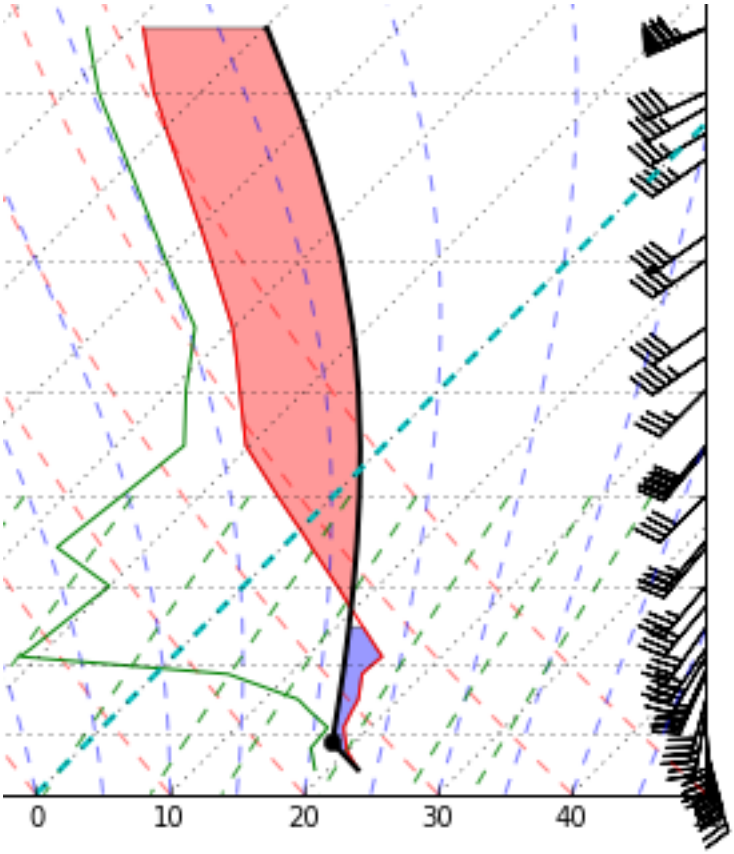
MetPy Documentation

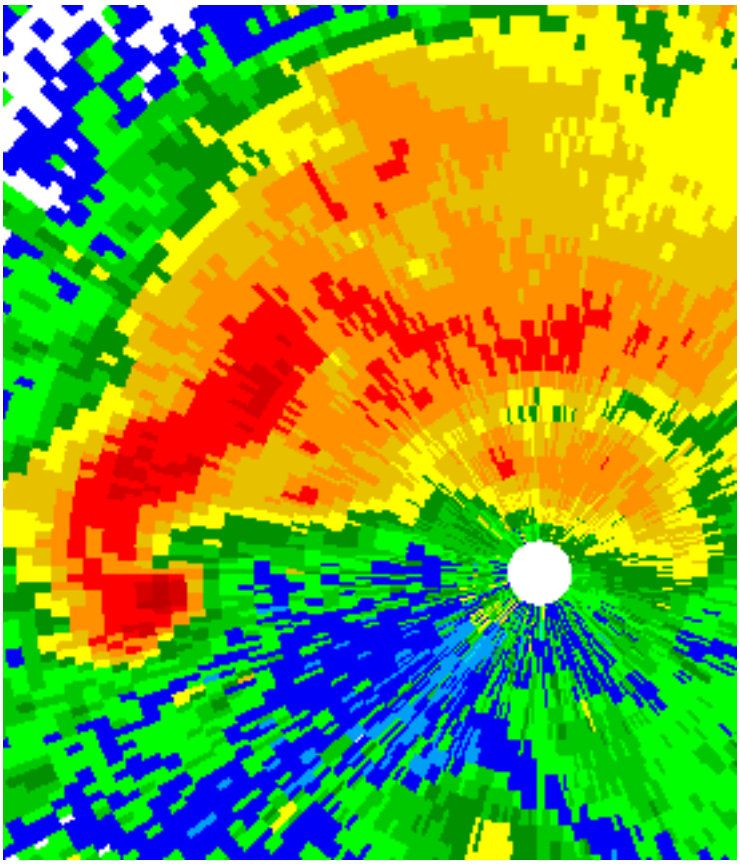
Release 0.4.2+0.g63a5224.dirty

MetPy Developers

November 18, 2016

Contents


0
10
20
30
40



MetPy is a collection of tools in Python for reading, visualizing, and performing calculations with weather data.

MetPy is still in an early stage of development, and as such **no APIs are considered stable.** While we won't break things just for fun, many things may still change as we work through design issues.

We support Python 2.7 as well as Python >= 3.3.

CHAPTER 1

Documentation

1.1 Installation Guide

1.1.1 Requirements

MetPy supports Python 2.7 as well as Python >= 3.3. Python 3.4 is the recommended version.

MetPy requires the following packages:

- NumPy >= 1.9.1
- SciPy >= 0.14.0
- Matplotlib >= 1.4.0
- pint >= 0.7

Installation Instructions for NumPy and SciPy can be found at: http://www.scipy.org/scipylib/download.html

Installation Instructions for Matplotlib can be found at: http://matplotlib.org/downloads.html

Pint is a pure python package and can be installed via `pip install pint`.

Python versions older than 3.4 require the enum34 package, which is a backport of the enum standard library module. It can be installed via `pip install enum34`.

PyProj is an optional dependency (if using the CDM interface to data files). It can also be installed via `pip install pyproj`, though it does require the Proj.4 library and a compiled extension.

1.1.2 Installation

The easiest way to install MetPy is through `pip`:

```
pip install metpy
```

If you are a user of the Conda pacakge manager, there are also up-to-date packages for MetPy (as well as its dependencies) available from the conda-forge channel:

```
conda install -c conda-forge metpy
```

The source code can also be grabbed from GitHub. From the base of the source directory, run:

```
pip install .
```

This will build and install MetPy into your current Python installation.

1.1.3 Examples

The MetPy source comes with a set of example IPython notebooks in the `examples/notebooks` directory. These can also be converted to standalone scripts (provided IPython is installed) using:

```
python setup.py examples
```

These examples are also seen within the documentation in the MetPy Examples.

1.2 Unit Support

To ensure correct calculations, MetPy relies upon the `pint` library to enforce unit-correctness. This simplifies the MetPy API by eliminating the need to specify units various functions. Instead, only the final results need to be converted to desired units. For more information on unit support, see the documentation for Pint. Particular attention should be paid to the support for temperature units.

1.2.1 Construction

To use units, the first step is to import the default MetPy units registry from the *`units`* module:

```
import numpy as np
from metpy.units import units
```

The unit registry encapsulates all of the available units, as well as any pertinent settings. The registry also understands unit prefixes and suffixes; this allows the registry to understand `'kilometer'` and `'meters'` in addition to the base `'meter'` unit.

In general, using units is only a small step on top of using the `numpy.ndarray` object. The easiest way to attach units to an array is to multiply by the units:

```
distance = np.arange(1, 5) * units.meters
```

It is also possible to directly construct a `pint.Quantity`, with a full units string:

```
time = units.Quantity(np.arange(2, 10, 2), 'sec')
```

Compound units can be constructed by the direct mathematical operations necessary:

```
g = 9.81 * units.meter / (units.second * units.second)
```

This verbose syntax can be reduced by using the unit registry's support for parsing units:

```
g = 9.81 * units('m/s^2')
```

1.2.2 Operations

With units attached, it is possible to perform mathematical operations, resulting in the proper units:

```
print(distance / time)
```

```
[ 0.5  0.5  0.5  0.5] meter / second
```

For multiplication and division, units can combine and cancel. For addition and subtraction, instead the operands must have compatible units. For instance, this works:

```
print(distance + distance)
```

```
[0 2 4 6 8] meter
```

But this does not:

```
print(distance + time)
```

```
DimensionalityError: Cannot convert from 'meter' ([length]) to 'second' ([time])
```

Even if the units are not identical, as long as they are dimensionally equivalent, the operation can be performed:

```
print(3 * units.inch + 5 * units.cm)
```

```
4.968503937007874 inch
```

1.2.3 Conversion

Converting a `Quantity` between units can be accomplished by using the `to()` method call, which constructs a new `Quantity` in the desired units:

```
print((1 * units.inch).to(units.mm))
```

```
25.400000000000002 millimeter
```

There is also the `ito()` method which performs the same operation in place. To simplify units, there is also the `to_base_units()` method, which converts a quantity to SI units, performing any needed cancellation:

```
Lf = 3.34e6 * units('J/kg')
print(Lf, Lf.to_base_units(), sep='\n')
```

```
3340000.0 joule / kilogram
3340000.0 meter ** 2 / second ** 2
```

`to_base_units()` can also be done in place via the `ito_base_units()` method.

1.3 MetPy Examples

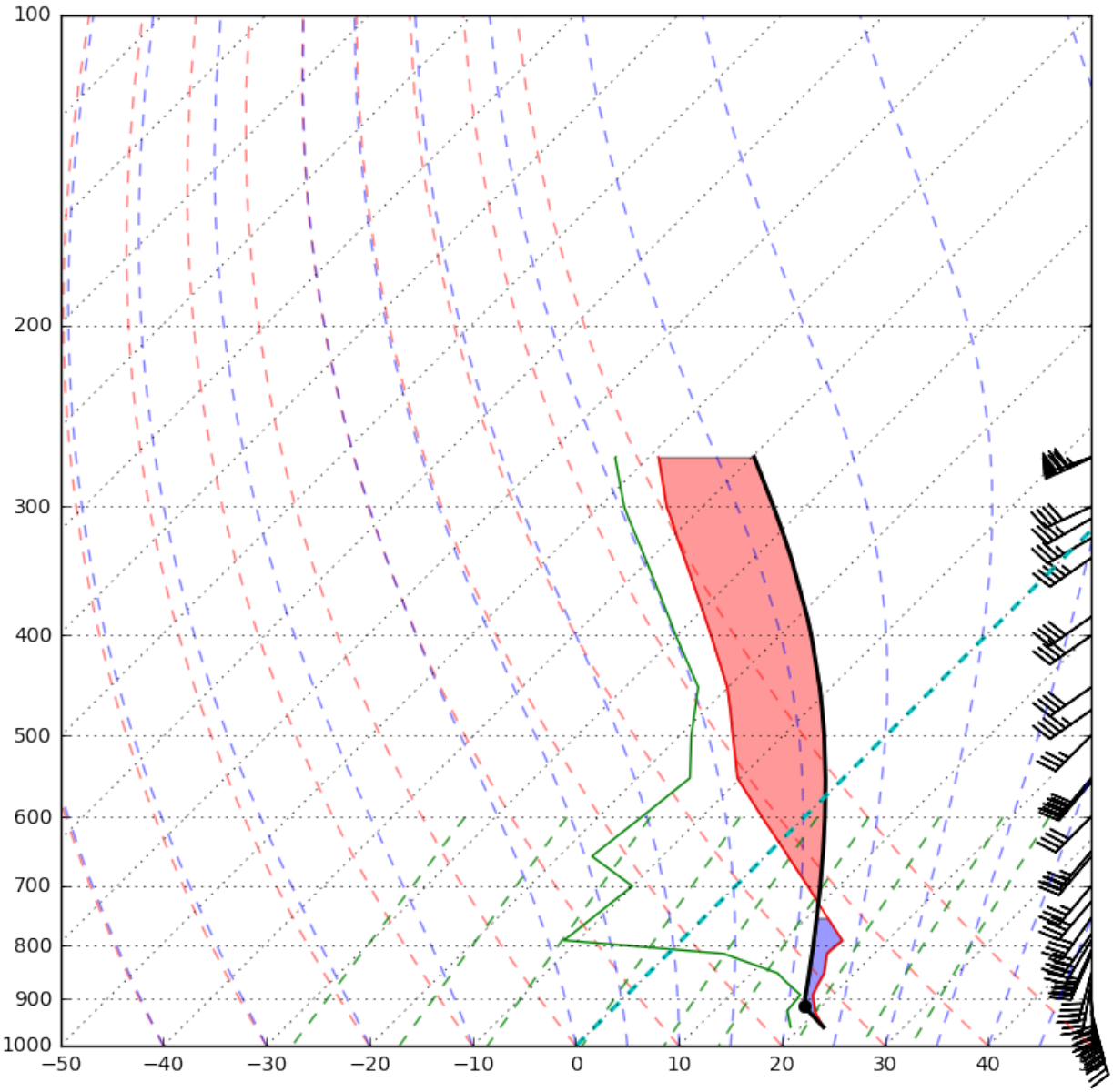
1.3.1 Advanced Sounding

Notebook

```
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np

from metpy.cbook import get_test_data
import metpy.calc as mpcalc
from metpy.io import get_upper_air_data
from metpy.plots import SkewT
from metpy.units import units, concatenate

%matplotlib inline
```

```
from metpy.io.upperair import UseSampleData
with UseSampleData():  # Only needed to use our local sample data
    # Download and parse the data
    dataset = get_upper_air_data(datetime(1999, 5, 4, 0), 'OUN')

p = dataset.variables['pressure'][:]
T = dataset.variables['temperature'][:]
Td = dataset.variables['dewpoint'][:]
u = dataset.variables['u_wind'][:]
v = dataset.variables['v_wind'][:]
```

```
# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(9, 9))
skew = SkewT(fig, rotation=45)

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
skew.plot(p, T, 'r')
skew.plot(p, Td, 'g')
skew.plot_barbs(p, u, v)
skew.ax.set_ylim(1000, 100)
skew.ax.set_xlim(-40, 60)

# Calculate LCL height and plot as black dot
l = mpcalc.lcl(p[0], T[0], Td[0])
lcl_temp = mpcalc.dry_lapse(concatenate((p[0], l)), T[0])[-1].to('degC')
skew.plot(l, lcl_temp, 'ko', markerfacecolor='black')

# Calculate full parcel profile and add to plot as black line
prof = mpcalc.parcel_profile(p, T[0], Td[0]).to('degC')
skew.plot(p, prof, 'k', linewidth=2)

# Example of coloring area between profiles
skew.ax.fill_betweenx(p, T, prof, where=T>=prof, facecolor='blue', alpha=0.4)
skew.ax.fill_betweenx(p, T, prof, where=T<prof, facecolor='red', alpha=0.4)

# An example of a slanted line at constant T -- in this case the 0
# isotherm
l = skew.ax.axvline(0, color='c', linestyle='--', linewidth=2)

# Add the relevant special lines
skew.plot_dry_adiabats()
skew.plot_moist_adiabats()
skew.plot_mixing_lines()

# Show the plot
plt.show()
```



1.3.2 Dewpoint and Mixing Ratio

Notebook

The goal of this notebook is to show an example using the units support in MetPy. In this example, we calculate the dewpoint, corresponding to a fixed value of mixing ratio, at two different surface pressure values.

```
# First import our calculation functions, as well as unit support
import metpy.calc as mcalc
from metpy.units import units
```

```
# Create a test value of mixing ratio in grams per kilogram
mixing = 10 * units('g/kg')
print(mixing)
```

```
10.0 gram / kilogram
```

```
# Now throw that value with units into the function to calculate
# the corresponding vapor pressure, given a surface pressure of 1000 mb
e = mcalc.vapor_pressure(1000. * units.mbar, mixing)
print(e)
```

```
15825.67178529092 gram * millibar / kilogram
```

```
# Take the odd units and force them to millibars
print(e.to(units.mbar))
```

```
15.82567178529092 millibar
```

```
# Take the raw vapor pressure and throw into the dewpoint function
td = mcalc.dewpoint(e)
print(td)
```

```
13.856458659577921 degC
```

```
# Which can of course be converted to Farenheit
print(td.to('degF'))
```

```
56.94162598724023 degF
```

```
# Now do the same thing for 850 mb, approximately the pressure of Denver
e = mcalc.vapor_pressure(850. * units.mbar, mixing)
print(e.to(units.mbar))
```

```
13.451821017497283 millibar
```

```
# And print the corresponding dewpoint
td = mcalc.dewpoint(e)
print(td, td.to('degF'))
```

```
11.378824018637602 degC 52.48188363354765 degF
```

1.3.3 Find Natural Neighbors Verification

Notebook

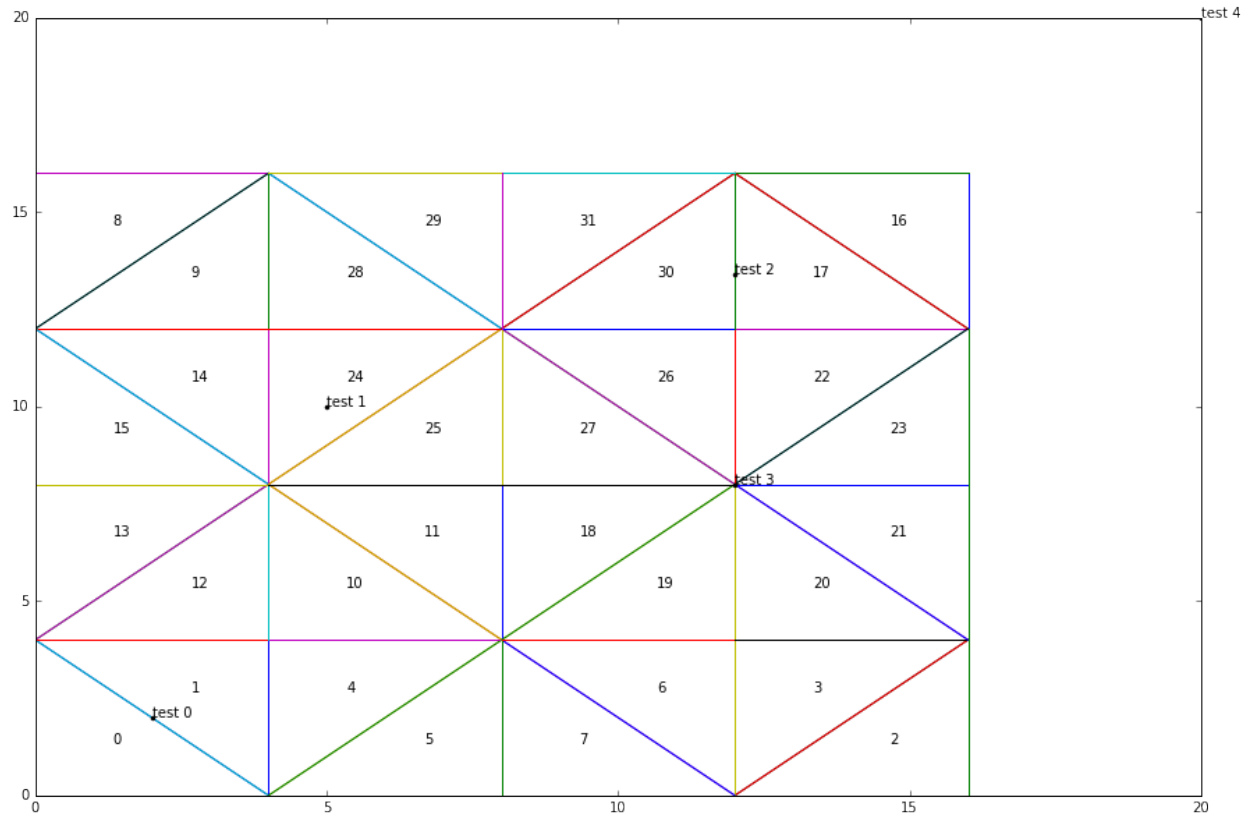
Finding natural neighbors in a triangulation

A triangle is a natural neighbor of a point if that point is within a circumradius of the circumcenter of a circumscribed circle containing the triangle.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (15, 10)
from scipy.spatial import Delaunay

from metpy.gridding.triangles import find_natural_neighbors
```

Create test observations, test points, and plot the triangulation and points.

```
x = list(range(0, 20, 4))
y = list(range(0, 20, 4))
gx, gy = np.meshgrid(x, y)
pts = np.vstack([gx.ravel(), gy.ravel()]).T
tri = Delaunay(pts)

grids = np.random.randint(0, 10, (5, 2))

for i in range(len(tri.simplices)):

    x, y = tri.points[tri.simplices[i]].T

    for j in range(3):
        plt.plot([x[j], x[(j+1)%3]], [y[j], y[(j+1)%3]])

    xave = np.mean(x)
    yave = np.mean(y)

    plt.annotate(str(i), xy=(xave, yave))

test_points = np.array([[2, 2], [5, 10], [12, 13.4], [12, 8], [20, 20]])

for i in range(len(test_points)):
    x = test_points[i][0]
    y = test_points[i][1]

    plt.plot(x, y, "k.", markersize=6)
    plt.annotate("test " + str(i), xy=(x, y))
```



Since finding natural neighbors already calculates circumcenters and circumradii, return that information for later use.

The key of the neighbors dictionary refers to the test point index, and the list of integers are the triangles that are natural neighbors of that particular test point.

Since point 4 is far away from the triangulation, it has no natural neighbors. Point 3 is at the confluence of several triangles so it has many natural neighbors.

```
neighbors, tri_info = find_natural_neighbors(tri, test_points)

neighbors
```

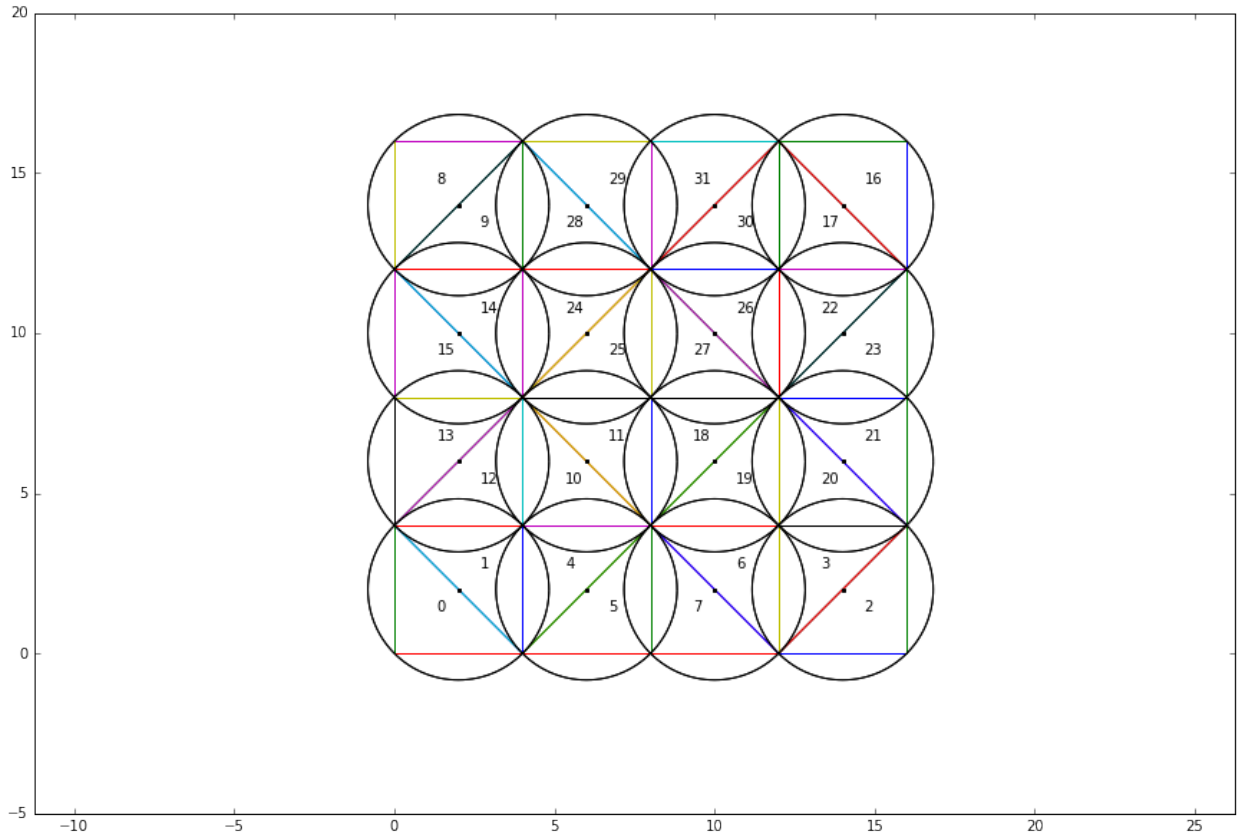
```
{0: [0, 1],
 1: [24, 25],
 2: [16, 17, 30, 31],
 3: [18, 19, 20, 21, 22, 23, 26, 27],
 4: []}
```

We can then use the information in tri_info later.

The dictionary key is the index of a particular triangle in the Delaunay triangulation data structure. 'cc' is that triangle's circumcenter, and 'r' is the radius of the circumcircle containing that triangle.

Using circumcenter and radius information from tri_info, plot circumcircles and circumcenters for each triangle.

