.,0.))     #transparent background

    #geographical projection of data into axes space
    proj_data = proj_inds_close.project_data(doppvelocity_qc)

    #RGB values for data to plot
    closeup_rgb = map_dvel.to_rgb(proj_data)

    #get corners of image in data space
    extent_data_space = ax2.get_extent()
```

(continues on next page)

(continued from previous page)

```
    #add alpha channel (transparency) to closeup image
    rgba = np.concatenate([closeup_rgb/255.,clip_alpha[...,np.newaxis]], axis=2)

    #plot image
    ax2.imshow(rgba, interpolation='nearest',
               origin='upper', extent=extent_data_space, zorder=100)
    ax2.set_aspect('auto')

    #line indicating closeup area
    circle = ax2.plot(closeup_lons, closeup_lats, transform=proj_pc, c=(0.,0.,0.),␣
↪zorder=300, linewidth=1.5)
    for line in ax2.lines:
        line.set_clip_on(False)

    #Show scale in inlay
    ax2.plot(scale_lons, scale_lats, transform=proj_pc, c=(0.,0.,0.), zorder=300,␣
↪linewidth=.8)
    ax2.annotate("5 km", size=18, xy=(.16, .25),  xycoords='axes fraction', zorder=310)


    #
    #DR
    #
    #axes for this plot
    pos = [sp_w, sp_h, rec_w, rec_h]
    ax = fig.add_axes(pos, projection=proj_rp)
    ax.set_extent(map_extent)
    ax.set_aspect('auto')

    ax.annotate('Depolarization ratio', size=30,
                xy=(xp, yp), xycoords='axes fraction')

    # Set up colormapping object
    map_dr = legs.PalObj(range_arr=[-36.,-24.,-12., 0.],
                         color_arr=['purple','blue','orange'],
                         dark_pos =['high',  'high','low'],
                         excep_val=[missing,       undetect],
                         excep_col=[missing_color, undetect_color])

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(dr)

    #plot data & palette
    map_dr.plot_data(ax=ax,data=proj_data, zorder=0,
                     palette='right', pal_units='[dB]', pal_format='{:3.0f}')

    #add political boundaries
    add_feature(ax)

    #radar circles and azimuths
    radar_ax_circ(ax, radar_lat, radar_lon)
```

(continues on next page)

(continued from previous page)

```
    #
    #Total quality index
    #
    #axes for this plot
    pos = [sp_w+(sp_w+rec_w+1./fig_w), sp_h, rec_w, rec_h]
    ax = fig.add_axes(pos, projection=proj_rp)
    ax.set_extent(map_extent)
    ax.set_aspect('auto')

    ax.annotate('Total quality index', size=30,
                xy=(xp, yp), xycoords='axes fraction')

    # Set up colormapping object
    pastel = [ [[255,190,187],[230,104, 96]],  #pale/dark red
               [[255,185,255],[147, 78,172]],  #pale/dark purple
               [[255,227,215],[205,144, 73]],  #pale/dark brown
               [[210,235,255],[ 58,134,237]],  #pale/dark blue
               [[223,255,232],[ 61,189, 63]] ] #pale/dark green
    map_qi = legs.PalObj(range_arr=[0., 1.],
                         dark_pos='high',
                         color_arr=pastel,
                         excep_val=[missing,       undetect],
                         excep_col=[missing_color, undetect_color])

    #geographical projection of data into axes space
    proj_data = proj_inds.project_data(tot_qi)

    #plot data & palette
    map_qi.plot_data(ax=ax,data=proj_data, zorder=0,
                     palette='right', pal_units='[unitless]', pal_format='{:3.1f}')
↪#palette options

    #add political boundaries
    add_feature(ax)

    #radar circles and azimuths
    radar_ax_circ(ax, radar_lat, radar_lon)

    ##uncomment to save figure
    #plt.savefig('radar_ppi.svg')

if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 7.588 seconds)

CHAPTER

EIGHT

INSTALLATION

- Conda installation is probbly the easiest.

```
conda install -c dja001 domutils
```

  Recent versions of cartopy have got super slow, I recommend the following combination for decent speed.

  - cartopy=0.19.0.post1
  - matplotlib=3.3.4

- Pip installation should also work

```
pip install domutils
```

- To use the domutils modules in your python code, load the different modules separately. For example:

```
>>> import domutils.legs as legs
>>> import domutils.geo_tools as geo_tools
```

CHAPTER

NINE

NEW FEATURE, BUG FIX, ETC.

If you want **domutils** to be modified in any way, start by opening an issue on github.

1. Create an issue on *domutils* github page. We will discuss the changes to be made and define a strategy for doing so.
2. Once the issue is created, fork the project. This will create your own github repo where you can make changes.
3. On your computer, clone the source code and go in the package directory

```
git clone git@github.com:<your-username>/domutils.git
cd domutils
```

4. Create a new branch whose name is related to the issue you opened at step 1 above. For example:

```
git checkout -b #666-include-cool-new-feature
```

5. Create a clean Anaconda development environment and activate it. You will need internet access for this.

```
conda env create --name domutils_dev_env -f docs/environment.yml
conda activate domutils_dev_env
```

6. It is a good practice to start by writing a unit test that will pass once your feature/bugfix is correctly implemented. Look in the

```
test/
```

   directories of the different _domutils_ modules for examples of such tests.

7. Modify the code to address the issue. Make sure to include examples and/or tests in the docstrings.
8. If applicable, describe the new functionality in the documentation.
9. Modify the files:

```
VERSION
CHANGELOG.md
```

   To reflect your changes.

10. Run unittest

```
python -m unittest discover
```

11. Run doctest

```
cd docs
make doctest
```

Make sure that there are no failures in the tests.

Note that the first time you run this command internet access is required as the test data will be downloaded from zenodo .

12. If you modified the documentation in functions docstrings, you probably want to check the changes by creating your local version of the documentation.

```
cd docs
make html
```

You can see the output in any web browser pointing to:

```
domutils/docs/_build/html/
```

13. While you are working, it is normal to commit changes several times on you local branch. However, before you push to your fork on github, it is probably a good idea to squash all you intermediate commits into one, or a few commits, that clearly link to the issue being worked on. The resulting squashed commit should pass the tests.

14. Once you are happy with the modifications, push the new version on your fork on github

```
git push -u origin #666-include-cool-new-feature
```

15. From the github web interface, create a pull request to me. We will then discuss the changes until they are accepted and merged into the master branch.

CHAPTER

TEN

TEST DATA

Data used in the examples and for running tests can be obtained by running

```
./download_test_data.sh
```

in the main directory of this package. This creates a *test_data/* directory containing all the test data.

CHAPTER

ELEVEN

CONTRIBUTORS

Great thanks to:

- To *Dikra Khedhaouiria* for
  - Including a stageIV reader to radar_tools
- To *Madalina Surcel* for
  - Her great contribution to geo_tools and
  - a review of the radar_tools section.

PYTHON MODULE INDEX

d

INDEX

R

T