```
def draw_circle(x, y, r, m, label):

    nx = x + r * np.cos(np.deg2rad(list(range(360))))
    ny = y + r * np.sin(np.deg2rad(list(range(360))))

    plt.plot(nx, ny, m, label=label)

for i in range(len(tri.simplices)):

    x, y = tri.points[tri.simplices[i]].T

    for j in range(3):
        plt.plot([x[j], x[(j+1)%3]], [y[j], y[(j+1)%3]])

    xave = np.mean(x)
    yave = np.mean(y)

    plt.annotate(str(i), xy=(xave, yave))

for idx, (item) in tri_info.items():

    plt.plot(item['cc'][0], item['cc'][1], "k.", markersize=5)
    draw_circle(item['cc'][0], item['cc'][1], item['r'], "k-", "")

plt.axes().set_aspect('equal', 'datalim')
```



1.3.4 GINI Water Vapor

Notebook

```
import numpy as np
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

from metpy.cbook import get_test_data
from metpy.io.gini import GiniFile
from metpy.plots.ctables import registry

%matplotlib inline
```

```
# Open the GINI file from the test data
f = GiniFile(get_test_data('WEST-CONUS_4km_WV_20151208_2200.gini'))
print(f)
```

```
GiniFile: GOES-15 West CONUS WV (6.5/6.7 micron)
    Time: 2015-12-08 22:00:19
    Size: 1280x1100
    Projection: lambert_conformal
    Lower Left Corner (Lon, Lat): (-133.4588, 12.19)
    Resolution: 4km
```

```
# Get a Dataset view of the data (essentially a NetCDF-like interface to the
# underlying data). Pull out the data, (x, y) coordinates, and the projection
```

```
# information.
ds = f.to_dataset()
x = ds.variables['x'][:]
y = ds.variables['y'][:]
dat = ds.variables['WV']
proj_var = ds.variables[dat.grid_mapping]
print(proj_var)
```

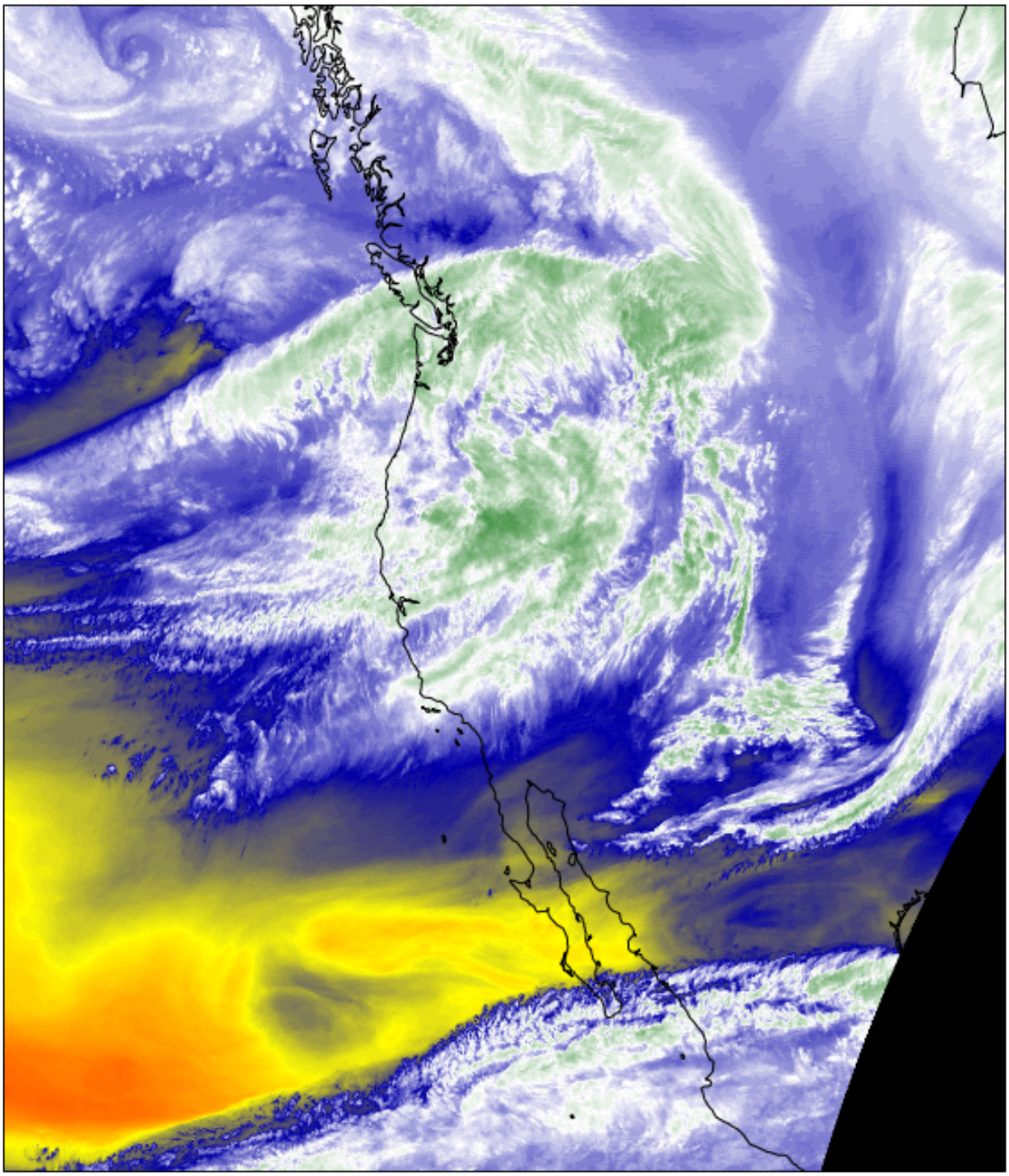
```
<class 'metpy.io.cdm.Variable'>: int32 Lambert_Conformal()
    grid_mapping_name: lambert_conformal_conic
    standard_parallel: 25.0
    longitude_of_central_meridian: -95.0
    latitude_of_projection_origin: 25.0
    earth_radius: 6371200.0
```

```
# Create CartoPy projection information for the file
globe = ccrs.Globe(ellipse='sphere', semimajor_axis=proj_var.earth_radius,
                   semiminor_axis=proj_var.earth_radius)
proj = ccrs.LambertConformal(central_longitude=proj_var.longitude_of_central_meridian,
                             central_latitude=proj_var.latitude_of_projection_origin,
                             standard_parallels=[proj_var.standard_parallel],
                             globe=globe)
```

```
# Plot the image
fig = plt.figure(figsize=(10, 20))
ax = fig.add_subplot(1, 1, 1, projection=proj)
wv_norm, wv_cmap = registry.get_with_steps('WVCIMSS', 0, 1)
im = ax.imshow(dat[:], cmap=wv_cmap, norm=wv_norm, zorder=0,
               extent=(x.min(), x.max(), y.min(), y.max()), origin='upper')
ax.coastlines(resolution='50m', zorder=2, color='black')
```

```
<cartopy.mpl.feature_artist.FeatureArtist at 0x10a62bcf8>
```

```
/Users/rmay/miniconda3/envs/metpy3/lib/python3.4/site-packages/matplotlib/artist.py:221: MatplotlibDe
axes property.  A removal date has not been set.
  warnings.warn(_get_axes_msg, mplDeprecation, stacklevel=1)
```



1.3.5 Hodograph Inset

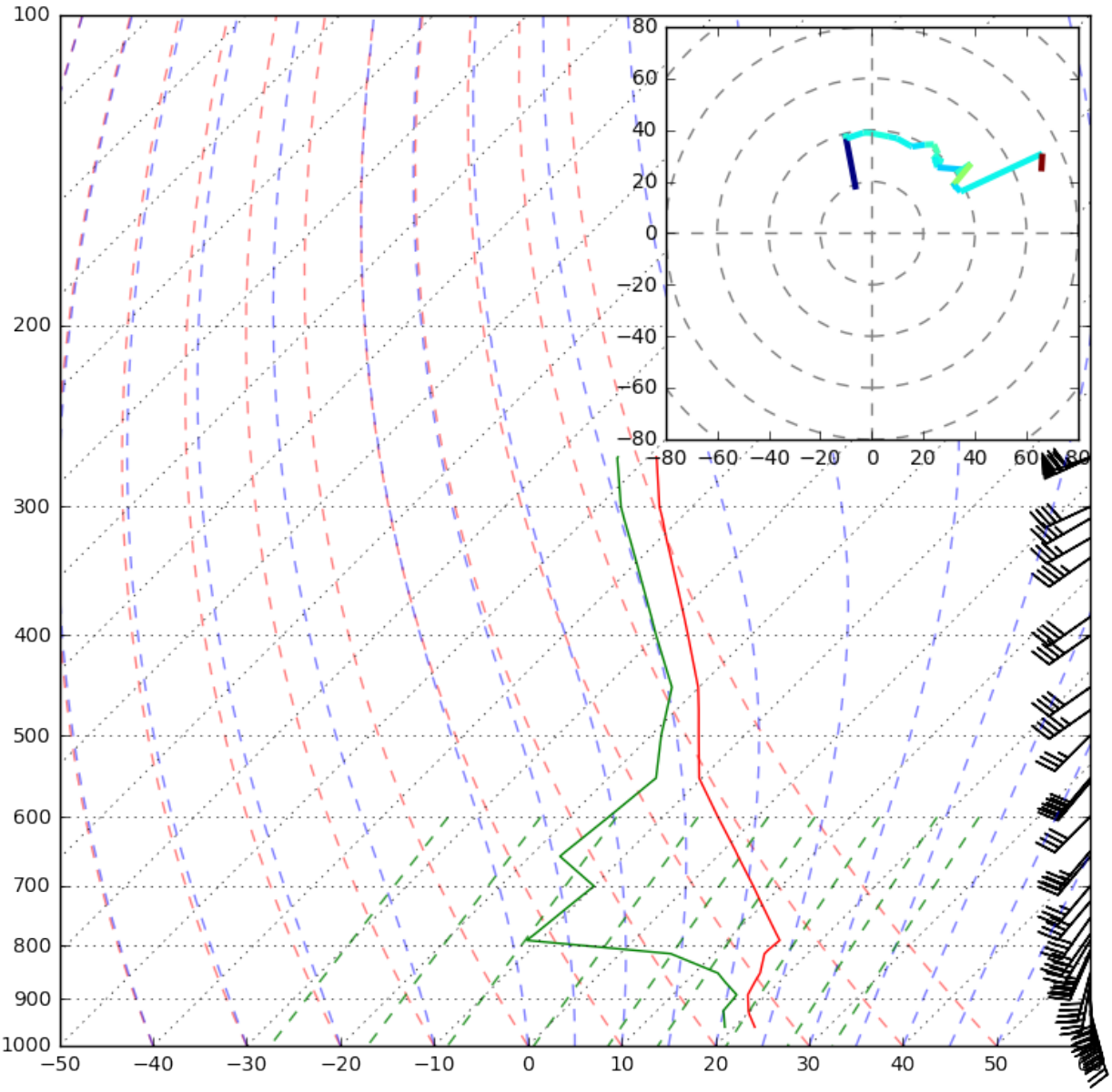
Notebook

```
from datetime import datetime

import numpy as np
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

from metpy.io import get_upper_air_data
from metpy.plots import SkewT, Hodograph
from metpy.units import units

%matplotlib inline
```

```
from metpy.io.upperair import UseSampleData
with UseSampleData():  # Only needed to use our local sample data
    # Download and parse the data
    dataset = get_upper_air_data(datetime(1999, 5, 4, 0), 'OUN')

p = dataset.variables['pressure'][:]
T = dataset.variables['temperature'][:]
Td = dataset.variables['dewpoint'][:]
u = dataset.variables['u_wind'][:]
v = dataset.variables['v_wind'][:]
```

```
# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(9, 9))

# Grid for plots
skew = SkewT(fig, rotation=45)

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
skew.plot(p, T, 'r')
skew.plot(p, Td, 'g')
skew.plot_barbs(p, u, v)
skew.ax.set_ylim(1000, 100)

# Add the relevant special lines
skew.plot_dry_adiabats()
skew.plot_moist_adiabats()
skew.plot_mixing_lines()

# Good bounds for aspect ratio
skew.ax.set_xlim(-50, 60)

# Create a hodograph
ax_hod = inset_axes(skew.ax, '40%', '40%', loc=1)
h = Hodograph(ax_hod, component_range=80.)
h.add_grid(increment=20)
h.plot_colormapped(u, v, np.hypot(u, v))

# Show the plot
plt.show()
```



1.3.6 Inverse Distance Verification

Notebook

Inverse Distance Verification: Cressman and Barnes

Two popular interpolation schemes that use inverse distance weighting of observations are the Barnes and Cressman analyses. The Cressman analysis is relatively straightforward and uses the ratio between distance of an observation from a grid cell and the maximum allowable distance to calculate the relative importance of an observation for calculating an interpolation value. Barnes uses the inverse exponential ratio of each distance between an observation and a grid cell and the average spacing of the observations over the domain.

Algorithmically:

1. A KDTree data structure is built using the locations of each observation.
2. All observations within a maximum allowable distance of a particular grid cell are found in O(log n) time.
3. Using the weighting rules for cressman or barnes analyses, the observations are given a proportional value, primarily based on their distance from the grid cell.
4. The sum of these proportional values is calculated and this value is used as the interpolated value.
5. Steps 2 through 4 are repeated for each grid cell.

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.spatial import cKDTree
from scipy.spatial.distance import cdist

from metpy.gridding import points
from metpy.gridding.gridding_functions import calc_kappa
from metpy.gridding.interpolation import cressman_point, barnes_point
from metpy.gridding.triangles import dist_2

%matplotlib inline
plt.rcParams['figure.figsize'] = (15, 10)

def draw_circle(x, y, r, m, label):

    nx = x + r * np.cos(np.deg2rad(list(range(360))))
    ny = y + r * np.sin(np.deg2rad(list(range(360))))

    plt.plot(nx, ny, m, label=label)
```

Generate random x and y coordinates, and observation values proportional to x * y.

Set up two test grid locations at (30, 30) and (60, 60).

```
np.random.seed(100)

pts = np.random.randint(0, 100, (10, 2))
xp = pts[:, 0]
yp = pts[:, 1]
z = (pts[:, 0] * pts[:, 0]) / 1000

sim_gridx = [30, 60]
sim_gridy = [30, 60]
```

Set up a cKDTree object and query all of the observations within “radius” of each grid point.

The variable “indices” represents the index of each matched coordinate within the cKDTree’s “data” list.

```
obs_tree = cKDTree(list(zip(xp, yp)))

grid_points = np.array(list(zip(sim_gridx, sim_gridy)))

radius = 40

indices = obs_tree.query_ball_point(grid_points, r=radius)
```

For grid 0, we will use cressman to interpolate its value.

```
x1, y1 = obs_tree.data[indices[0]].T
cress_dist = dist_2(sim_gridx[0], sim_gridy[0], x1, y1)
```

```
cress_obs = z[indices[0]]

cress_val = cressman_point(cress_dist, cress_obs, radius)
```

For grid 1, we will use barnes to interpolate its value.

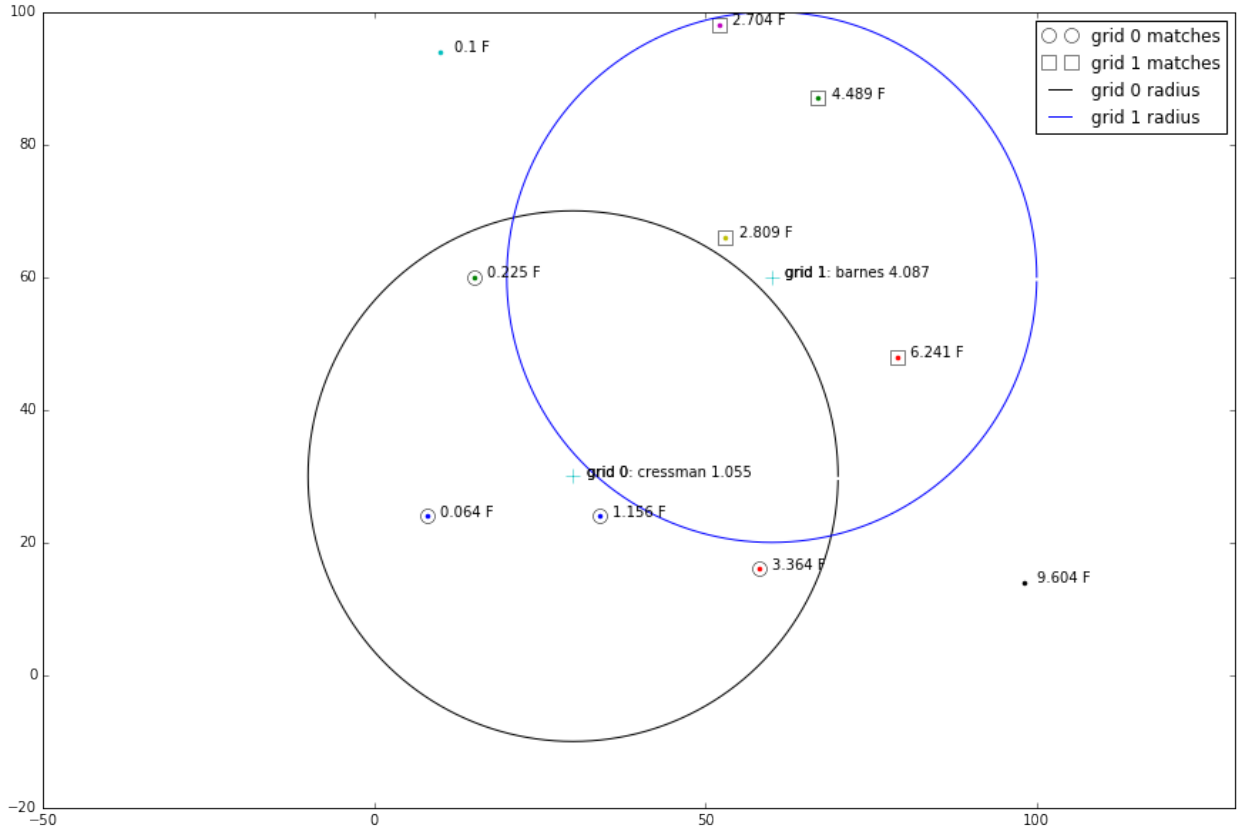
We need to calculate kappa–the average distance between observations over the domain.

```
x2, y2 = obs_tree.data[indices[1]].T
barnes_dist = dist_2(sim_gridx[1], sim_gridy[1], x2, y2)
barnes_obs = z[indices[1]]

ave_spacing = np.mean((cdist(list(zip(xp, yp)), list(zip(xp, yp)))))

kappa = calc_kappa(ave_spacing)

barnes_val = barnes_point(barnes_dist, barnes_obs, kappa)
```

Plot all of the affilitated information and interpolation values.

```
for i in range(len(z)):
    plt.plot(pts[i, 0], pts[i, 1], ".")
    plt.annotate(str(z[i]) + " F", xy=(pts[i, 0]+2, pts[i, 1]))

plt.plot(sim_gridx, sim_gridy, "+", markersize=10)

plt.annotate("grid 0", xy = (sim_gridx[0]+2, sim_gridy[0]))
plt.annotate("grid 1", xy = (sim_gridx[1]+2, sim_gridy[1]))

plt.plot(x1, y1, "ko", fillstyle='none', markersize=10, label="grid 0 matches")
plt.plot(x2, y2, "ks", fillstyle='none', markersize=10, label="grid 1 matches")

draw_circle(sim_gridx[0], sim_gridy[0], m="k-", r=radius, label="grid 0 radius")
draw_circle(sim_gridx[1], sim_gridy[1], m="b-", r=radius, label="grid 1 radius")

plt.annotate("grid 0: cressman " + ("%.3f" % cress_val),  xy = (sim_gridx[0]+2, sim_gridy[0]))
plt.annotate("grid 1: barnes "   + ("%.3f" % barnes_val), xy = (sim_gridx[1]+2, sim_gridy[1]))

plt.axes().set_aspect('equal', 'datalim')
plt.legend()
```

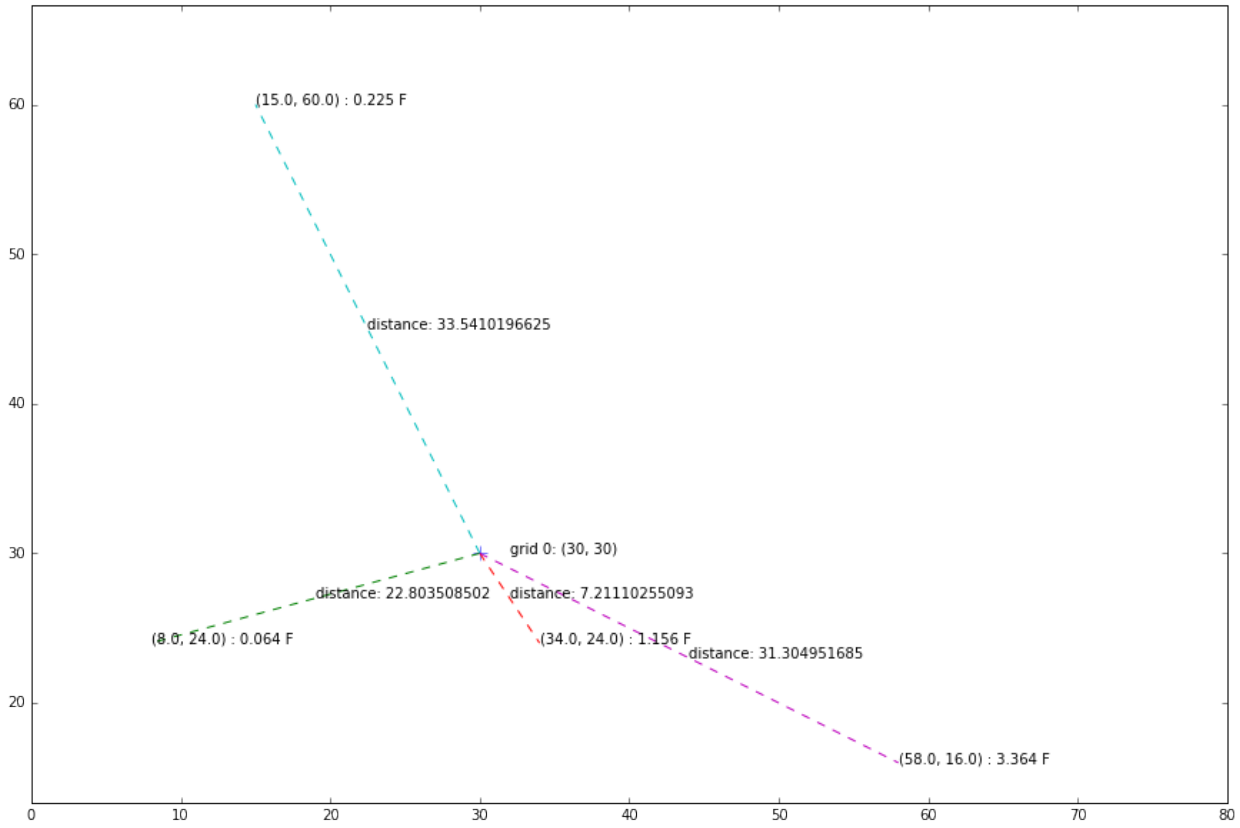
```
<matplotlib.legend.Legend at 0x1e5cee189b0>
```



For each point, we will do a manual check of the interpolation values by doing a step by step and visual breakdown.

Plot the grid point, observations within radius of the grid point, their locations, and their distances from the grid point.

```
plt.annotate("grid 0: " + "(" + str(sim_gridx[0]) + ", " + str(sim_gridy[0]) + ")", xy = (sim_gridx[0
plt.plot(sim_gridx[0], sim_gridy[0], "+", markersize=10)

mx, my = obs_tree.data[indices[0]].T
mz = z[indices[0]]

for i in range(len(mz)):
    d = np.sqrt((sim_gridx[0]-mx[i])**2 + (my[i]-sim_gridy[0])**2)
    plt.plot([sim_gridx[0], mx[i]], [sim_gridy[0], my[i]], "--")

    xave = np.mean([sim_gridx[0], mx[i]])
    yave = np.mean([sim_gridy[0], my[i]])

    plt.annotate("distance: " + str(d), xy=(xave, yave))

    plt.annotate("(" + str(mx[i]) + ", " + str(my[i]) + ") : " + str(mz[i]) + " F", xy=(mx[i], my[i])

plt.xlim(0, 80)
plt.ylim(0, 80)
plt.axes().set_aspect('equal', 'datalim')
```



Step through the cressman calculations.

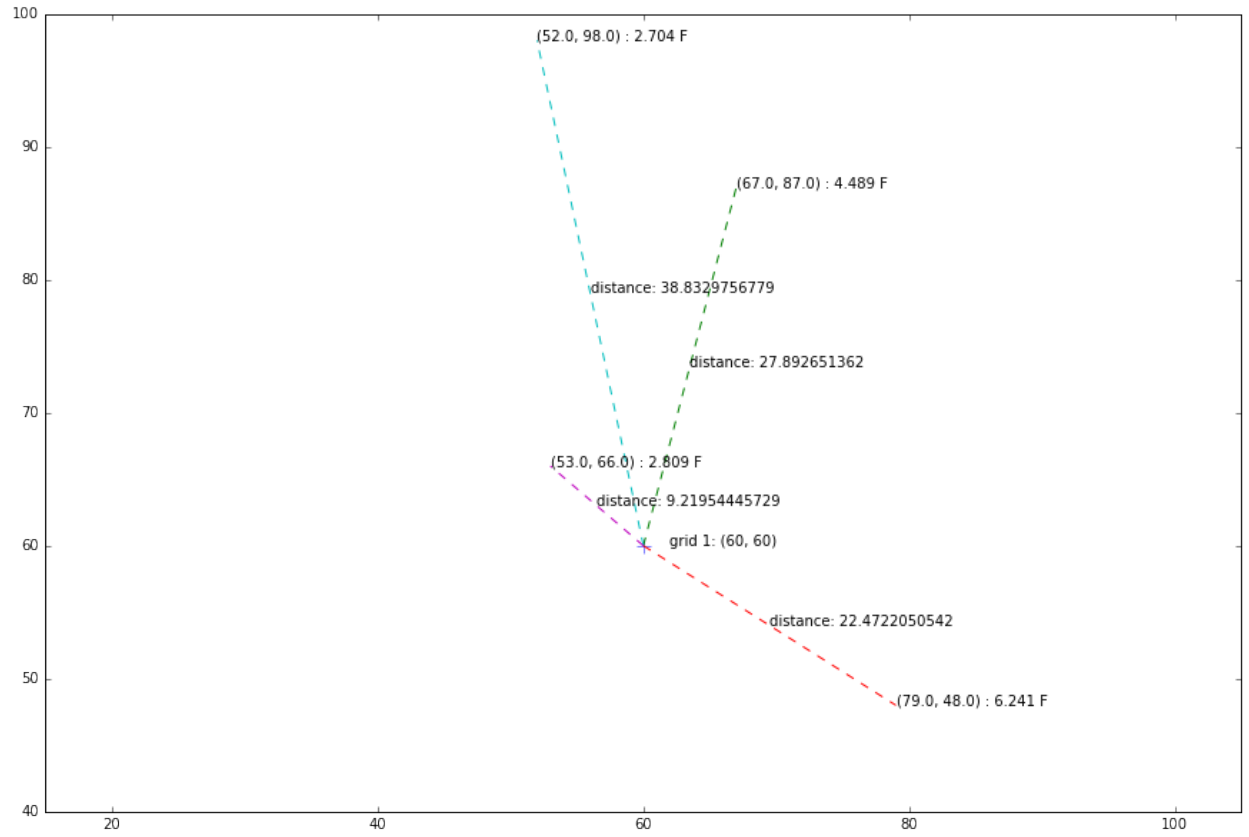
```
dists = np.array([22.803508502, 7.21110255093, 31.304951685, 33.5410196625])

values = np.array([0.064, 1.156, 3.364, 0.225])

cres_weights = (radius*radius - dists*dists) / (radius*radius + dists*dists)

total_weights = np.sum(cres_weights)

proportion = cres_weights / total_weights

value = values * proportion

val = cressman_point(cress_dist, cress_obs, radius)

print("Manual cressman value for grid 1: ", np.sum(value))

print("Metpy cressman value for grid 1:  ", val)
```

```
Manual cressman value for grid 1:  1.05499444404
Metpy cressman value for grid 1:   1.05499444404
```

Now repeat for grid 1, except use barnes interpolation.

```
plt.annotate("grid 1: " + "(" + str(sim_gridx[1]) + ", " + str(sim_gridy[1]) + ")", xy = (sim_gridx[1
plt.plot(sim_gridx[1], sim_gridy[1], "+", markersize=10)

mx, my = obs_tree.data[indices[1]].T
mz = z[indices[1]]
```

```
for i in range(len(mz)):
    d = np.sqrt((sim_gridx[1]-mx[i])**2 + (my[i]-sim_gridy[1])**2)
    plt.plot([sim_gridx[1], mx[i]], [sim_gridy[1], my[i]], "--")

    xave = np.mean([sim_gridx[1], mx[i]])
    yave = np.mean([sim_gridy[1], my[i]])

    plt.annotate("distance: " + str(d), xy=(xave, yave))

    plt.annotate("(" + str(mx[i]) + ", " + str(my[i]) + ") : " + str(mz[i]) + " F", xy=(mx[i], my[i])

plt.xlim(40, 80)
plt.ylim(40, 100)
plt.axes().set_aspect('equal', 'datalim')
```



Step through barnes calculations.

```
dists = np.array([9.21954445729, 22.4722050542, 27.892651362, 38.8329756779])

values = np.array([2.809, 6.241, 4.489, 2.704])

weights = np.exp(-dists**2 / kappa)

total_weights = np.sum(weights)
value = np.sum(values * (weights / total_weights))

val = barnes_point(barnes_dist, barnes_obs, kappa)

print("Manual barnes value: ", value)
```

```
print("Metpy barnes value:  ", val)
```

```
Manual barnes value:  4.08718241061
Metpy barnes value:   4.08718241061
```

1.3.7 NEXRAD Level 2 File

Notebook

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import ma

from metpy.cbook import get_test_data
from metpy.io.nexrad import Level2File
from metpy.plots import ctables

%matplotlib inline
```

```
# Open the file
name = get_test_data('KTLX20130520_201643_V06.gz', as_file_obj=False)
f = Level2File(name)
```

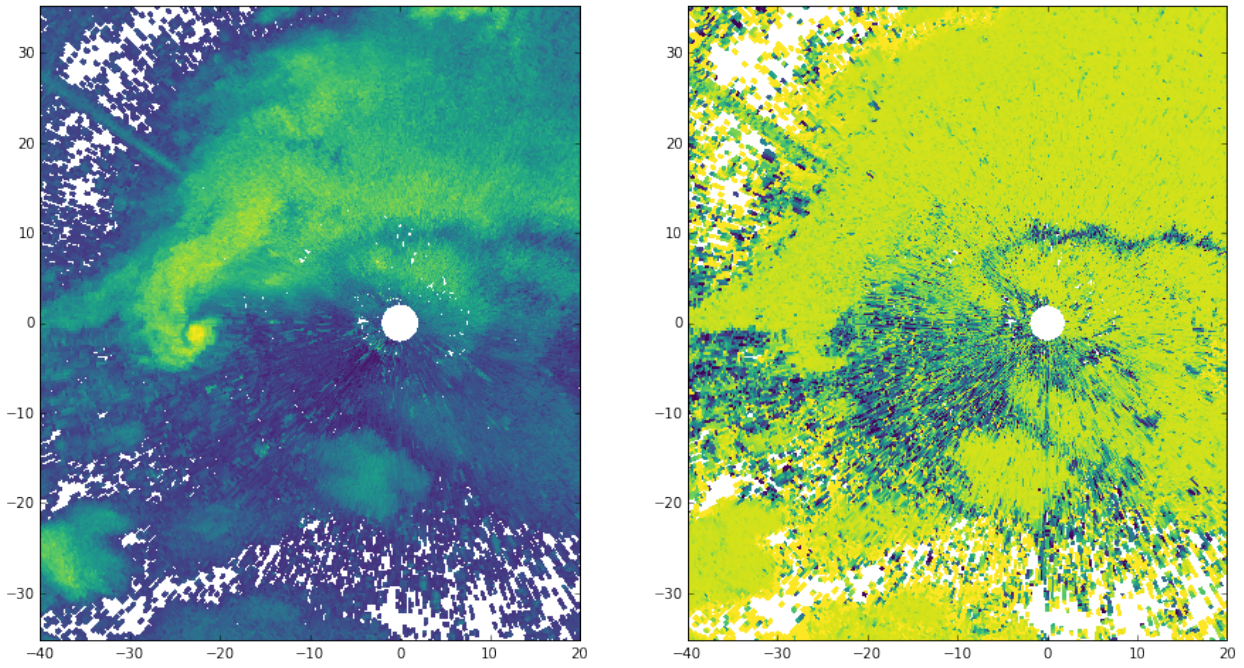
```
f.sweeps[0][0]
```

```
(Msg31DataHdr(stid=b'KTLX', time_ms=73003850, date=15846, az_num=1, az_angle=123.20343017578125, comp
 VolConsts(type=b'R', name=b'VOL', size=44, major=1, minor=0, lat=35.33305740356445, lon=-97.2774810
 ElConsts(type=b'R', name=b'ELV', size=12, atmos_atten=-0.012, calib_dbz0=-42.4375),
 RadConstsV1(type=b'R', name=b'RAD', size=20, unamb_range=466.0, noise_h=-79.71426391601562, noise_v=
 {b'PHI': (DataBlockHdr(type=b'D', name=b'PHI', reserved=0, num_gates=1192, first_gate=2.125, gate_wi
   array([ 32.79150829,  41.60642988,  30.32333025, ...,          nan,
                  nan,          nan])),
  b'REF': (DataBlockHdr(type=b'D', name=b'REF', reserved=0, num_gates=1832, first_gate=2.125, gate_wi
   array([  6.5,   2.5,  11. , ...,   nan,   nan,   nan])),
  b'RHO': (DataBlockHdr(type=b'D', name=b'RHO', reserved=0, num_gates=1192, first_gate=2.125, gate_wi
   array([ 0.995     ,  0.95833333,  0.99833333, ...,         nan,
                nan,          nan])),
  b'ZDR': (DataBlockHdr(type=b'D', name=b'ZDR', reserved=0, num_gates=1192, first_gate=2.125, gate_wi
   array([ 2.375 ,  6.1875,  3.75  , ...,     nan,     nan,     nan]))})
```

```
# Pull data out of the file
sweep = 0

# First item in ray is header, which has azimuth angle
az = np.array([ray[0].az_angle for ray in f.sweeps[sweep]])

# 5th item is a dict mapping a var name (byte string) to a tuple
# of (header, data array)
ref_hdr = f.sweeps[sweep][0][4][b'REF'][0]
ref_range = np.arange(ref_hdr.num_gates) * ref_hdr.gate_width + ref_hdr.first_gate
ref = np.array([ray[4][b'REF'][1] for ray in f.sweeps[sweep]])

rho_hdr = f.sweeps[sweep][0][4][b'RHO'][0]
rho_range = (np.arange(rho_hdr.num_gates + 1) - 0.5) * rho_hdr.gate_width + rho_hdr.first_gate
rho = np.array([ray[4][b'RHO'][1] for ray in f.sweeps[sweep]])
```

```
fig, axes = plt.subplots(1, 2, figsize=(15, 8))
for var_data, var_range, ax in zip((ref, rho), (ref_range, rho_range), axes):
    # Turn into an array, then mask
    data = ma.array(var_data)
    data[np.isnan(data)] = ma.masked

    # Convert az,range to x,y
    xlocs = var_range * np.sin(np.deg2rad(az[:, np.newaxis]))
    ylocs = var_range * np.cos(np.deg2rad(az[:, np.newaxis]))

    # Plot the data
    cmap = ctables.registry.get_colortable('viridis')
    ax.pcolormesh(xlocs, ylocs, data, cmap=cmap)
    ax.set_aspect('equal', 'datalim')
    ax.set_xlim(-40, 20)
    ax.set_ylim(-30, 30)

plt.show()
```

```
/Users/jleeman/anaconda/lib/python2.7/site-packages/matplotlib/collections.py:590: FutureWarning: ele
  if self._edgecolors == str('face'):
```



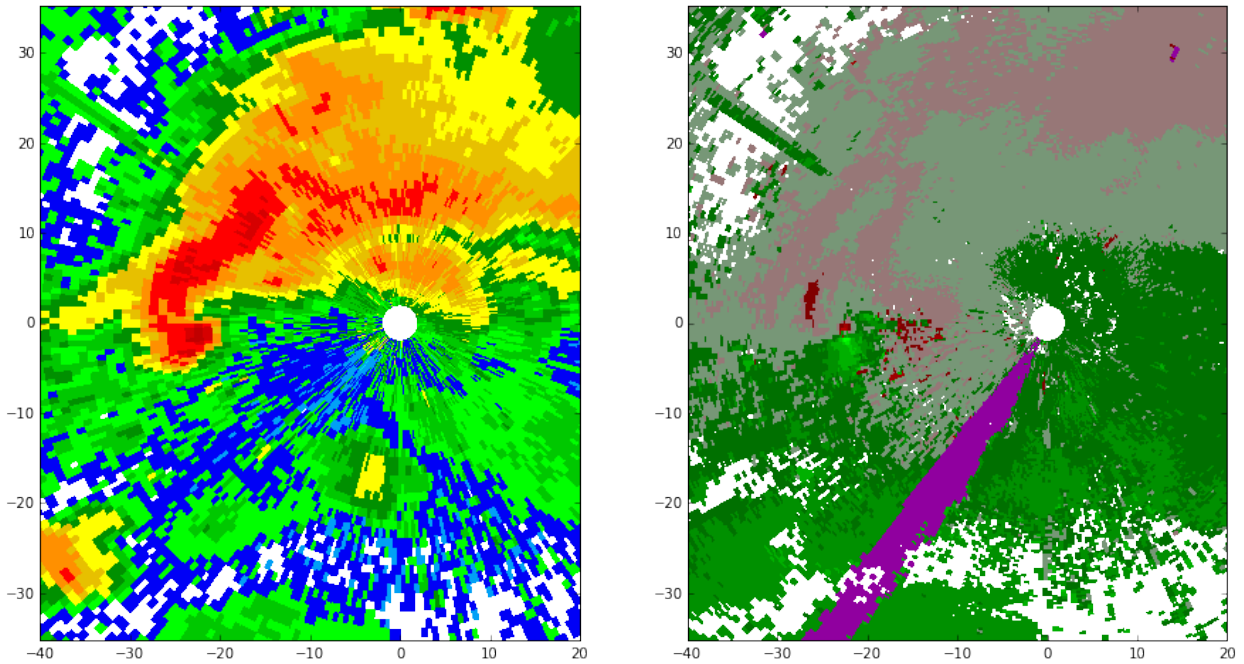
1.3.8 NEXRAD Level 3 File

Notebook

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import ma

from metpy.cbook import get_test_data
from metpy.io.nexrad import Level3File
from metpy.plots import ctables
```

```
%matplotlib inline
```

```
fig, axes = plt.subplots(1, 2, figsize=(15, 8))
for v, ctable, ax in zip(('N0Q', 'N0U'), ('NWSReflectivity', 'NWSVelocity'), axes):
    # Open the file
    name = get_test_data('nids/KOUN_SDUS54_%sTLX_201305202016' % v, as_file_obj=False)
    f = Level3File(name)

    # Pull the data out of the file object
    datadict = f.sym_block[0][0]

    # Turn into an array, then mask
    data = ma.array(datadict['data'])
    data[data==0] = ma.masked

    # Grab azimuths and calculate a range based on number of gates
    az = np.array(datadict['start_az'] + [datadict['end_az'][-1]])
    rng = np.linspace(0, f.max_range, data.shape[-1] + 1)

    # Convert az,range to x,y
    xlocs = rng * np.sin(np.deg2rad(az[:, np.newaxis]))
    ylocs = rng * np.cos(np.deg2rad(az[:, np.newaxis]))

    # Plot the data
    norm, cmap = ctables.registry.get_with_steps(ctable, 16, 16)
    ax.pcolormesh(xlocs, ylocs, data, norm=norm, cmap=cmap)
    ax.set_aspect('equal', 'datalim')
    ax.set_xlim(-40, 20)
    ax.set_ylim(-30, 30)

plt.show()
```



1.3.9 Natural Neighbor Verification

Notebook

Find natural neighbors visual test

A triangle is a natural neighbor for a point if the circumscribed circle of the triangle contains that point. It is important that we correctly grab the correct triangles for each point before proceeding with the interpolation.

Algorithmically:

1. We place all of the grid points in a KDTree. These provide worst-case O(n) time complexity for spatial searches.
2. We generate a Delaunay Triangulation using the locations of the provided observations.
3. For each triangle, we calculate its circumcenter and circumradius. Using KDTree, we then assign each grid a triangle that has a circumcenter within a circumradius of the grid's location.
4. The resulting dictionary uses the grid index as a key and a set of natural neighbor triangles in the form of triangle codes from the Delaunay triangulation. This dictionary is then iterated through to calculate interpolation values.
5. We then traverse the ordered natural neighbor edge vertices for a particular grid cell in groups of 3 (n - 1, n, n + 1), and perform calculations to generate proportional polygon areas.

Circumcenter of (n - 1), n, grid_location

Circumcenter of (n + 1), n, grid_location

Determine what existing circumcenters (ie, Delaunay circumcenters) are associated with vertex n, and add those as polygon vertices. Calculate the area of this polygon.

6. Increment the current edges to be checked, i.e.:

n - 1 = n, n = n + 1, n + 1 = n + 2

7. Repeat steps 5 & 6 until all of the edge combinations of 3 have been visited.
8. Repeat steps 4 through 7 for each grid cell.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (15, 10)

from metpy.gridding.triangles import *
from metpy.gridding.interpolation import nn_point

from scipy.spatial import cKDTree, Delaunay, delaunay_plot_2d, Voronoi, voronoi_plot_2d
```

For a test case, we generate 10 random points and observations, where the observation values are just the x coordinate value times the y coordinate value divided by 1000.

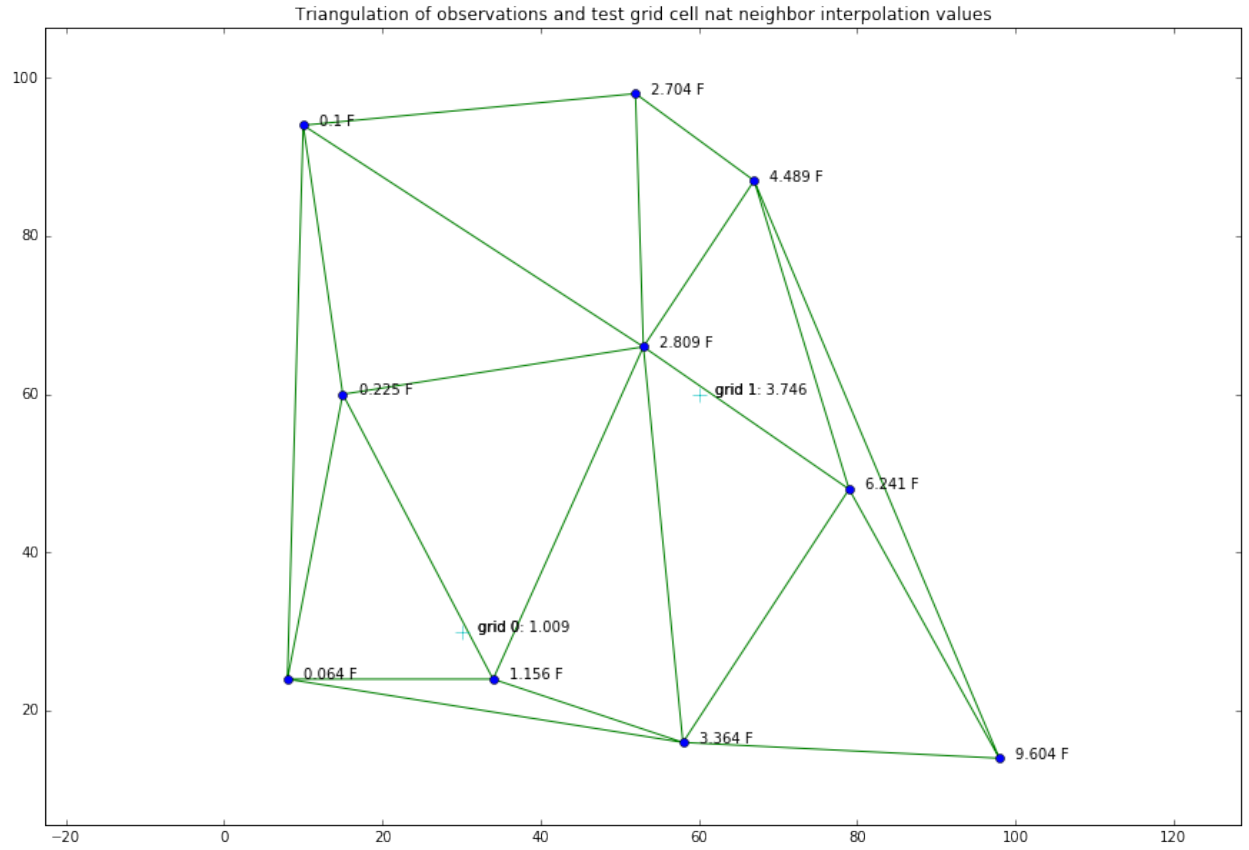
We then create two test points (grid 0 & grid 1) at which we want to estimate a value using natural neighbor interpolation.

The locations of these observations are then used to generate a Delaunay triangulation.

```
np.random.seed(100)

pts = np.random.randint(0, 100, (10, 2))
xp = pts[:, 0]
yp = pts[:, 1]
z = (pts[:, 0] * pts[:, 0]) / 1000
```

```
tri = Delaunay(pts)


delaunay_plot_2d(tri)

for i in range(len(z)):
    plt.annotate(str(z[i]) + " F", xy=(pts[i, 0]+2, pts[i, 1]))

sim_gridx = [30, 60]
sim_gridy = [30, 60]

plt.plot(sim_gridx, sim_gridy, "+", markersize=10)

plt.annotate("grid 0", xy = (sim_gridx[0]+2, sim_gridy[0]))
plt.annotate("grid 1", xy = (sim_gridx[1]+2, sim_gridy[1]))

plt.axes().set_aspect('equal', 'datalim')
plt.title("Triangulation of observations and test grid cell nat neighbor interpolation values")

members, tri_info = find_natural_neighbors(tri, list(zip(sim_gridx, sim_gridy)))

val = nn_point(xp, yp, z, [sim_gridx[0], sim_gridy[0]], tri, members[0], tri_info)
plt.annotate("grid 0: " + ("%.3f" % val), xy = (sim_gridx[0]+2, sim_gridy[0]))

val = nn_point(xp, yp, z, [sim_gridx[1], sim_gridy[1]], tri, members[1], tri_info)
plt.annotate("grid 1: " + ("%.3f" % val), xy = (sim_gridx[1]+2, sim_gridy[1]))
```

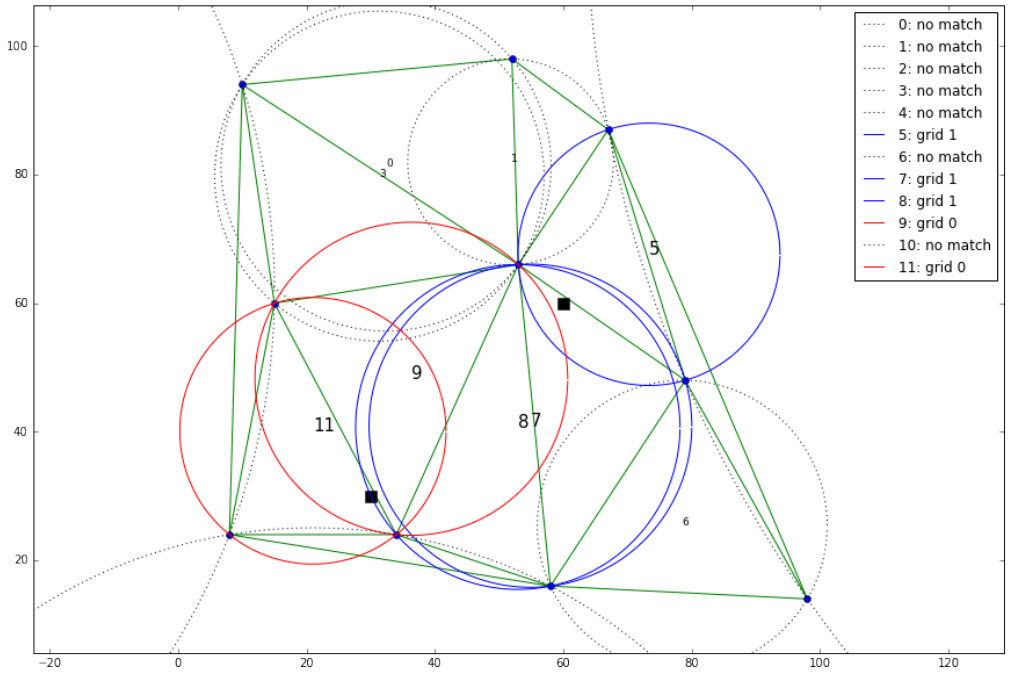
```
<matplotlib.text.Annotation at 0x256e74116d8>
```



Using the circumcenter and circumcircle radius information from metpy.mapping.triangle.find_natural_neighbors, we can visualy examine the results to see if they are correct.

```
from metpy.gridding import triangles
from metpy.gridding.interpolation import nn_point

def draw_circle(x, y, r, m, label):

    nx = x + r * np.cos(np.deg2rad(list(range(360))))
    ny = y + r * np.sin(np.deg2rad(list(range(360))))

    plt.plot(nx, ny, m, label=label)

members, tri_info = triangles.find_natural_neighbors(tri, list(zip(sim_gridx, sim_gridy)))

delaunay_plot_2d(tri)


plt.plot(sim_gridx, sim_gridy, "ks", markersize=10)

for i, info in tri_info.items():

    x_t = info['cc'][0]
    y_t = info['cc'][1]

    if i in members[1] and i in members[0]:

        draw_circle(x_t, y_t, info['r'], 'm-', str(i) + ": grid 1 & 2")

        plt.annotate(str(i), xy=(x_t, y_t), fontsize=15)

    elif i in members[0]:

        draw_circle(x_t, y_t, info['r'], 'r-', str(i) + ": grid 0")

        plt.annotate(str(i), xy=(x_t, y_t), fontsize=15)

    elif i in members[1]:

        draw_circle(x_t, y_t, info['r'], 'b-', str(i) + ": grid 1")

        plt.annotate(str(i), xy=(x_t, y_t), fontsize=15)

    else:

        draw_circle(x_t, y_t, info['r'], 'k:', str(i) + ": no match")

        plt.annotate(str(i), xy=(x_t, y_t), fontsize=9)

plt.axes().set_aspect('equal', 'datalim')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x256e7ce64a8>
```



Whaaa.... the circle from triangle 8 looks pretty darn close.. why isn't grid 0 included in that circle?

```
from scipy.spatial.distance import euclidean

x_t, y_t = tri_info[8]['cc']
r = tri_info[8]['r']

print("Distance between grid0 and Triangle 8's circumcenter:", euclidean([x_t,y_t], [sim_gridx[0], si
print("Triangle 8's circumradius:", r)
```

```
Distance between grid0 and Triangle 8's circumcenter: 25.30650398368644
Triangle 8's circumradius: 25.2587678
```

Lets do a manual check of the above interpolation value for grid 0 (southernmost grid)

Grab the circumcenters and radii for natural neighbors

```
cc = np.array([tri_info[m]['cc'] for m in members[0]])
r = np.array([tri_info[m]['r'] for m in members[0]])

print("circumcenters:\n", cc)
print("radii\n", r)
```

```
circumcenters:
 [[ 36.32995951  48.24358974]
 [ 21.          40.15277778]]
radii
 [ 24.35529419  20.73432492]
```

Draw the natural neighbor triangles and their circumcenters. Also plot a voronoi diagram which serves as a comple-

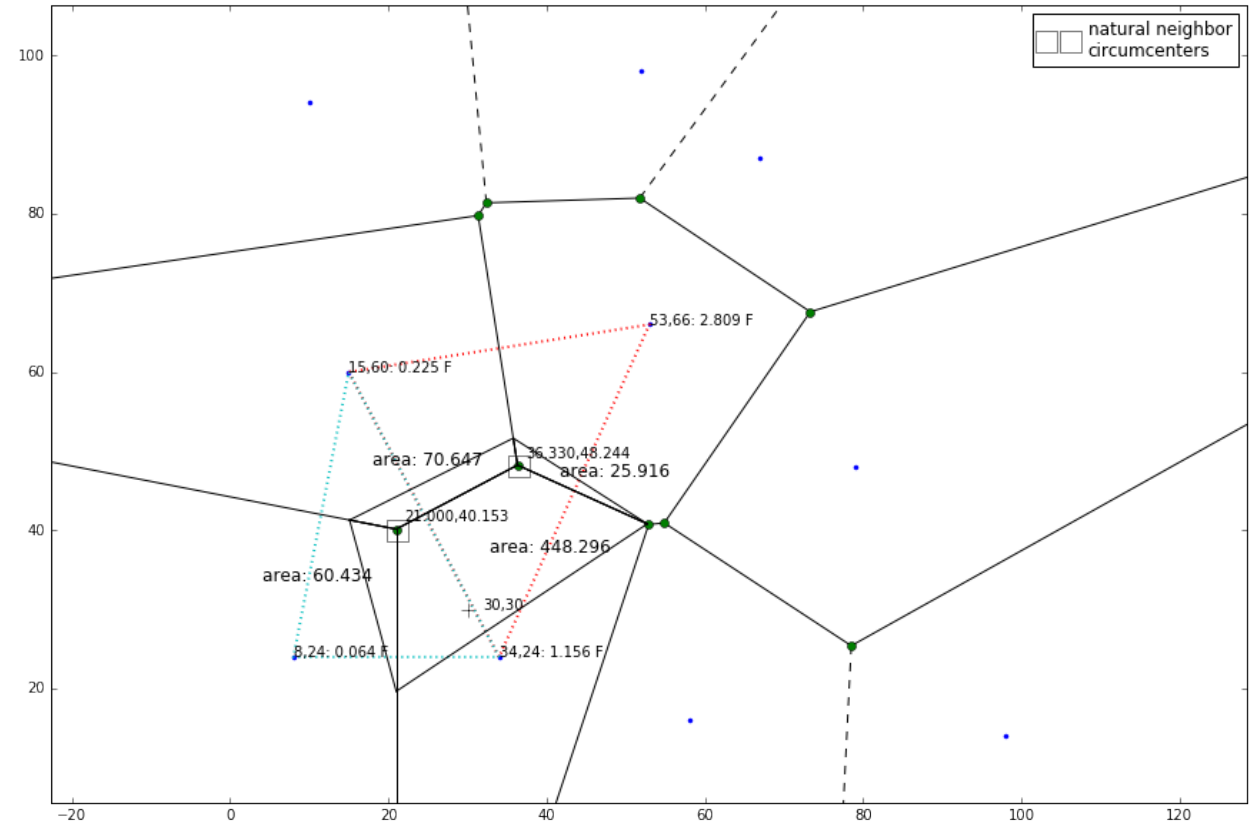
mentary (but not neccesary) spatial data structure that we use here simply to show areal ratios. Notice that the two natural neighbor triangle circumcenters are also vertices in the voronoi plot (green dots), and the observations are in the the polygons (blue dots).

```
from scipy.spatial import Voronoi, voronoi_plot_2d, ConvexHull
from metpy.gridding import polygons

vor = Voronoi(list(zip(xp, yp)))
voronoi_plot_2d(vor)

nn_ind = np.array([0, 5, 7, 8])

z_0 = z[nn_ind]
x_0 = xp[nn_ind]
y_0 = yp[nn_ind]

for i in range(len(nn_ind)):
    lab = str(x_0[i]) + "," + str(y_0[i]) + ": " + ("%.3f" % z_0[i]) + " F"
    plt.annotate(lab, xy=(x_0[i], y_0[i]))

plt.plot(sim_gridx[0], sim_gridy[0], "k+", markersize=10)
plt.annotate(str(sim_gridx[0]) + "," + str(sim_gridy[0]), xy=(sim_gridx[0]+2, sim_gridy[0]))

def plot_triangle(triangle):
    x = [triangle[0,0], triangle[1,0], triangle[2,0], triangle[0,0]]
    y = [triangle[0,1], triangle[1,1], triangle[2,1], triangle[0,1]]

    plt.plot(x, y, ":", linewidth=2)

tris = tri.points[tri.simplices[members[0]]]

plt.plot(cc[:, 0], cc[:, 1], "ks", markersize=15, fillstyle='none', label="natural neighbor\ncircumce

for i in range(len(cc)):

    lab = ("%.3f" % cc[i, 0]) + "," + ("%.3f" % cc[i, 1])
    plt.annotate(lab, xy=(cc[i, 0]+1, cc[i, 1]+1))

for t in tris:
    plot_triangle(t)

plt.legend()
plt.axes().set_aspect('equal', 'datalim')

polygon = list()

polygon.append(cc[0])
#polygon.append(cc[1])

cc1 = circumcenter([53, 66], [15, 60], [30, 30])
cc2 = circumcenter([34, 24], [53, 66], [30, 30])

polygon.append(cc1)
polygon.append(cc2)


pts = np.array([polygon[i] for i in ConvexHull(polygon).vertices])
A = polygons.area(pts)
```

```
for i in range(len(pts)):

    plt.plot([pts[i][0], pts[(i+1)%len(pts)][0]], [pts[i][1], pts[(i+1)%len(pts)][1]], 'k-')

avex = np.mean(pts[:,0])
avey = np.mean(pts[:,1])
plt.annotate("area: " + ("%.3f" % A), xy=(avex, avey), fontsize=12)

polygon = list()

polygon.append(cc[0])
polygon.append(cc[1])

cc1 = circumcenter([53, 66], [15, 60], [30, 30])
cc2 = circumcenter([15, 60], [8, 24], [30, 30])

polygon.append(cc1)
polygon.append(cc2)


pts = np.array([polygon[i] for i in ConvexHull(polygon).vertices])
A = polygons.area(pts)

for i in range(len(pts)):

    plt.plot([pts[i][0], pts[(i+1)%len(pts)][0]], [pts[i][1], pts[(i+1)%len(pts)][1]], 'k-')

avex = np.mean(pts[:,0])
avey = np.mean(pts[:,1])
plt.annotate("area: " + ("%.3f" % A), xy=(avex-9, avey+3), fontsize=12)

polygon = list()

#polygon.append(cc[0])
polygon.append(cc[1])

cc1 = circumcenter([8, 24], [34, 24], [30, 30])
cc2 = circumcenter([15, 60], [8, 24], [30, 30])

polygon.append(cc1)
polygon.append(cc2)


pts = np.array([polygon[i] for i in ConvexHull(polygon).vertices])
A = polygons.area(pts)

for i in range(len(pts)):

    plt.plot([pts[i][0], pts[(i+1)%len(pts)][0]], [pts[i][1], pts[(i+1)%len(pts)][1]], 'k-')

avex = np.mean(pts[:,0])
avey = np.mean(pts[:,1])
plt.annotate("area: " + ("%.3f" % A), xy=(avex-15, avey), fontsize=12)


polygon = list()

polygon.append(cc[0])
```

```
polygon.append(cc[1])

cc1 = circumcenter([8, 24], [34, 24], [30, 30])
cc2 = circumcenter([34, 24], [53, 66], [30, 30])

polygon.append(cc1)
polygon.append(cc2)


pts = np.array([polygon[i] for i in ConvexHull(polygon).vertices])
A = polygons.area(pts)

for i in range(len(pts)):

    plt.plot([pts[i][0], pts[(i+1)%len(pts)][0]], [pts[i][1], pts[(i+1)%len(pts)][1]], 'k-')

avex = np.mean(pts[:,0])
avey = np.mean(pts[:,1])
plt.annotate("area: " + ("%.3f" % A), xy=(avex, avey), fontsize=12)
```

```
<matplotlib.text.Annotation at 0x256e7ddbd68>
```



Put all of the generated polygon areas and their affiliated values in arrays.

Calculate the total area of all of the generated polygons.

```
areas = np.array([60.434, 448.296, 25.916, 70.647])
values = np.array([0.064, 1.156, 2.809, 0.225])

total_area = np.sum(areas)
```

```
total_area
```

```
605.29300000000012
```

For each polygon area, calculate its percent of total area.

```
proportions = areas / np.sum(areas)

proportions
```

```
array([ 0.09984256,  0.74062644,  0.04281563,  0.11671538])
```

Multiply the percent of total area by the respective values.

```
contributions = proportions * values

contributions
```

```
array([ 0.00638992,  0.85616417,  0.1202691 ,  0.02626096])
```

The sum of this array is the interpolation value!

```
interpolation_value = np.sum(contributions)
function_output = nn_point(xp, yp, z, [sim_gridx[0], sim_gridy[0]], tri, members[0], tri_info)

interpolation_value, function_output
```

```
(1.0090841476772403, 1.0090842444256041)
```

The values are slightly different due to truncating the area values in the above visual example to the 3rd decimal place.

1.3.10 Point Interpolation

Notebook

Comparing different point interpolation approaches.

```
import numpy as np
import sys
import matplotlib.pyplot as plt
import cartopy
import cartopy.crs as ccrs

from metpy.gridding.gridding_functions import (interpolate, remove_nan_observations,
                                               remove_repeat_coordinates)
from metpy.cbook import get_test_data

from matplotlib.colors import BoundaryNorm

%matplotlib inline
plt.rcParams['figure.figsize'] = (15, 10)
```

```
def make_string_list(arr):
    return [s.decode('ascii') for s in arr]
```

```
def station_test_data(variable_names, proj_from=None, proj_to=None):

    f = get_test_data('station_data.txt')

    all_data = np.loadtxt(f, skiprows=1, delimiter=',',
                          usecols=(1, 2, 3, 4, 5, 6, 7, 17, 18, 19),
                          dtype=np.dtype([('stid', '3S'), ('lat', 'f'), ('lon', 'f'),
                                          ('slp', 'f'), ('air_temperature', 'f'),
                                          ('cloud_fraction', 'f'), ('dewpoint', 'f'),
                                          ('weather', '16S'),
                                          ('wind_dir', 'f'), ('wind_speed', 'f')]))

    all_stids = make_string_list(all_data['stid'])

    data = np.concatenate([all_data[all_stids.index(site)].reshape(1, ) for site in all_stids])

    value = data[variable_names]
    lon = data['lon']
    lat = data['lat']

    # lon = lon[~np.isnan(value)]
    # lat = lat[~np.isnan(value)]
    # value = value[~np.isnan(value)]

    if proj_from is not None and proj_to is not None:

        try:

            proj_points = proj_to.transform_points(proj_from, lon, lat)
            return proj_points[:, 0], proj_points[:, 1], value

        except Exception as e:

            print(e)
            return None

    return lon, lat, value
```

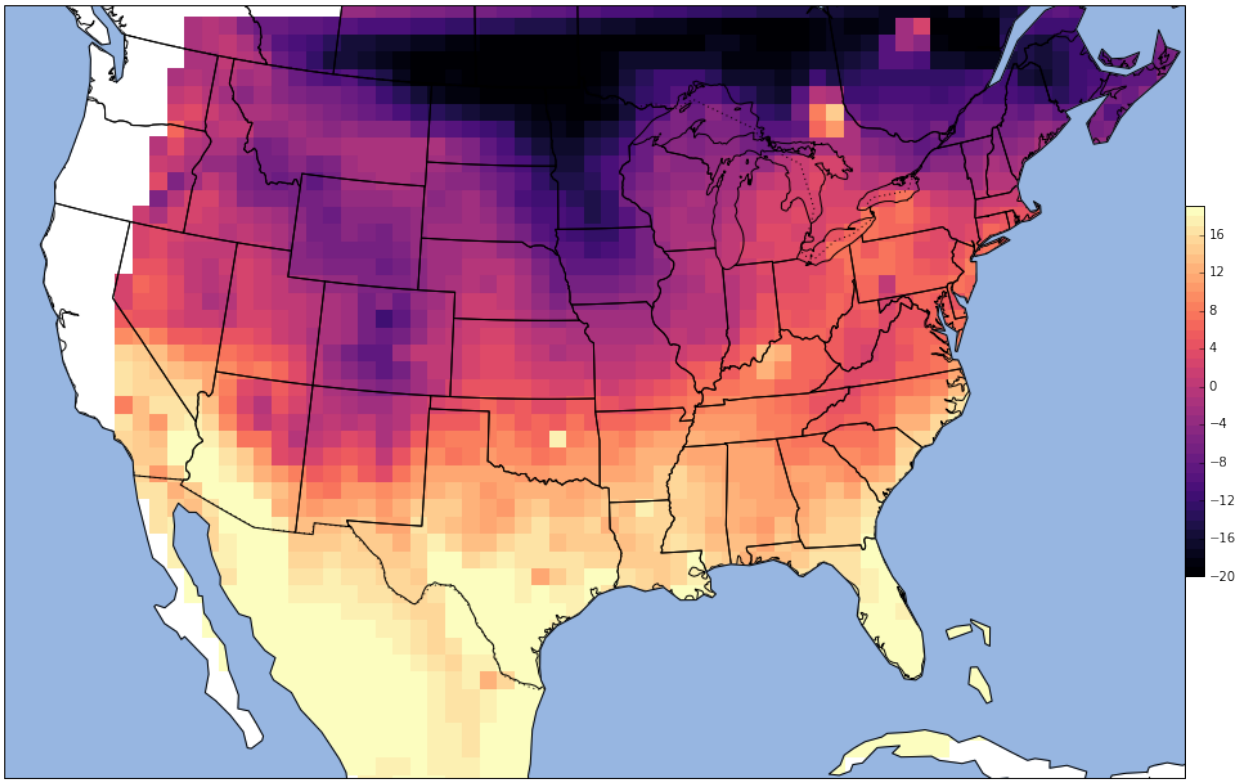
```
from_proj = ccrs.Geodetic()
to_proj = ccrs.AlbersEqualArea(central_longitude=-97.0000, central_latitude=38.0000)

levels = list(range(-20, 20, 1))
cmap = plt.get_cmap('magma')
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

x, y, temp = station_test_data("air_temperature", from_proj, to_proj)

x, y, temp = remove_nan_observations(x, y, temp)
x, y, temp = remove_repeat_coordinates(x, y, temp)
```

Scipy.interpolate linear

```
gx, gy, img = interpolate(x, y, temp, interp_type='linear', hres=75000)
img = np.ma.masked_where(np.isnan(img), img)
view = plt.axes([0,0,1,1], projection=to_proj)
view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
```

```
                                    category='cultural',
                                    name='admin_1_states_provinces_lakes',
                                    scale='50m',
                                    facecolor='none'))
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')

mmb = view.pcolormesh(gx, gy, img, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0, boundaries=levels)
```

```
<matplotlib.colorbar.Colorbar at 0x1109f4ef0>
```



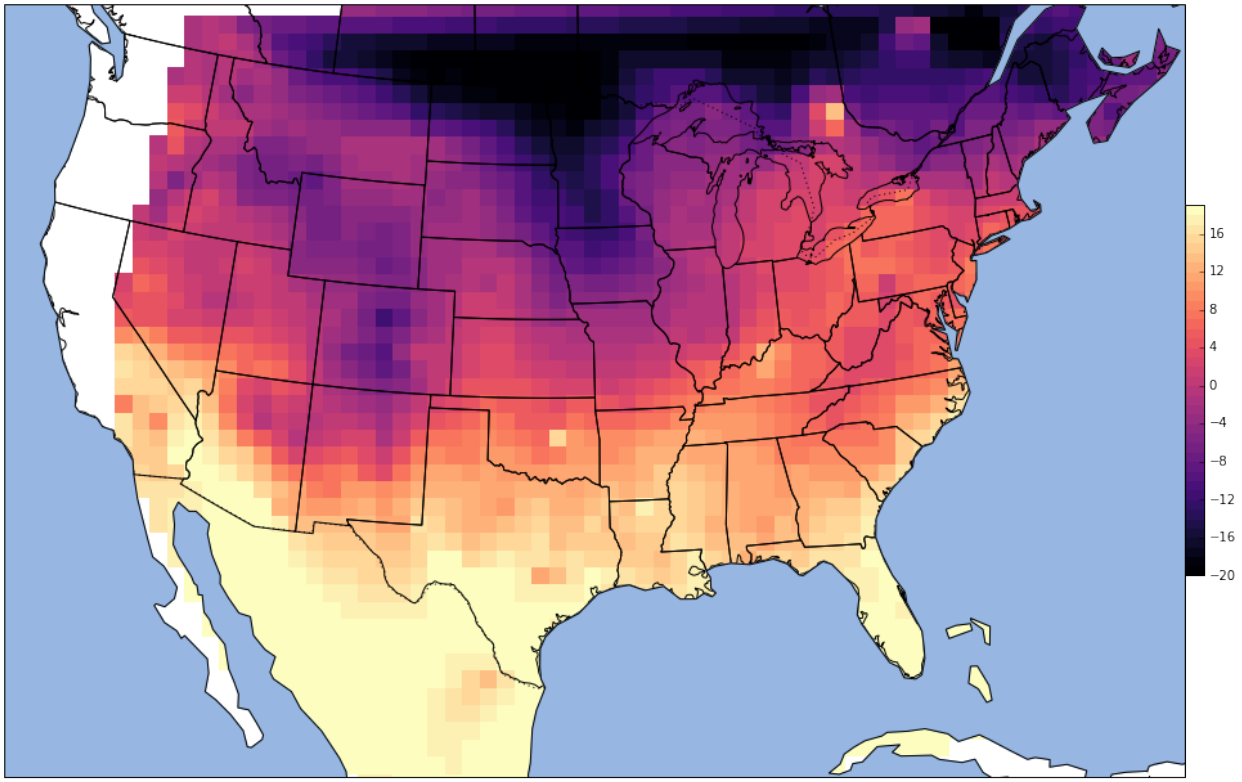
Natural neighbor interpolation (MetPy implementation)

https://github.com/Unidata/MetPy/files/138653/cwp-657.pdf

```
gx, gy, img = interpolate(x, y, temp, interp_type='natural_neighbor', hres=75000)
img = np.ma.masked_where(np.isnan(img), img)
view = plt.axes([0,0,1,1], projection=to_proj)
view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
                                    category='cultural',
                                    name='admin_1_states_provinces_lakes',
                                    scale='50m',
                                    facecolor='none'))
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')
```

```
mmb = view.pcolormesh(gx, gy, img, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0, boundaries=levels)
```

```
<matplotlib.colorbar.Colorbar at 0x1134762b0>
```



Cressman interpolation

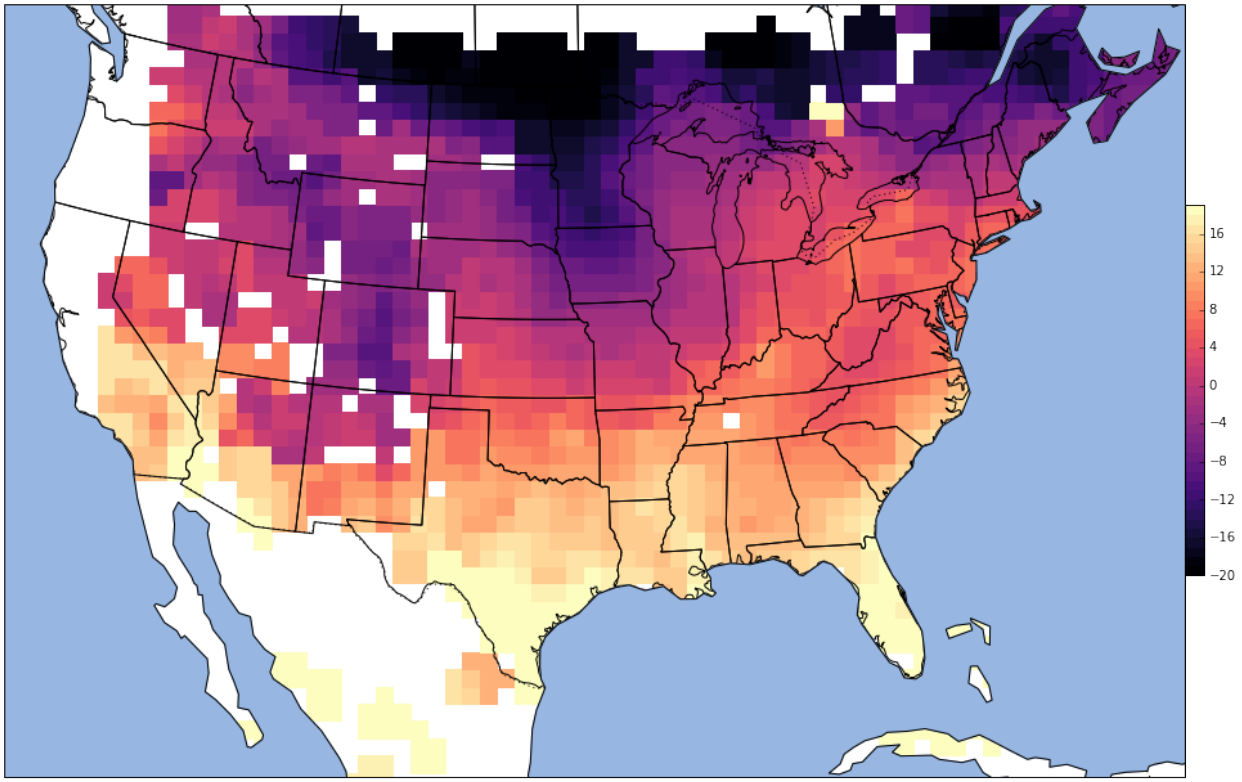
search_radius = 100 km

grid resolution = 25 km

min_neighbors = 1

```
gx, gy, img = interpolate(x, y, temp, interp_type='cressman', minimum_neighbors=1, hres=75000, searcl
img = np.ma.masked_where(np.isnan(img), img)
view = plt.axes([0,0,1,1], projection=to_proj)
view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
                                category='cultural',
                                name='admin_1_states_provinces_lakes',
                                scale='50m',
                                facecolor='none'))
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')

mmb = view.pcolormesh(gx, gy, img, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0, boundaries=levels)
```

```
<matplotlib.colorbar.Colorbar at 0x113456b70>
```



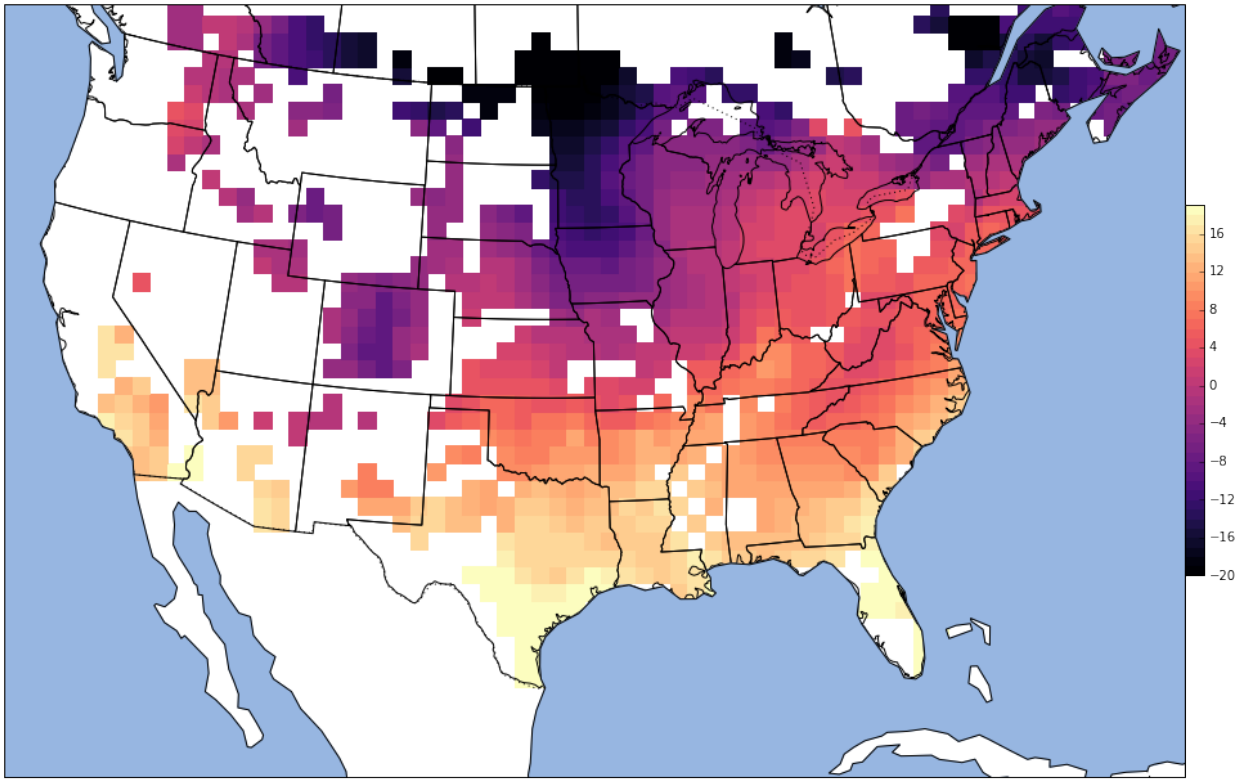
Barnes Interpolation

search_radius = 100km

min_neighbors = 3

```
gx, gy, img1 = interpolate(x, y, temp, interp_type='barnes', hres=75000, search_radius=100000)
img1 = np.ma.masked_where(np.isnan(img1), img1)
view = plt.axes([0,0,1,1], projection=to_proj)
view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
                                category='cultural',
                                name='admin_1_states_provinces_lakes',
                                scale='50m',
                                facecolor='none'))
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')

mmb = view.pcolormesh(gx, gy, img1, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0, boundaries=levels)
```

```
<matplotlib.colorbar.Colorbar at 0x115838ba8>
```

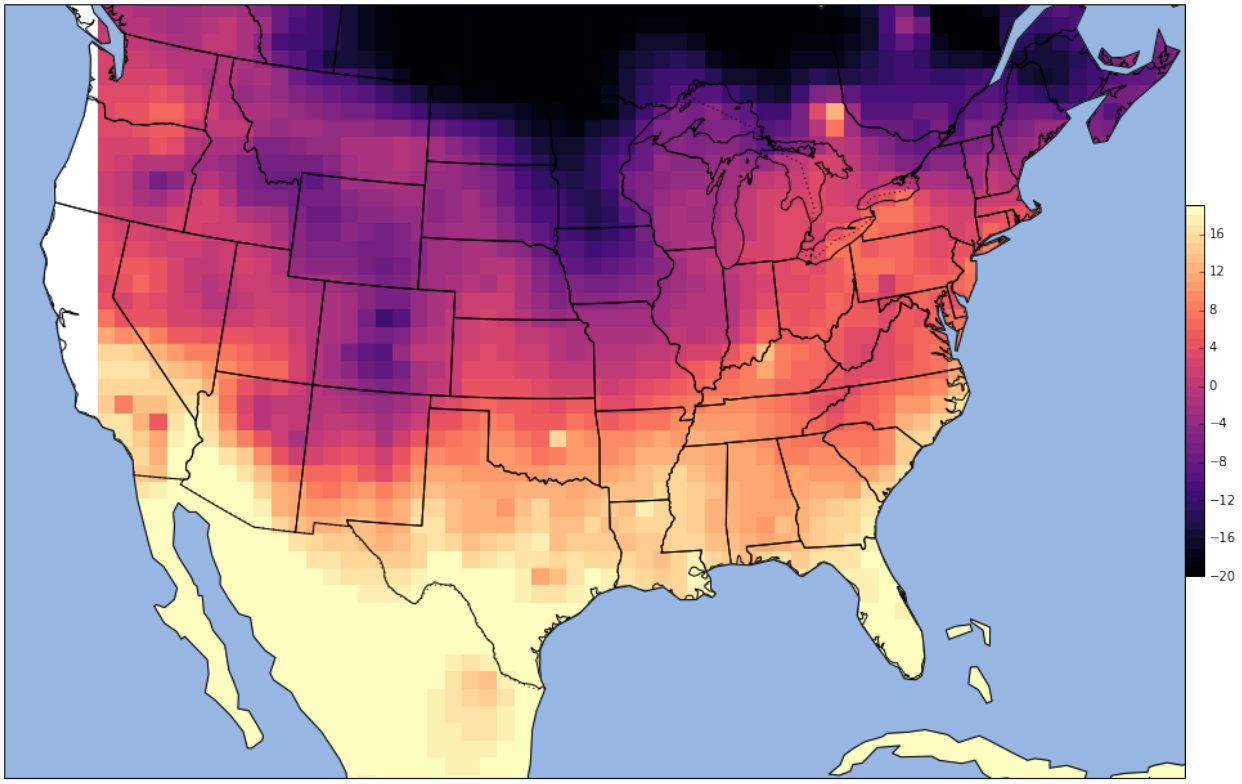


Radial basis functions interpolation

linear

```
gx, gy, img = interpolate(x, y, temp, interp_type='rbf', hres=75000, rbf_func='linear', rbf_smooth=0)
img = np.ma.masked_where(np.isnan(img), img)
view = plt.axes([0,0,1,1], projection=to_proj)
view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
                                category='cultural',
                                name='admin_1_states_provinces_lakes',
                                scale='50m',
                                facecolor='none'))
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')

mmb = view.pcolormesh(gx, gy, img, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0, boundaries=levels)
```

```
<matplotlib.colorbar.Colorbar at 0x115e65940>
```



1.3.11 Simple Sounding

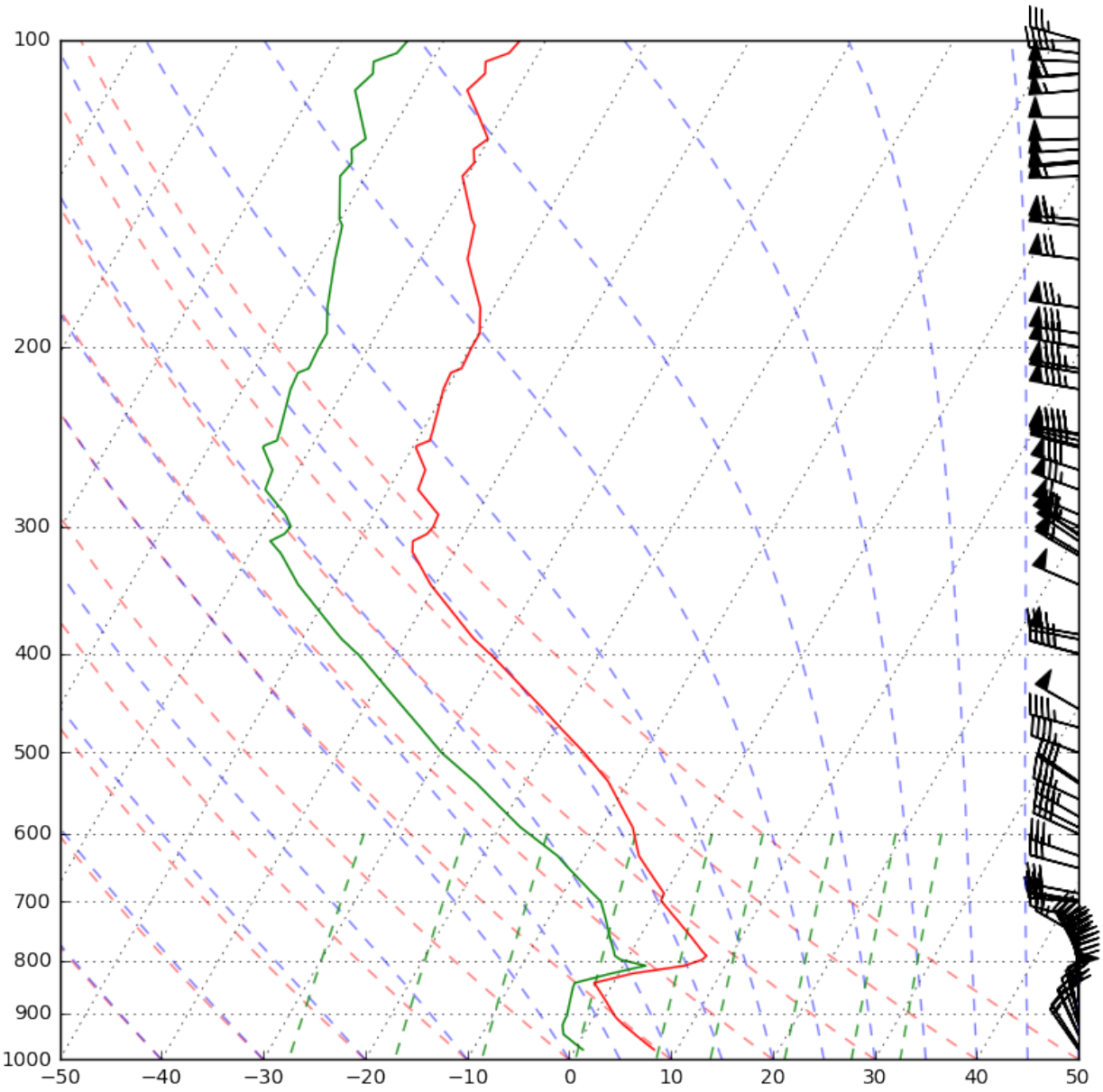
Notebook

```
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np

from metpy.cbook import get_test_data
from metpy.calc import resample_nn_1d
from metpy.io import get_upper_air_data
from metpy.plots import SkewT
from metpy.units import units

%matplotlib inline

# Change default to be better for skew-T
plt.rcParams['figure.figsize'] = (9, 9)
```

```
from metpy.io.upperair import UseSampleData
with UseSampleData():  # Only needed to use our local sample data
    # Download and parse the data
    dataset = get_upper_air_data(datetime(2013, 1, 20, 12), 'OUN')

p = dataset.variables['pressure'][:]
T = dataset.variables['temperature'][:]
Td = dataset.variables['dewpoint'][:]
u = dataset.variables['u_wind'][:]
```

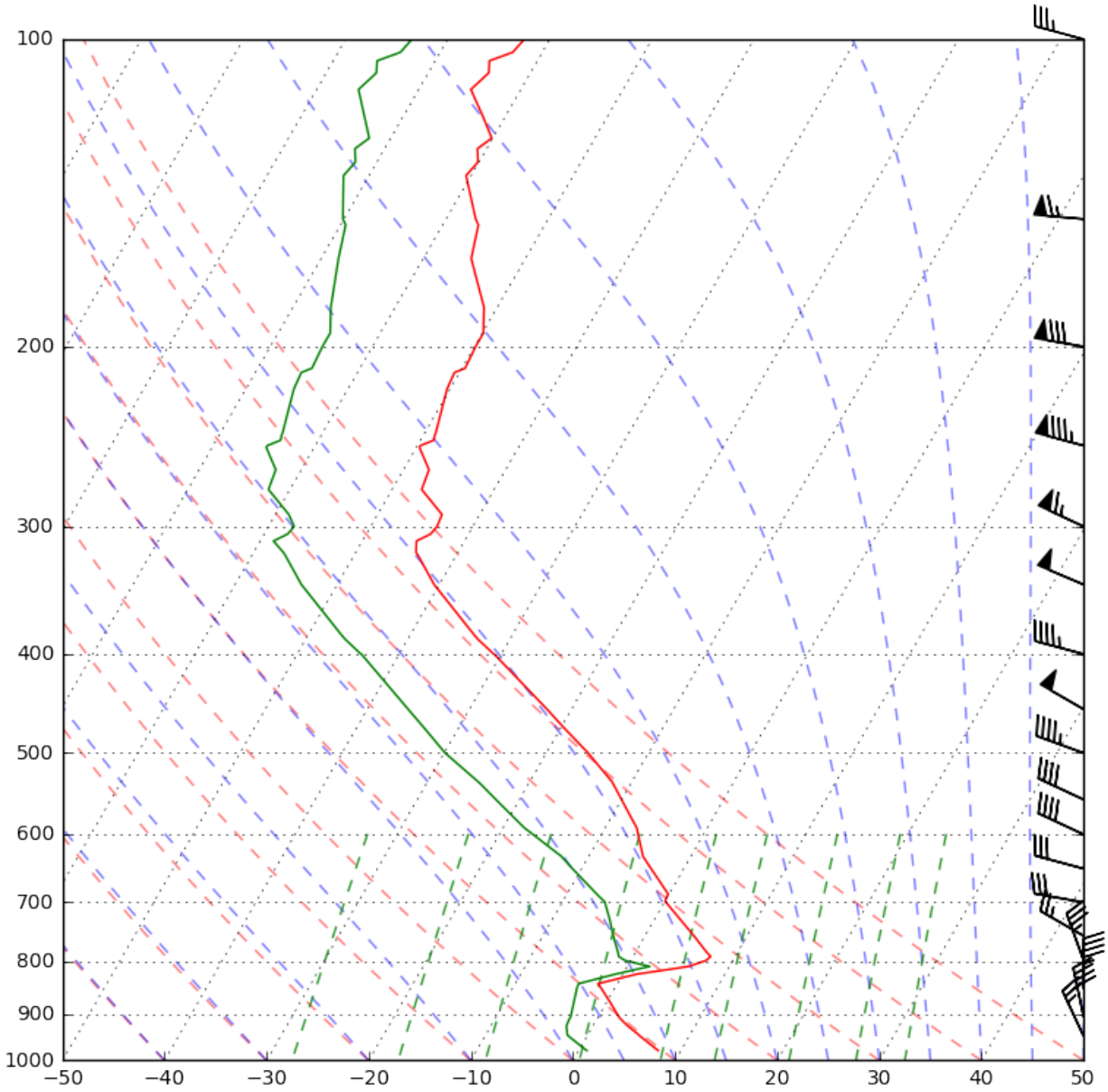
```
v = dataset.variables['v_wind'][:]
```

```
# Create a skewT using matplotlib's default figure size
skew = SkewT()

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
skew.plot(p, T, 'r')
skew.plot(p, Td, 'g')
skew.plot_barbs(p, u, v)

# Add the relevant special lines
skew.plot_dry_adiabats()
skew.plot_moist_adiabats()
skew.plot_mixing_lines()
skew.ax.set_ylim(1000, 100)

# Show the plot
plt.show()
```




```
# Example of defining your own vertical barb spacing
skew = SkewT()

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
skew.plot(p, T, 'r')
skew.plot(p, Td, 'g')

#Set spacing interval
#Example: Every 50 mb from 1000 to 100 mb
my_interval = np.arange(100, 1000, 50) * units('mbar')

#Get indexes of values closest to defined interval
ix = resample_nn_1d(p, my_interval)

#Plot only values nearest to defined interval values
```

```
skew.plot_barbs(p[ix], u[ix], v[ix])

# Add the relevant special lines
skew.plot_dry_adiabats()
skew.plot_moist_adiabats()
skew.plot_mixing_lines()
skew.ax.set_ylim(1000, 100)

# Show the plot
plt.show()
```



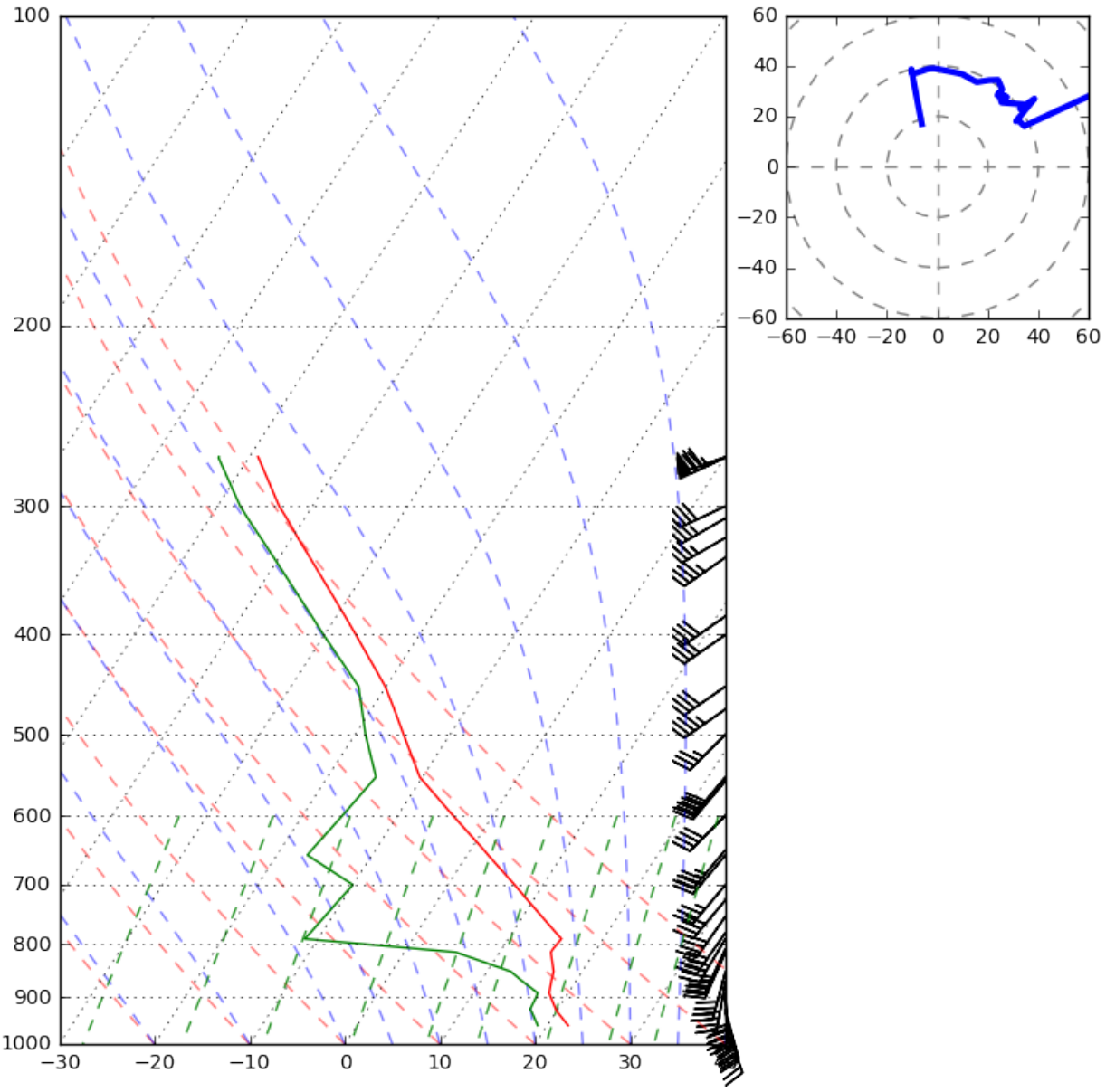
1.3.12 Skew-T Layout

Notebook

```
from datetime import datetime

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from metpy.cbook import get_test_data
from metpy.io import get_upper_air_data
from metpy.plots import SkewT, Hodograph

%matplotlib inline
```

```
from metpy.io.upperair import UseSampleData
with UseSampleData():  # Only needed to use our local sample data
    # Download and parse the data
    dataset = get_upper_air_data(datetime(1999, 5, 4, 0), 'OUN')

p = dataset.variables['pressure'][:]
T = dataset.variables['temperature'][:]
Td = dataset.variables['dewpoint'][:]
u = dataset.variables['u_wind'][:]
v = dataset.variables['v_wind'][:]
```

```
# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(9, 9))

# Grid for plots
gs = gridspec.GridSpec(3, 3)
skew = SkewT(fig, rotation=45, subplot=gs[:, :2])

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
skew.plot(p, T, 'r')
skew.plot(p, Td, 'g')
skew.plot_barbs(p, u, v)
skew.ax.set_ylim(1000, 100)

# Add the relevant special lines
skew.plot_dry_adiabats()
skew.plot_moist_adiabats()
skew.plot_mixing_lines()

# Good bounds for aspect ratio
skew.ax.set_xlim(-30, 40)

# Create a hodograph
ax = fig.add_subplot(gs[0, -1])
h = Hodograph(ax, component_range=60.)
h.add_grid(increment=20)
h.plot(u, v)

# Show the plot
plt.show()
```



1.3.13 Station Plot

Notebook

This example makes a station plot, complete with sky cover and weather symbols. The station plot itself is pretty straightforward, but there is a bit of code to perform the data-wrangling (hopefully that situation will improve in the future). Certainly, if you have existing point data in a format you can work with trivially, the station plot will be simple.

```
import matplotlib.pyplot as plt
import numpy as np

from metpy.calc import get_wind_components
from metpy.cbook import get_test_data
from metpy.plots import StationPlot
```

```
from metpy.plots.wx_symbols import sky_cover, current_weather
from metpy.units import units

%matplotlib inline
```

```
# Utility function for working with the text data we get back in arrays
def make_string_list(arr):
    return [s.decode('ascii') for s in arr]
```

The setup

First read in the data. We use `numpy.loadtxt` to read in the data and use a structured `numpy.dtype` to allow different types for the various columns. This allows us to handle the columns with string data.

```
f = get_test_data('station_data.txt')

all_data = np.loadtxt(f, skiprows=1, delimiter=',',
                      usecols=(1, 2, 3, 4, 5, 6, 7, 17, 18, 19),
                      dtype=np.dtype([('stid', '3S'), ('lat', 'f'), ('lon', 'f'),
                                      ('slp', 'f'), ('air_temperature', 'f'),
                                      ('cloud_fraction', 'f'), ('dewpoint', 'f'),
                                      ('weather', '16S'),
                                      ('wind_dir', 'f'), ('wind_speed', 'f')]))
```

This sample data has *way* too many stations to plot all of them. Instead, we just select a few from around the U.S. and pull those out of the data file.

```
# Get the full list of stations in the data
all_stids = make_string_list(all_data['stid'])

# Pull out these specific stations
whitelist = ['OKC', 'ICT', 'GLD', 'MEM', 'BOS', 'MIA', 'MOB', 'ABQ', 'PHX', 'TTF',
             'ORD', 'BIL', 'BIS', 'CPR', 'LAX', 'ATL', 'MSP', 'SLC', 'DFW', 'NYC', 'PHL',
             'PIT', 'IND', 'OLY', 'SYR', 'LEX', 'CHS', 'TLH', 'HOU', 'GJT', 'LBB', 'LSV',
             'GRB', 'CLT', 'LNK', 'DSM', 'BOI', 'FSD', 'RAP', 'RIC', 'JAN', 'HSV', 'CRW',
             'SAT', 'BUY', '0CO', 'ZPC', 'VIH']

# Loop over all the whitelisted sites, grab the first data, and concatenate them
data = np.concatenate([all_data[all_stids.index(site)].reshape(1,) for site in whitelist])
```

Now that we have the data we want, we need to perform some conversions: - Get a list of strings for the station IDs - Get wind components from speed and direction - Convert cloud fraction values to integer codes [0 - 8] - Map METAR weather codes to WMO codes for weather symbols

```
# Get all of the station IDs as a list of strings
stid = make_string_list(data['stid'])

# Get the wind components, converting from m/s to knots as will be appropriate
# for the station plot
u, v = get_wind_components((data['wind_speed'] * units('m/s')).to('knots'),
                           data['wind_dir'] * units.degree)

# Convert the fraction value into a code of 0-8, which can be used to pull out
# the appropriate symbol
cloud_frac = (8 * data['cloud_fraction']).astype(int)

# Map weather strings to WMO codes, which we can use to convert to symbols
```

```
# Only use the first symbol if there are multiple
wx_text = make_string_list(data['weather'])
wx_codes = {'':0, 'HZ':5, 'BR':10, '-DZ':51, 'DZ':53, '+DZ':55,
            '-RA':61, 'RA':63, '+RA':65, '-SN':71, 'SN':73, '+SN':75}
wx = [wx_codes[s.split()[0] if ' ' in s else s] for s in wx_text]
```

Now all the data wrangling is finished, just need to set up plotting and go

```
# Set up the map projection and set up a cartopy feature for state borders
import cartopy.crs as ccrs
import cartopy.feature as feat
proj = ccrs.LambertConformal(central_longitude=-95, central_latitude=35,
                             standard_parallels=[35])
state_boundaries = feat.NaturalEarthFeature(category='cultural',
                                            name='admin_1_states_provinces_lines',
                                            scale='110m', facecolor='none')
```

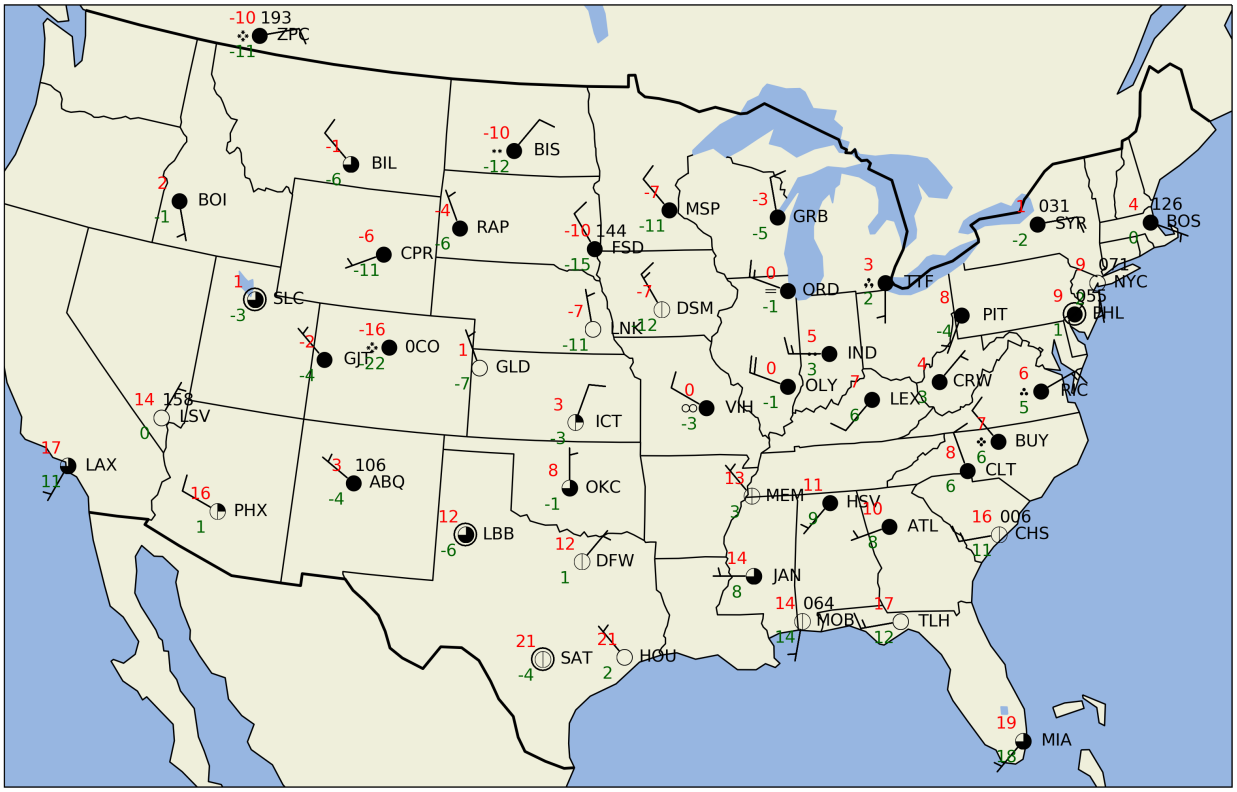
```
# Change the DPI of the resulting figure. Higher DPI drastically improves the
# look of the text rendering
from matplotlib import rcParams
rcParams['savefig.dpi'] = 255
```

The payoff

```
# Create the figure and an axes set to the projection
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(1, 1, 1, projection=proj)

# Add some various map elements to the plot to make it recognizable
ax.add_feature(feat.LAND, zorder=-1)
ax.add_feature(feat.OCEAN, zorder=-1)
ax.add_feature(feat.LAKES, zorder=-1)
ax.coastlines(resolution='110m', zorder=2, color='black')
ax.add_feature(state_boundaries)
ax.add_feature(feat.BORDERS, linewidth='2', edgecolor='black')

# Set plot bounds
ax.set_extent((-118, -73, 23, 50))

#
# Here's the actual station plot
#

# Start the station plot by specifying the axes to draw on, as well as the
# lon/lat of the stations (with transform). We also the fontsize to 12 pt.
stationplot = StationPlot(ax, data['lon'], data['lat'], transform=ccrs.PlateCarree(),
                          fontsize=12)

# Plot the temperature and dew point to the upper and lower left, respectively, of
# the center point. Each one uses a different color.
stationplot.plot_parameter('NW', data['air_temperature'], color='red')
stationplot.plot_parameter('SW', data['dewpoint'], color='darkgreen')

# A more complex example uses a custom formatter to control how the sea-level pressure
# values are plotted. This uses the standard trailing 3-digits of the pressure value
# in tenths of millibars.
stationplot.plot_parameter('NE', data['slp'],
```

```
                        formatter=lambda v: format(10 * v, '.0f')[-3:])

# Plot the cloud cover symbols in the center location. This uses the codes made above and
# uses the `sky_cover` mapper to convert these values to font codes for the
# weather symbol font.
stationplot.plot_symbol('C', cloud_frac, sky_cover)

# Same this time, but plot current weather to the left of center, using the
# `current_weather` mapper to convert symbols to the right glyphs.
stationplot.plot_symbol('W', wx, current_weather)

# Add wind barbs
stationplot.plot_barb(u, v)

# Also plot the actual text of the station id. Instead of cardinal directions,
# plot further out by specifying a location of 2 increments in x and 0 in y.
stationplot.plot_text((2, 0), stid)
```

```
/Users/rmay/miniconda3/envs/metpy3/lib/python3.4/site-packages/matplotlib/artist.py:221: MatplotlibDe
axes property.  A removal date has not been set.
  warnings.warn(_get_axes_msg, mplDeprecation, stacklevel=1)
```



1.3.14 Station Plot with Layout

Notebook

This example makes a station plot, complete with sky cover and weather symbols, using a station plot layout built into MetPy. The station plot itself is straightforward, but there is a bit of code to perform the data-wrangling (hopefully that situation will improve in the future). Certainly, if you have existing point data in a format you can work with

trivially, the station plot will be simple.

The `StationPlotLayout` class is used to standardize the plotting various parameters (i.e. temperature), keeping track of the location, formatting, and even the units for use in the station plot. This makes it easy (if using standardized names) to re-use a given layout of a station plot.

```
import matplotlib.pyplot as plt
import numpy as np

from metpy.calc import get_wind_components
from metpy.cbook import get_test_data
from metpy.plots import StationPlot, StationPlotLayout, simple_layout
from metpy.units import units

%matplotlib inline
```

```
# Utility function for working with the text data we get back in arrays
def make_string_list(arr):
    return [s.decode('ascii') for s in arr]
```

The setup

First read in the data. We use `numpy.loadtxt` to read in the data and use a structured `numpy.dtype` to allow different types for the various columns. This allows us to handle the columns with string data.

```
f = get_test_data('station_data.txt')
all_data = np.loadtxt(f, skiprows=1, delimiter=',',
                      usecols=(1, 2, 3, 4, 5, 6, 7, 17, 18, 19),
                      dtype=np.dtype([('stid', '3S'), ('lat', 'f'), ('lon', 'f'),
                                      ('slp', 'f'), ('air_temperature', 'f'),
                                      ('cloud_fraction', 'f'), ('dewpoint', 'f'),
                                      ('weather', '16S'),
                                      ('wind_dir', 'f'), ('wind_speed', 'f')]))
```

This sample data has *way* too many stations to plot all of them. Instead, we just select a few from around the U.S. and pull those out of the data file.

```
# Get the full list of stations in the data
all_stids = make_string_list(all_data['stid'])

# Pull out these specific stations
whitelist = ['OKC', 'ICT', 'GLD', 'MEM', 'BOS', 'MIA', 'MOB', 'ABQ', 'PHX', 'TTF',
             'ORD', 'BIL', 'BIS', 'CPR', 'LAX', 'ATL', 'MSP', 'SLC', 'DFW', 'NYC', 'PHL',
             'PIT', 'IND', 'OLY', 'SYR', 'LEX', 'CHS', 'TLH', 'HOU', 'GJT', 'LBB', 'LSV',
             'GRB', 'CLT', 'LNK', 'DSM', 'BOI', 'FSD', 'RAP', 'RIC', 'JAN', 'HSV', 'CRW',
             'SAT', 'BUY', '0CO', 'ZPC', 'VIH']

# Loop over all the whitelisted sites, grab the first data, and concatenate them
data_arr = np.concatenate([all_data[all_stids.index(site)].reshape(1,) for site in whitelist])
```

First, look at the names of variables that the layout is expecting:

```
simple_layout.names()
```

```
['air_temperature',
 'eastward_wind',
 'northward_wind',
 'cloud_coverage',
```

```
 'present_weather',
 'air_pressure_at_sea_level',
 'dew_point_temperature']
```

Next grab the simple variables out of the data we have (attaching correct units), and put them into a dictionary that we will hand the plotting function later:

```
# This is our container for the data
data = dict()

# Copy out to stage everything together. In an ideal world, this would happen on
# the data reading side of things, but we're not there yet.
data['longitude'] = data_arr['lon']
data['latitude'] = data_arr['lat']
data['air_temperature'] = data_arr['air_temperature'] * units.degC
data['dew_point_temperature'] = data_arr['dewpoint'] * units.degC
data['air_pressure_at_sea_level'] = data_arr['slp'] * units('mbar')
```

Notice that the names (the keys) in the dictionary are the same as those that the layout is expecting.

Now perform a few conversions: - Get wind components from speed and direction - Convert cloud fraction values to integer codes [0 - 8] - Map METAR weather codes to WMO codes for weather symbols

```
# Get the wind components, converting from m/s to knots as will be appropriate
# for the station plot
u, v = get_wind_components(data_arr['wind_speed'] * units('m/s'),
                           data_arr['wind_dir'] * units.degree)
data['eastward_wind'], data['northward_wind'] = u, v

# Convert the fraction value into a code of 0-8, which can be used to pull out
# the appropriate symbol
data['cloud_coverage'] = (8 * data_arr['cloud_fraction']).astype(int)

# Map weather strings to WMO codes, which we can use to convert to symbols
# Only use the first symbol if there are multiple
wx_text = make_string_list(data_arr['weather'])
wx_codes = {'':0, 'HZ':5, 'BR':10, '-DZ':51, 'DZ':53, '+DZ':55,
            '-RA':61, 'RA':63, '+RA':65, '-SN':71, 'SN':73, '+SN':75}
data['present_weather'] = [wx_codes[s.split()[0] if ' ' in s else s] for s in wx_text]
```

All the data wrangling is finished, just need to set up plotting and go:

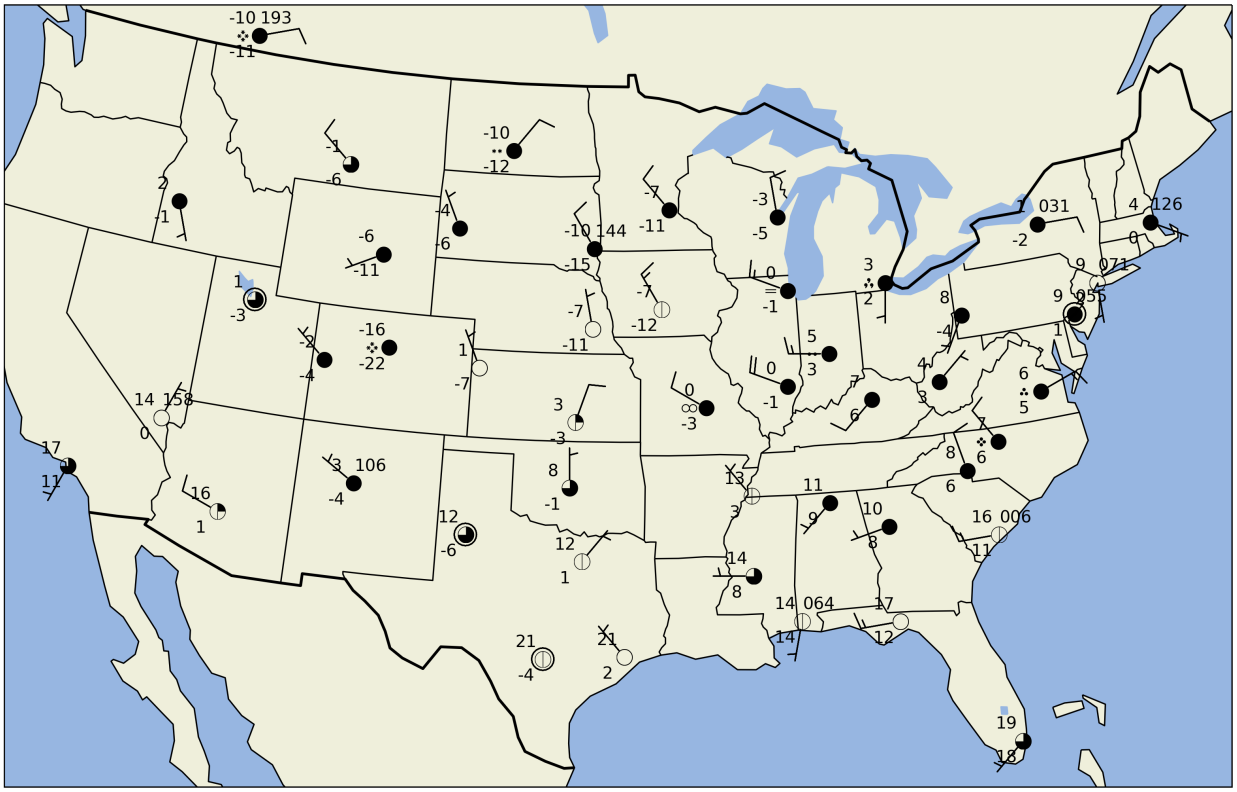
```
# Set up the map projection and set up a cartopy feature for state borders
import cartopy.crs as ccrs
import cartopy.feature as feat
proj = ccrs.LambertConformal(central_longitude=-95, central_latitude=35,
                             standard_parallels=[35])
state_boundaries = feat.NaturalEarthFeature(category='cultural',
                                            name='admin_1_states_provinces_lines',
                                            scale='110m', facecolor='none')
```

```
# Change the DPI of the resulting figure. Higher DPI drastically improves the
# look of the text rendering
from matplotlib import rcParams
rcParams['savefig.dpi'] = 255
```

The payoff

```
# Create the figure and an axes set to the projection
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(1, 1, 1, projection=proj)

# Add some various map elements to the plot to make it recognizable
ax.add_feature(feat.LAND, zorder=-1)
ax.add_feature(feat.OCEAN, zorder=-1)
ax.add_feature(feat.LAKES, zorder=-1)
ax.coastlines(resolution='110m', zorder=2, color='black')
ax.add_feature(state_boundaries)
ax.add_feature(feat.BORDERS, linewidth='2', edgecolor='black')

# Set plot bounds
ax.set_extent((-118, -73, 23, 50))

#
# Here's the actual station plot
#

# Start the station plot by specifying the axes to draw on, as well as the
# lon/lat of the stations (with transform). We also the fontsize to 12 pt.
stationplot = StationPlot(ax, data['longitude'], data['latitude'],
                          transform=ccrs.PlateCarree(), fontsize=12)

# The layout knows where everything should go, and things are standardized using
# the names of variables. So the layout pulls arrays out of `data` and plots them
# using `stationplot`.
simple_layout.plot(stationplot, data)
```

```
/Users/rmay/miniconda3/envs/metpy3/lib/python3.4/site-packages/matplotlib/artist.py:221: MatplotlibDe
axes property.  A removal date has not been set.
  warnings.warn(_get_axes_msg, mplDeprecation, stacklevel=1)
```



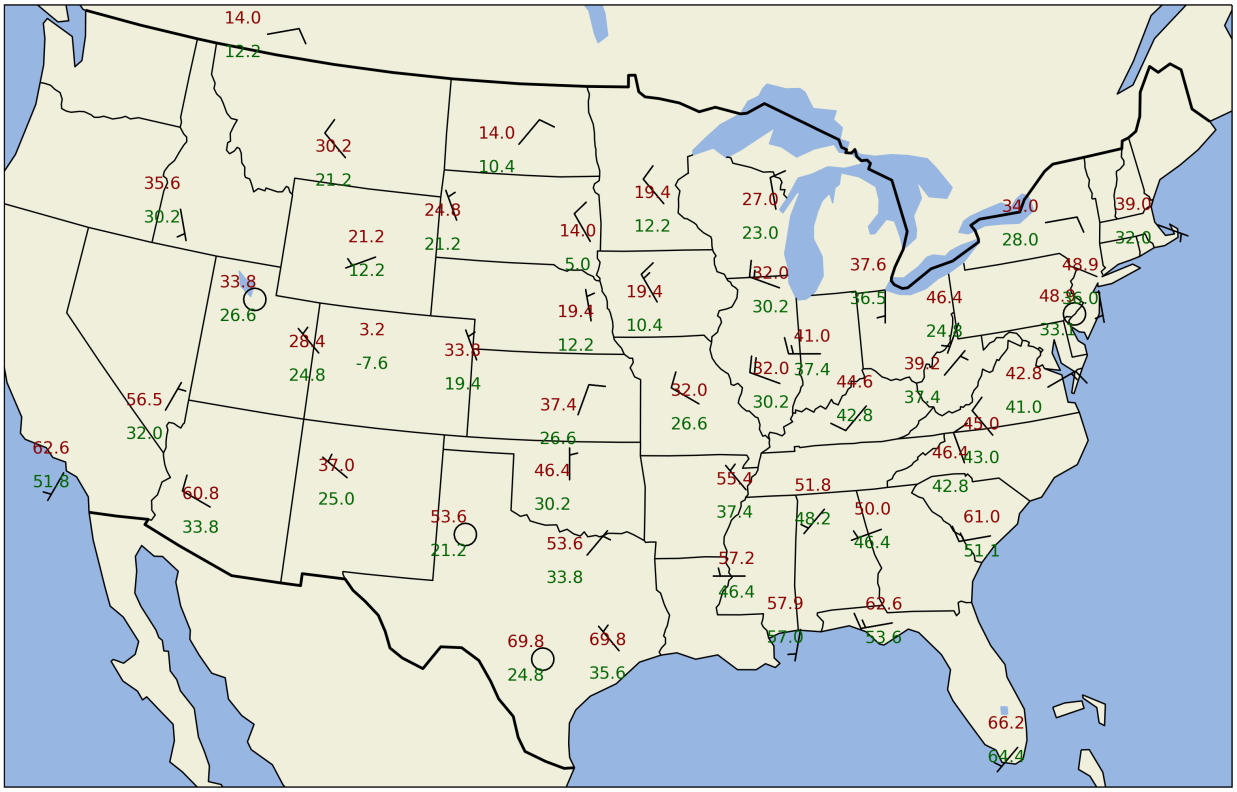
or instead, a custom layout can be used:

```
# Just winds, temps, and dewpoint, with colors. Dewpoint and temp will be plotted
# out to Farenheit tenths. Extra data will be ignored
custom_layout = StationPlotLayout()
custom_layout.add_barb('eastward_wind', 'northward_wind', units='knots')
custom_layout.add_value('NW', 'air_temperature', fmt='.1f', units='degF', color='darkred')
custom_layout.add_value('SW', 'dew_point_temperature', fmt='.1f', units='degF',
                        color='darkgreen')

# Also, we'll add a field that we don't have in our dataset. This will be ignored
custom_layout.add_value('E', 'precipitation', fmt='0.2f', units='inch', color='blue')
```

```
# Create the figure and an axes set to the projection
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(1, 1, 1, projection=proj)

# Add some various map elements to the plot to make it recognizable
ax.add_feature(feat.LAND, zorder=-1)
ax.add_feature(feat.OCEAN, zorder=-1)
ax.add_feature(feat.LAKES, zorder=-1)
ax.coastlines(resolution='110m', zorder=2, color='black')
ax.add_feature(state_boundaries)
ax.add_feature(feat.BORDERS, linewidth='2', edgecolor='black')

# Set plot bounds
ax.set_extent((-118, -73, 23, 50))

#
# Here's the actual station plot
#
```

```
# Start the station plot by specifying the axes to draw on, as well as the
# lon/lat of the stations (with transform). We also the fontsize to 12 pt.
stationplot = StationPlot(ax, data['longitude'], data['latitude'],
                          transform=ccrs.PlateCarree(), fontsize=12)

# The layout knows where everything should go, and things are standardized using
# the names of variables. So the layout pulls arrays out of `data` and plots them
# using `stationplot`.
custom_layout.plot(stationplot, data)
```

```
/Users/rmay/miniconda3/envs/metpy3/lib/python3.4/site-packages/matplotlib/artist.py:221: MatplotlibDe
axes property.  A removal date has not been set.
  warnings.warn(_get_axes_msg, mplDeprecation, stacklevel=1)
```



1.3.15 Wind SLP Interpolation

Notebook

Interpolating wind and pressure to grids using station observations

```
import warnings

import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

```
import cartopy
import cartopy.crs as ccrs
import cartopy.feature as feat

from metpy.gridding.gridding_functions import (interpolate, remove_nan_observations,
                                               remove_repeat_coordinates)
from metpy.cbook import get_test_data
from metpy.calc import get_wind_components
from metpy.units import units
from metpy.plots import StationPlot

from scipy.ndimage.filters import gaussian_filter

from_proj = ccrs.Geodetic()
to_proj = ccrs.AlbersEqualArea(central_longitude=-97.0000, central_latitude=38.0000)

plt.rcParams['figure.figsize'] = 20,15
warnings.filterwarnings('ignore')
```

```
def make_string_list(arr):
    return [s.decode('ascii') for s in arr]

def station_test_data(variable_names, proj_from=None, proj_to=None):

    f = get_test_data('station_data.txt')

    all_data = np.loadtxt(f, skiprows=1, delimiter=',',
                          usecols=(1, 2, 3, 4, 5, 6, 7, 17, 18, 19),
                          dtype=np.dtype([('stid', '3S'), ('lat', 'f'), ('lon', 'f'),
                                          ('slp', 'f'), ('air_temperature', 'f'),
                                          ('cloud_fraction', 'f'), ('dewpoint', 'f'),
                                          ('weather', '16S'),
                                          ('wind_dir', 'f'), ('wind_speed', 'f')]))

    all_stids = make_string_list(all_data['stid'])

    data = np.concatenate([all_data[all_stids.index(site)].reshape(1, ) for site in all_stids])

    value = data[variable_names]
    lon = data['lon']
    lat = data['lat']

    if proj_from is not None and proj_to is not None:
            proj_points = proj_to.transform_points(proj_from, lon, lat)
            return proj_points[:, 0], proj_points[:, 1], value

    return lon, lat, value
```

Get pressure information using the sample station data

```
xp, yp, pres = station_test_data(['slp'], from_proj, to_proj)
```

Remove all missing data from pressure

```
pres = np.array([p[0] for p in pres])

xp, yp, pres = remove_nan_observations(xp, yp, pres)
```

Interpolate pressure as usual

```
slpgridx, slpgridy, slp = interpolate(xp, yp, pres, interp_type="cressman", minimum_neighbors=1,
                                      search_radius = 400000, hres=100000)
```

Get wind information

```
x, y, wind = station_test_data(['wind_speed', 'wind_dir'], from_proj, to_proj)
```

Remove bad data from wind information

```
wind_speed = np.array([w[0] for w in wind])
wind_dir = np.array([w[1] for w in wind])

good_indices = np.where((~np.isnan(wind_dir)) & (~np.isnan(wind_speed)))

x = x[good_indices]
y = y[good_indices]
wind_speed = wind_speed[good_indices]
wind_dir = wind_dir[good_indices]
```

Calculate u and v components of wind and then interpolate both.

Both will have the same underlying grid so throw away grid returned from v interpolation.

```
u, v = get_wind_components((wind_speed * units('m/s')).to('knots'),
                           wind_dir * units.degree)


windgridx, windgridy, uwind = interpolate(x, y, np.array(u), interp_type="cressman",
                                          search_radius = 400000, hres=100000)

_, _, vwind = interpolate(x, y, np.array(v), interp_type="cressman", search_radius = 400000, hres=100
```

Get temperature information

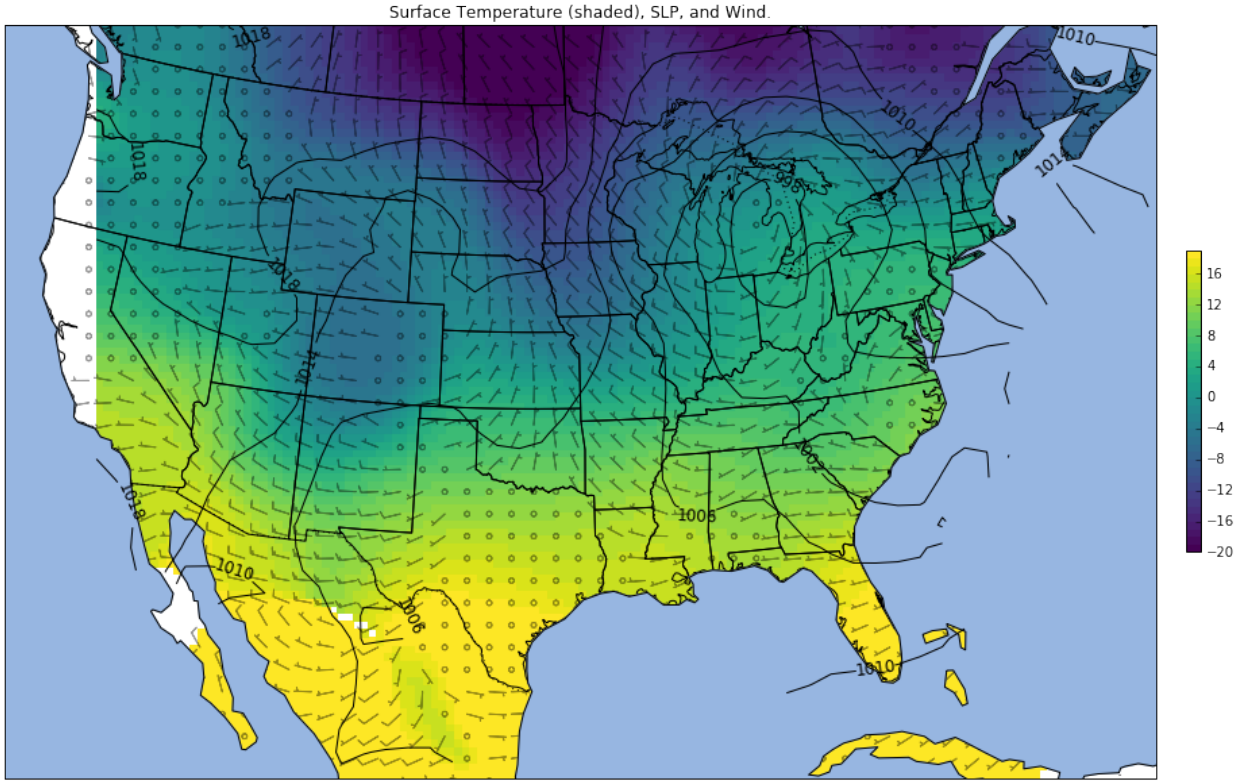
```
from matplotlib.colors import BoundaryNorm

levels = list(range(-20, 20, 1))
cmap = plt.get_cmap('viridis')
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

xt, yt, t = station_test_data("air_temperature", from_proj, to_proj)
xt, yt, t = remove_nan_observations(xt, yt, t)

tempx, tempy, temp = interpolate(xt, yt, t, interp_type='cressman', minimum_neighbors=3,
                                 search_radius = 400000, hres=35000)

temp = np.ma.masked_where(np.isnan(temp), temp)
```

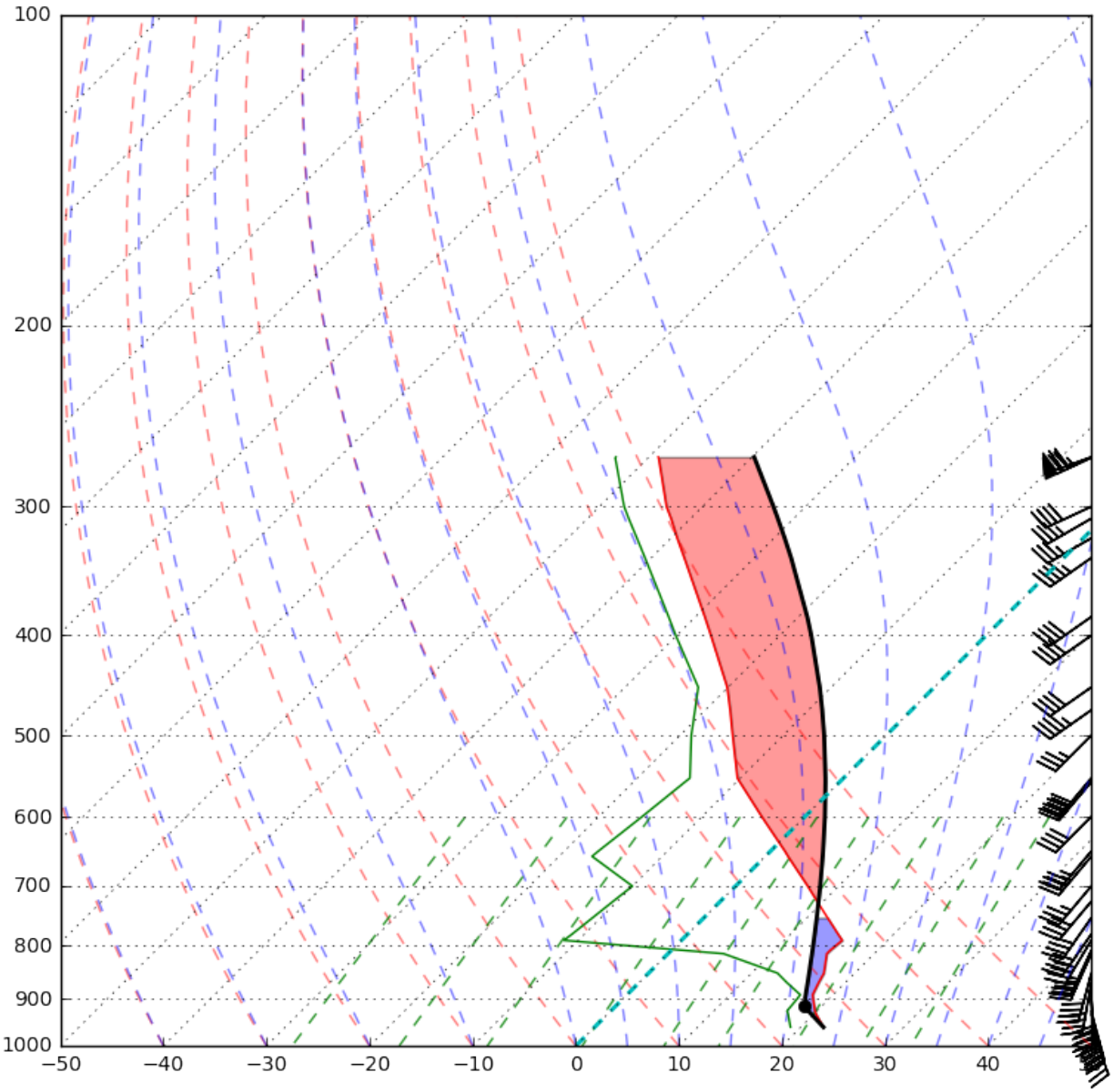
Set up the map and plot the interpolated grids appropriately.

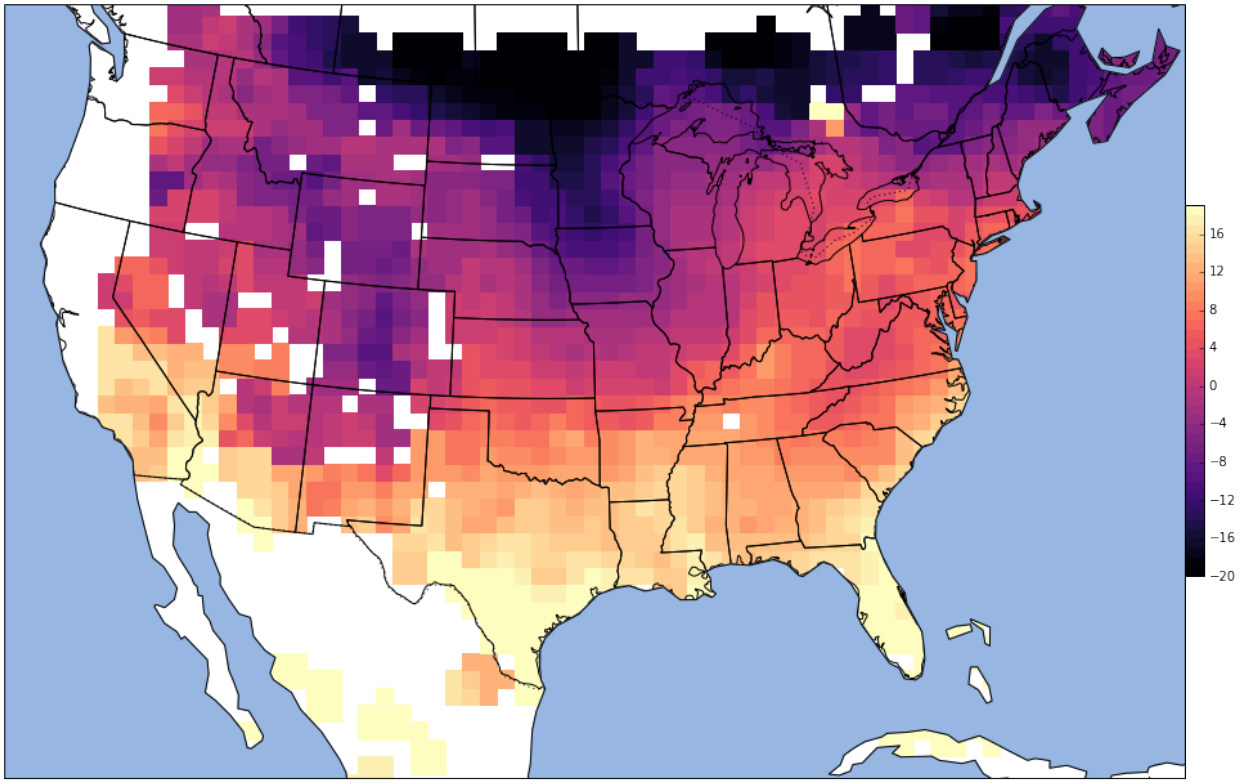
```
fig = plt.figure(figsize=(20, 10))
view = fig.add_subplot(1, 1, 1, projection=to_proj)


view.set_extent([-120, -70, 20, 50])
view.add_feature(cartopy.feature.NaturalEarthFeature(
                                 category='cultural',
                                 name='admin_1_states_provinces_lakes',
                                 scale='50m',
                                 facecolor='none'))
```

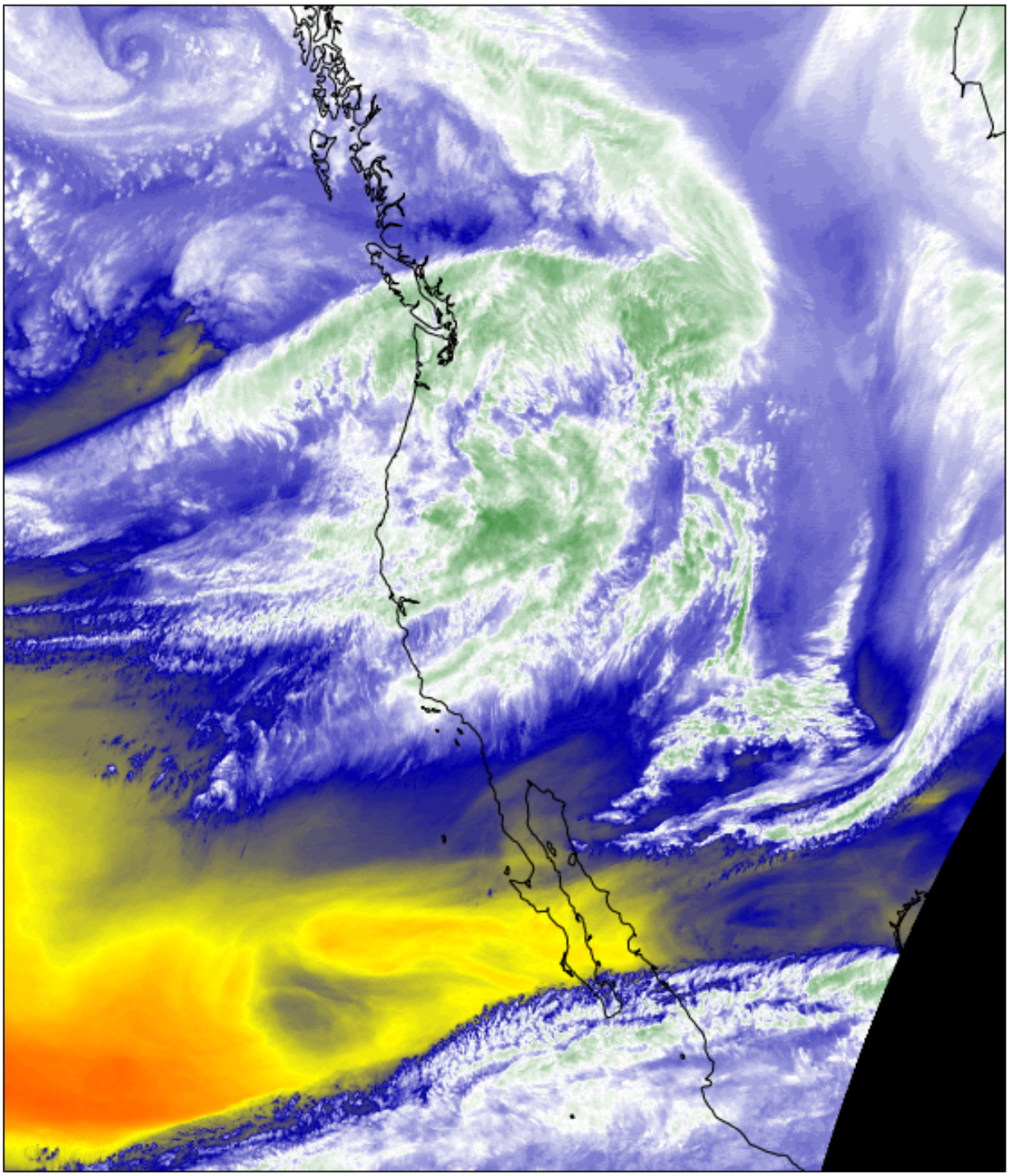
```
view.add_feature(cartopy.feature.OCEAN)
view.add_feature(cartopy.feature.COASTLINE)
view.add_feature(cartopy.feature.BORDERS, linestyle=':')

cs = view.contour(slpgridx, slpgridy, slp, colors=['k',], levels=list(range(990, 1034, 4)))

plt.clabel(cs, inline=1, fontsize=12, fmt='%i')

mmb = view.pcolormesh(tempx, tempy, temp, cmap=cmap, norm=norm)
plt.colorbar(mmb, shrink=.4, pad=0.02, boundaries=levels)

view.barbs(windgridx, windgridy, uwind, vwind, alpha=.4, length=5)

plt.title("Surface Temperature (shaded), SLP, and Wind.")
```

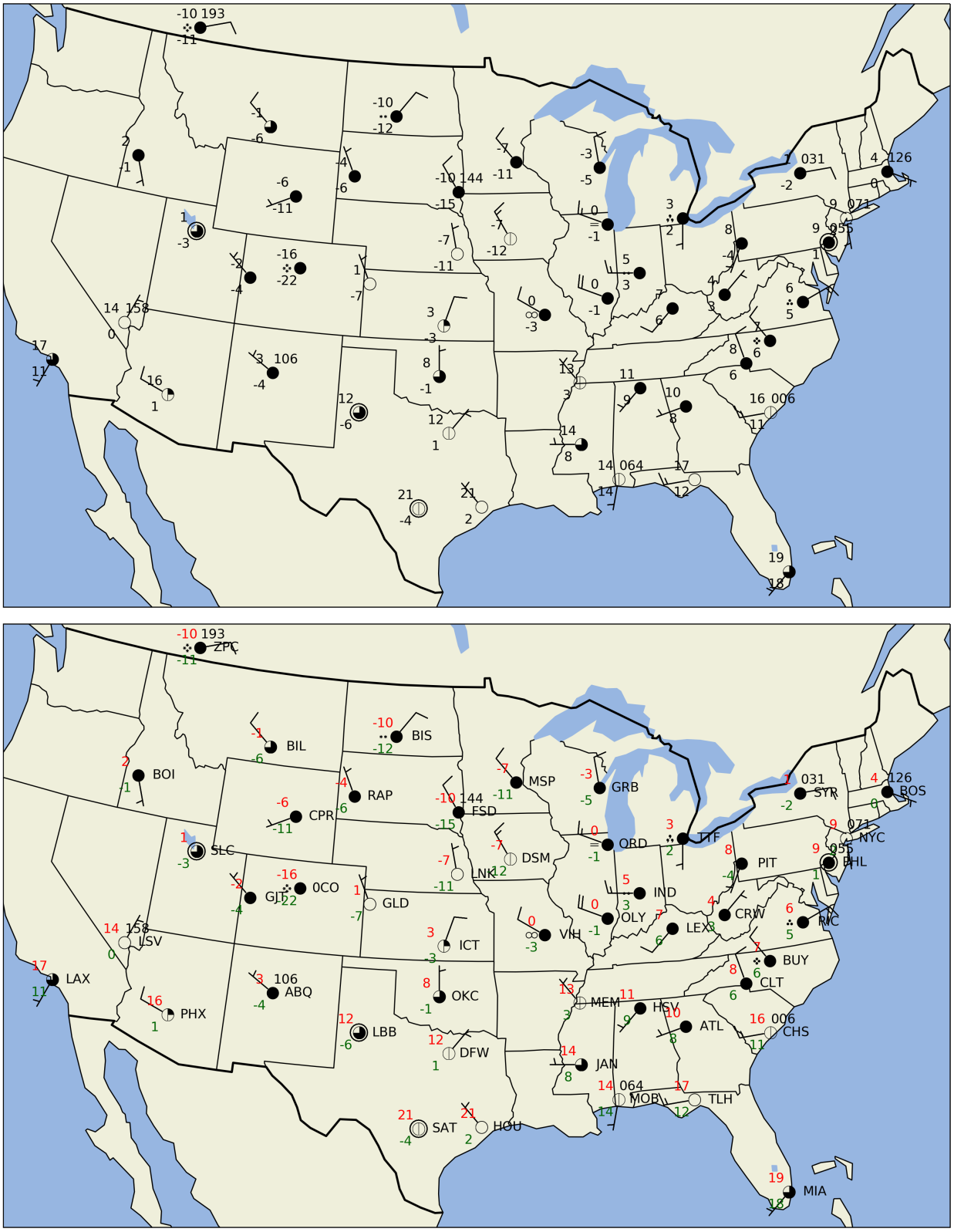
```
<matplotlib.text.Text at 0x110d5bb00>
```



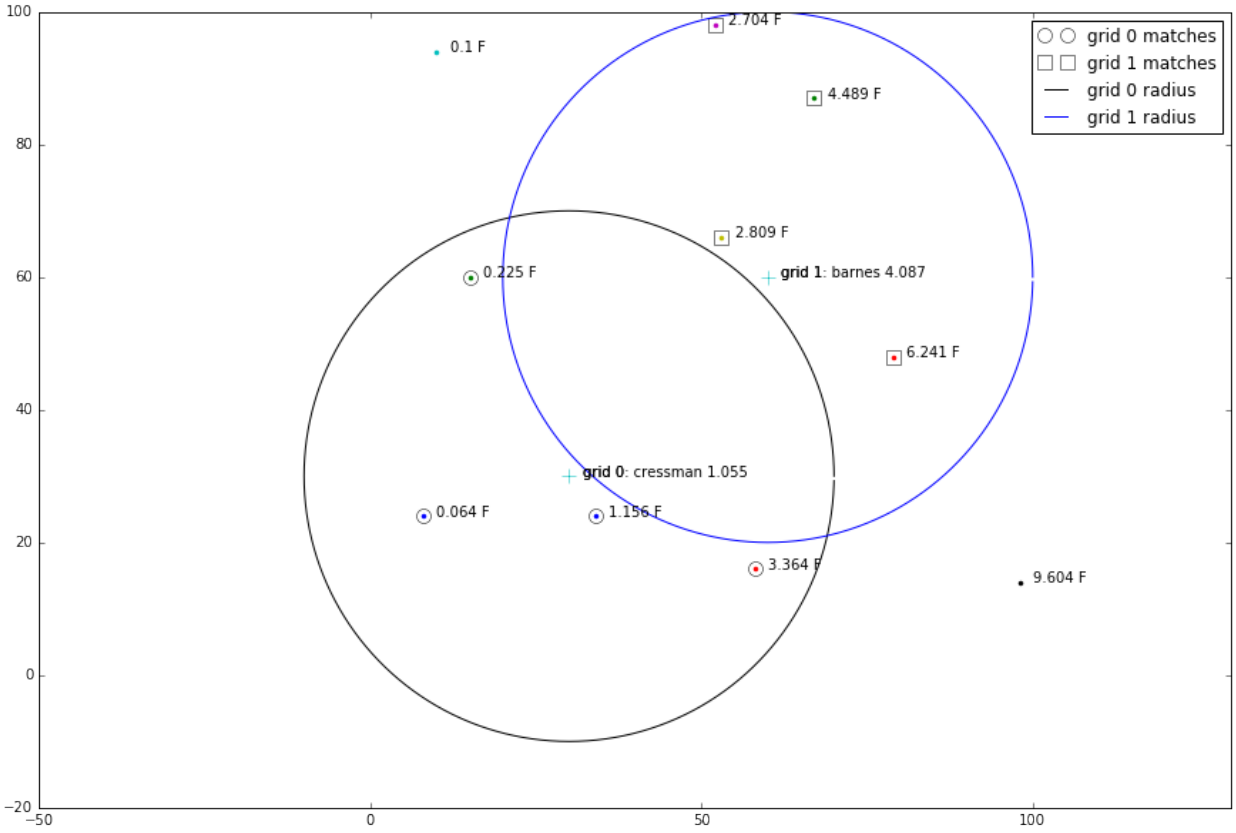

100
200
300
400
500
600
700
800
900
1000
−50
−40
−30
−20
−10
0
10
20
30
40

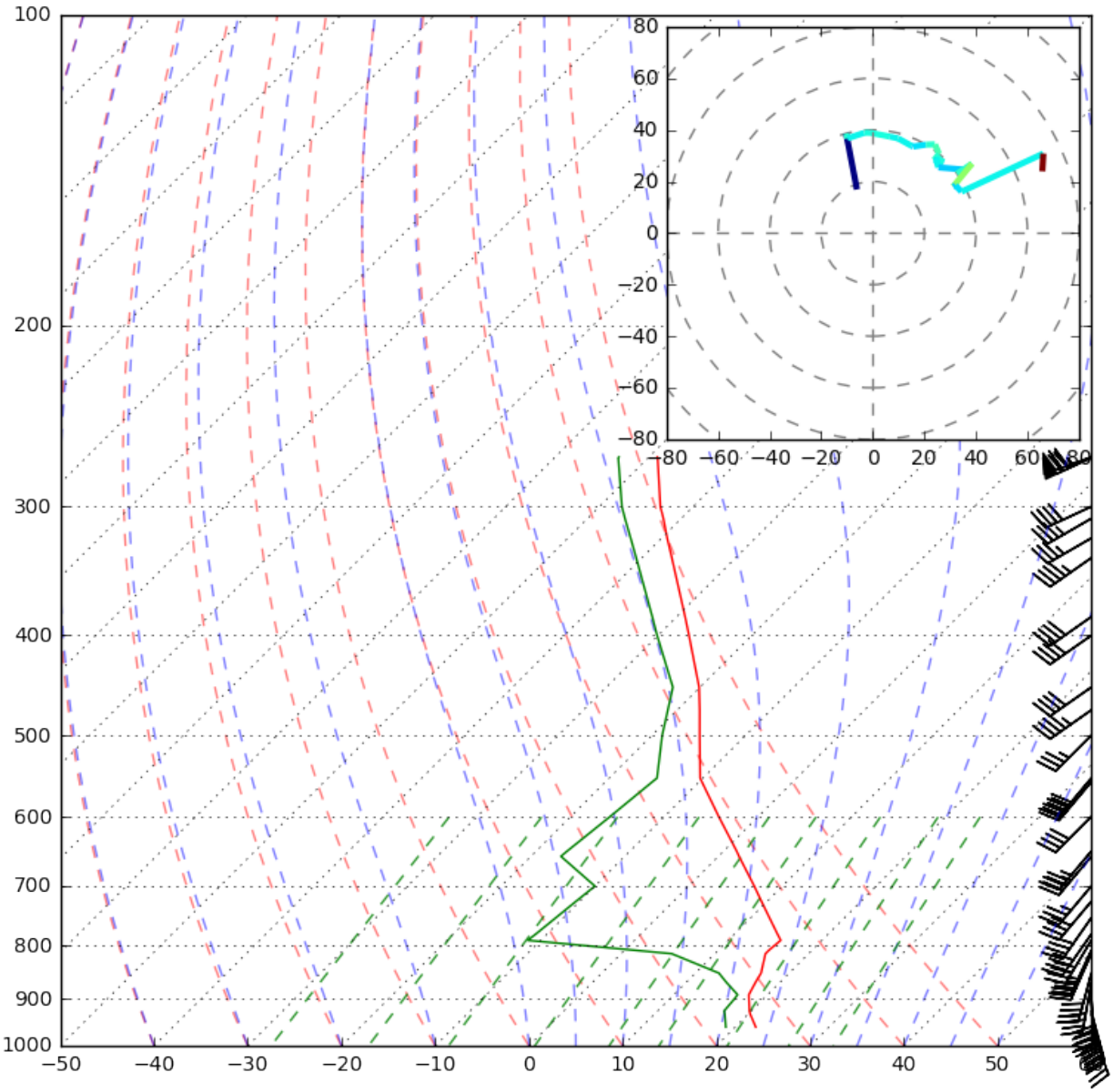

16
12
8
4
0
−4
−8
−12
−16
−20

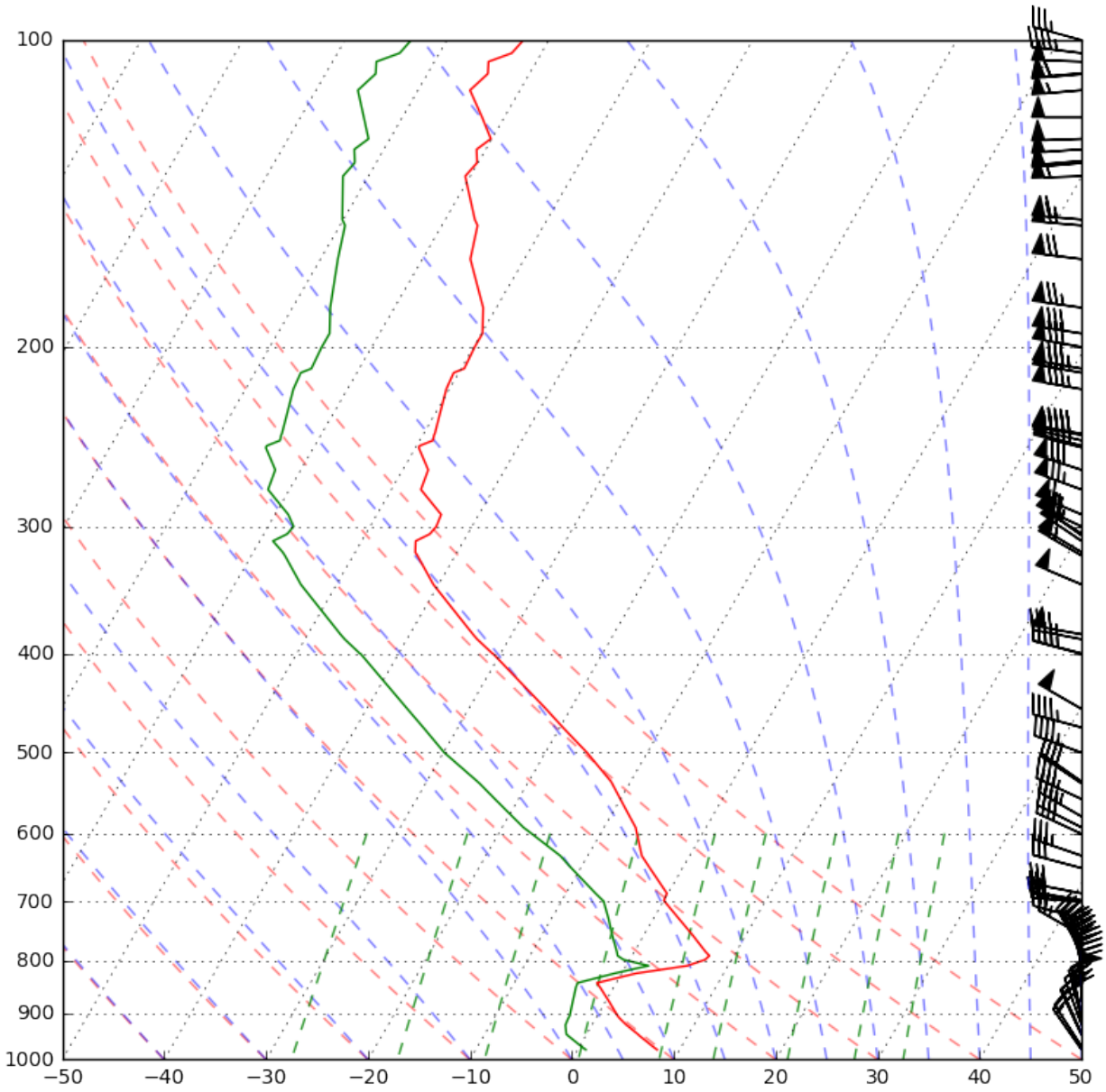


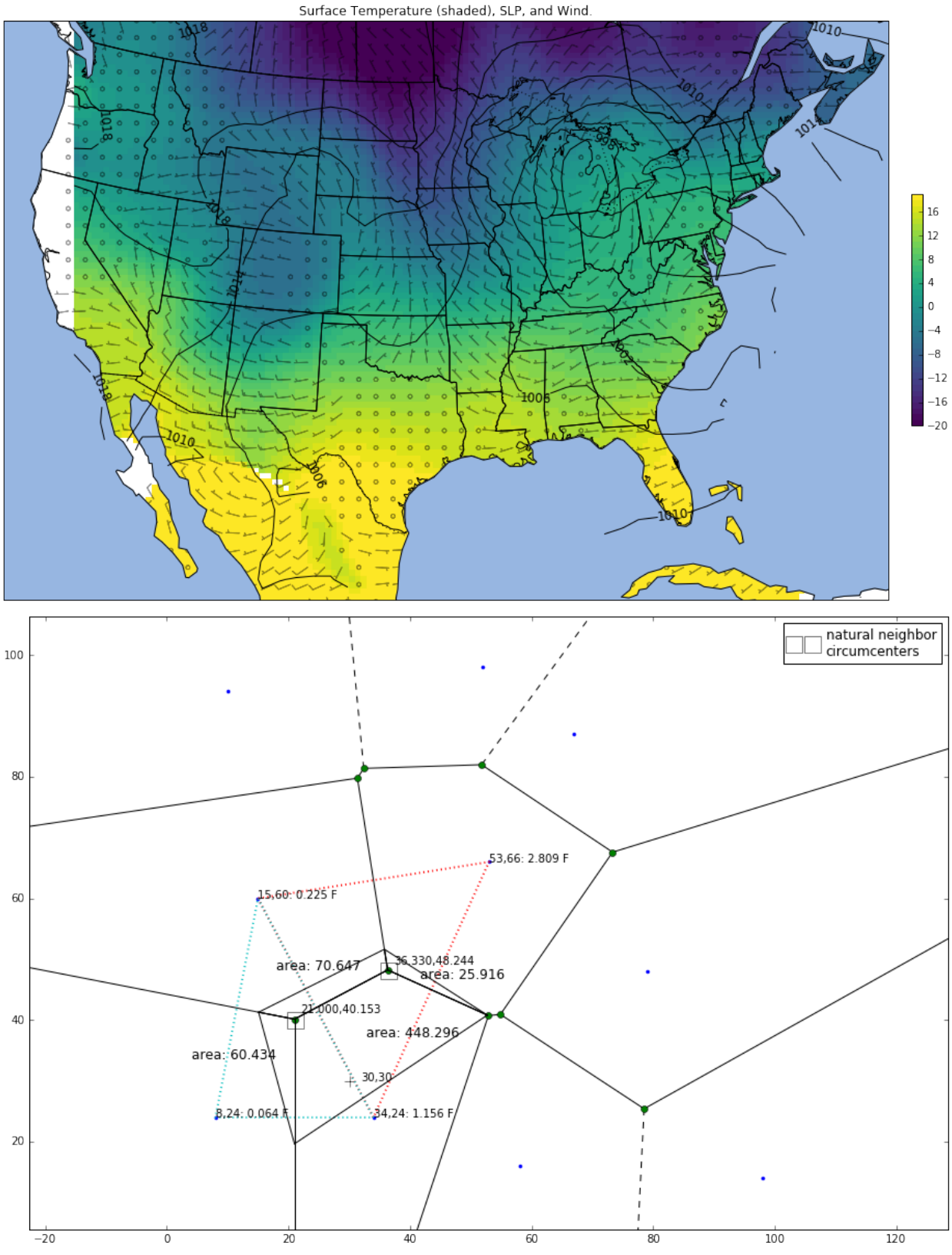

-10 193
-11
-10
-12
-1
-6
2
-1
-4
-6
-6
-11
-10 144
-15
-7
-11
-7
-12
-3
-5
0
-1
3
2
1 031
-2
4 126
0
9 071
9 055
8
-4
1
1
-3
-2
-16
-22
-4
1
-7
-11
-7
5
0
-1
3
7
6
4
3
6
5
14 158
0
17
11
3 106
-4
16
1
3
-3
8
-1
0
-3
8
6
6
13
11
3
9
10
8
16 006
11
12
-6
12
1
14
8
14 064
17
14
12
21
-4
21
2
19
18

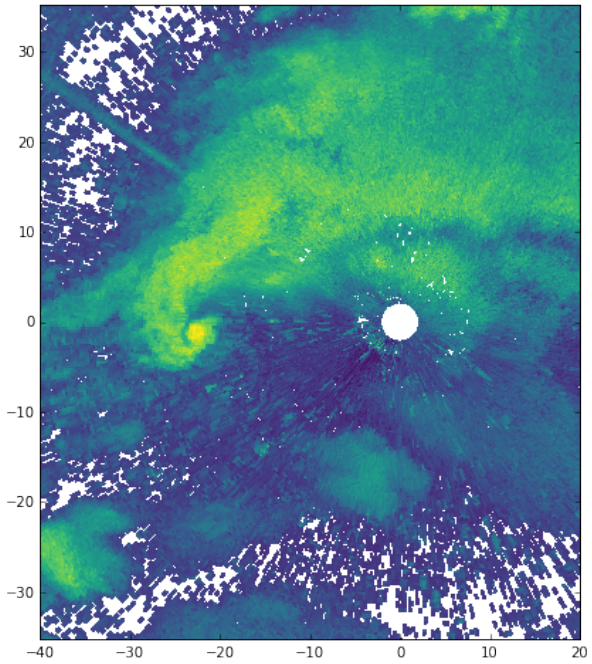
-10 193
ZPC
-11
-10
BIS
-12
-1
BIL
-6
2
BOI
-1
-4
RAP
-6
-6
CPR
-11
-10 144
FSD
-15
-7
-11
-7
MSP
-3
GRB
-5
1 031
SYR
-2
4 126
BOS
0
9 071
NYC
9 055
PHL
1
1
SLC
-3
-16
-22
0CO
-4
1
GLD
-7
-11
-7
DSM
12
NK
0
-1
ORD
3
2
TTF
8
PIT
-4
5
IND
0
3
OLY
7
6
LEX
4
CRW
6
5
RIC
14 158
LSV
0
3
ICT
-3
0
-3
VIH
-1
17
11
LAX
3 106
-4
ABQ
16
1
PHX
8
-1
OKC
12
-6
LBB
12
1
DFW
13
11
3
9
MEM
HSV
10
8
ATL
14
8
JAN
14 064
MOB
17
14
12
TLH
16 006
11
CHS
8
6
BUY
6
CLT
21
-4
SAT
21
2
HOU
19
18
MIA

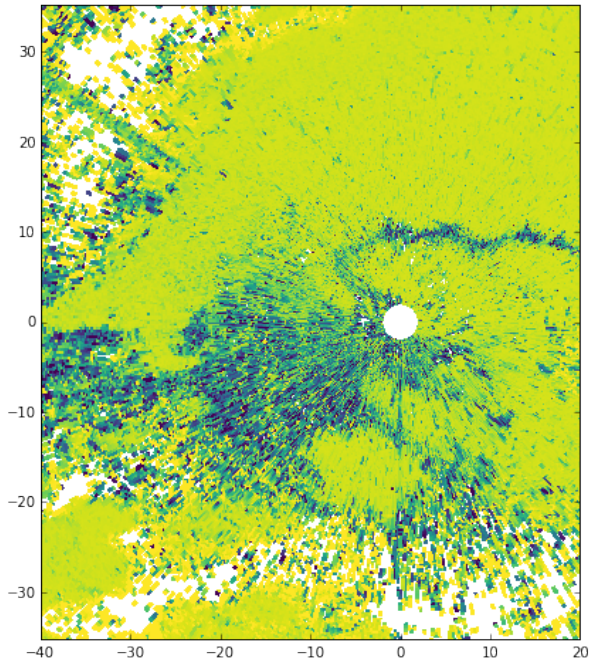

grid 0 matches
grid 1 matches
grid 0 radius
grid 1 radius
2.704 F
0.1 F
4.489 F
2.809 F
0.225 F
grid 1: barnes 4.087
6.241 F
grid 0: cressman 1.055
0.064 F
1.156 F
3.364 F
9.604 F
100
80
60
40
20
0
−20
−50
0
50
100

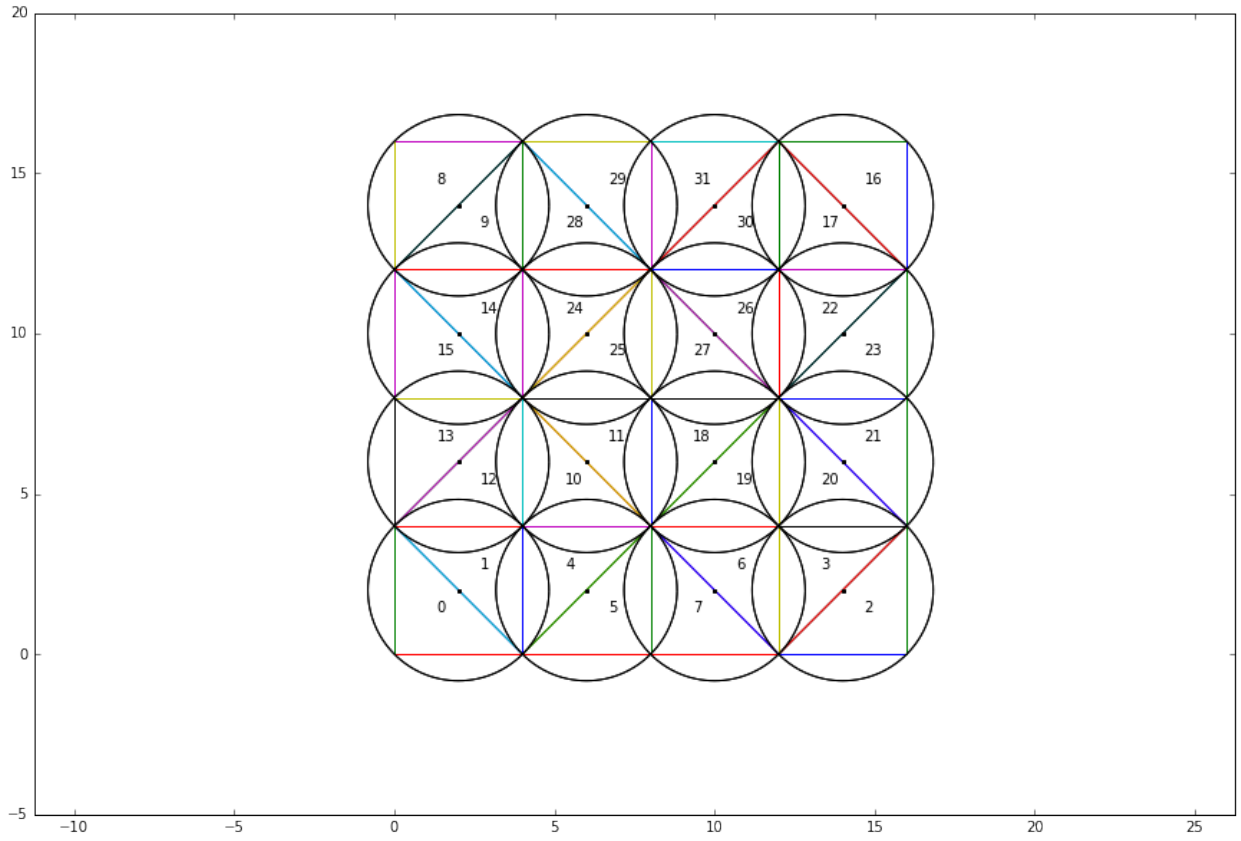

100
200
300
400
500
600
700
800
900
1000
−50
−40
−30
−20
−10
0
10
20
30
40
50
80
60
40
20
0
−20
−40
−60
−80
−80
−60
−40
−20
0
20
40
60
80

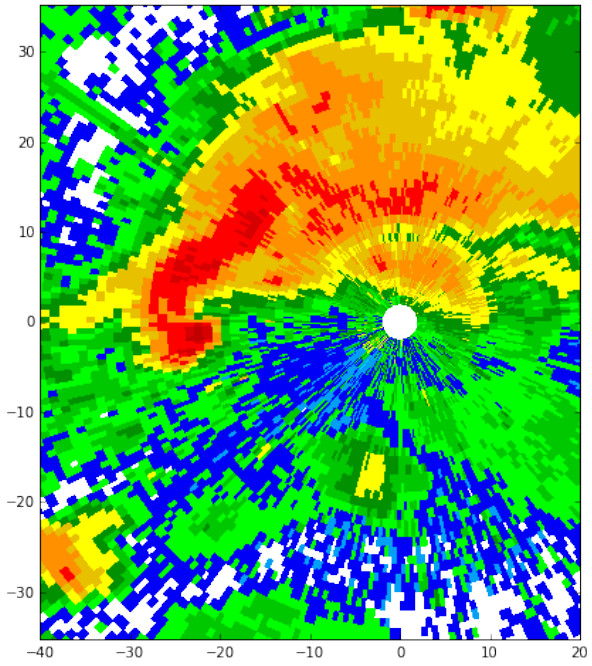

100
200
300
400
500
600
700
800
900
1000
−50
−40
−30
−20
−10
0
10
20
30
40
50

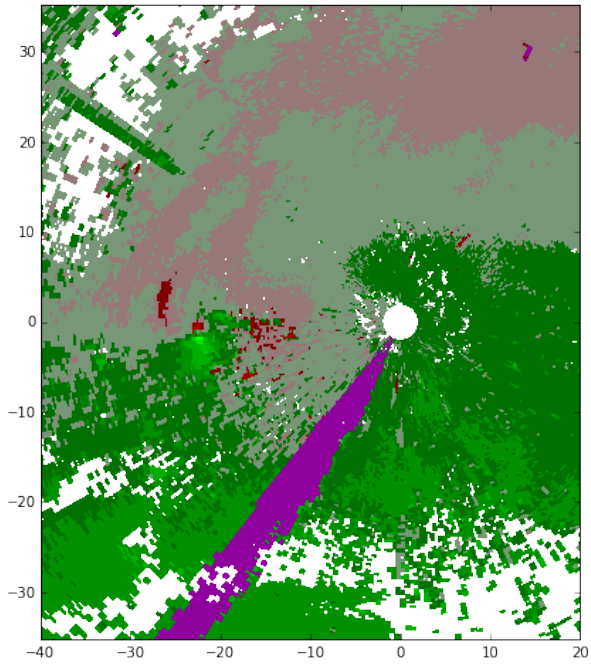

Surface Temperature (shaded), SLP, and Wind.
natural neighbor
circumcenters
53,66: 2.809 F
15,60: 0.225 F
area: 70.647
16.330,48.244
area: 25.916
21.000,40.153
area: 448.296
area: 60.434
30,30
8,24: 0.064 F
34,24: 1.156 F

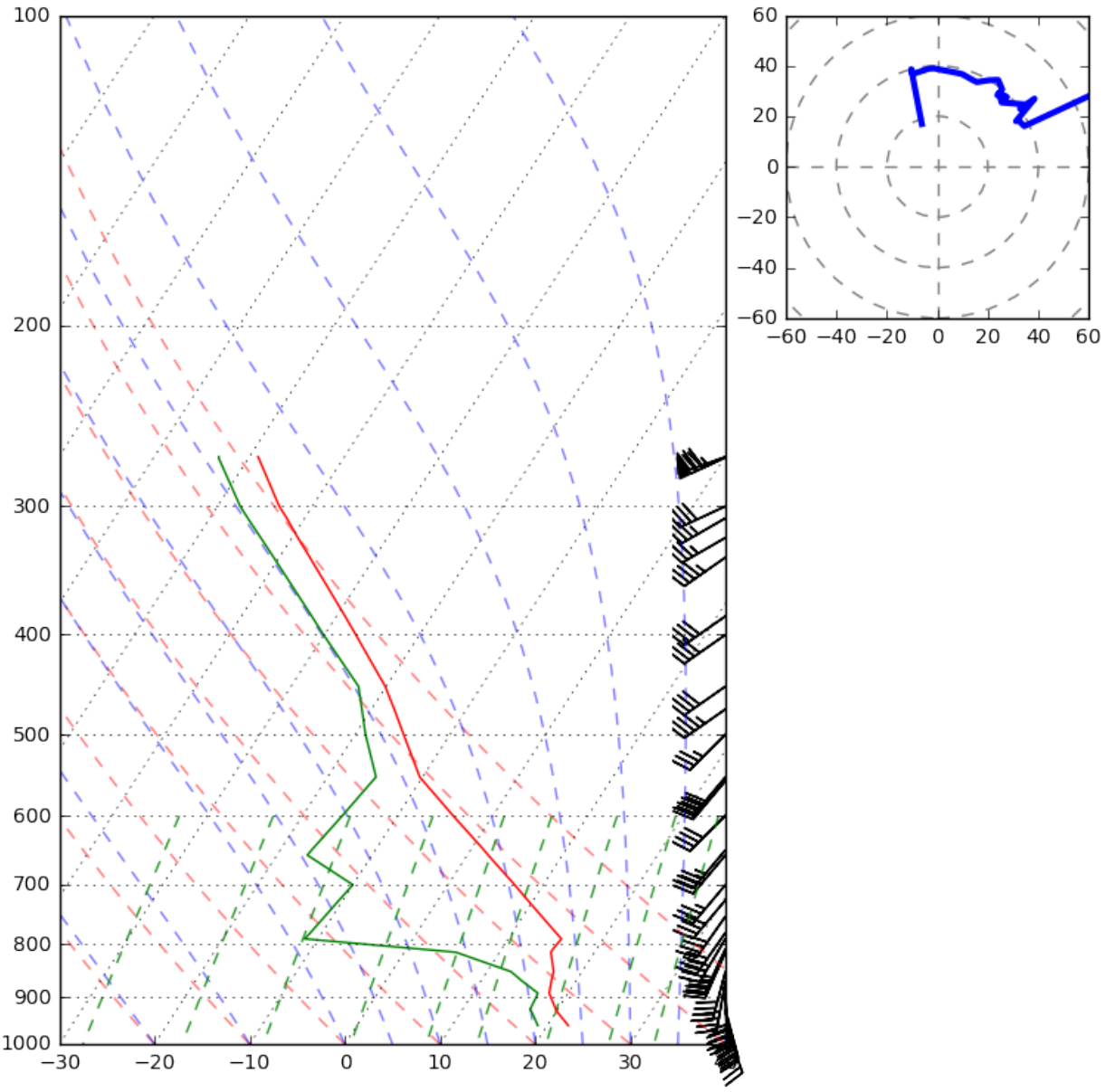

30
20
10
0
−10
−20
−30
−40
−30
−20
−10
0
10
20


30
20
10
0
−10
−20
−30
−40
−30
−20
−10
0
10
20


20
15
10
5
0
−5
−10
−5
0
5
10
15
20
25
8
29
31
16
9
28
30
17
14
24
26
22
15
25
27
23
13
11
18
21
12
10
19
20
1
4
6
3
0
5
7
2


30
20
10
0
−10
−20
−30
−40
−30
−20
−10
0
10
20


30
20
10
0
−10
−20
−30
−40
−30
−20
−10
0
10
20



Dewpoint and Mixing Ratio

1.4 The MetPy API

1.4.1 `metpy.constants`

This is a collection of meteorologically significant constants.

Earth

Name	Abbr.	Units	Description
earth_avg_radius	Re	m	Avg. radius of the Earth
earth_gravity	g	m s^-2	Avg. gravity acceleration on Earth
earth_avg_angular_vel	omega	rad s^-1	Avg. angular velocity of Earth
earth_sfc_avg_dist_sun	d	m	Avg. distance of the Earth from the Sun
earth_solar_irradiance	S	W m^-2	Avg. solar irradiance of Earth
earth_max_declination	delta	degrees	Max. solar declination angle of Earth
earth_orbit_eccentricity		None	Avg. eccentricity of Earth's orbit

Water

Name	Abbr.	Units	Description
water_molecular_weight	Mw	g mol^-1	Molecular weight of water
water_gas_constant	Rv	J (K kg)^-1	Gas constant for water vapor
density_water	rho_l	kg m^-3	Nominal density of liquid water at 0C
wv_specific_heat_press	Cp_v	J (K kg)^-1	Specific heat at constant pressure for water vapor
wv_specific_heat_vol	Cv_v	J (K kg)^-1	Specific heat at constant volume for water vapor
water_specific_heat	Cp_l	J (K kg)^-1	Specific heat of liquid water at 0C
water_heat_vaporization	Lv	J kg^-1	Latent heat of vaporization for liquid water at 0C
water_heat_fusion	Lf	J kg^-1	Latent heat of fusion for liquid water at 0C
ice_specific_heat	Cp_i	J (K kg)^-1	Specific heat of ice at 0C
density_ice	rho_i	kg m^-3	Density of ice at 0C

Dry Air

Name	Abbr.	Units	Description
dry_air_molecular_weight	Md	g / mol	Nominal molecular weight of dry air at the surface of th Earth
dry_air_gas_constant	Rd	J (K kg)^-1	Gas constant for dry air at the surface of the Earth
dry_air_spec_heat_press	Cp_d	J (K kg)^-1	Specific heat at constant pressure for dry air
dry_air_spec_heat_vol	Cv_d	J (K kg)^-1	Specific heat at constant volume for dry air
dry_air_density_stp	rho_d	kg m^-3	Density of dry air at 0C and 1000mb

General Meteorology Constants

Name	Abbr.	Units	Description
pot_temp_ref_press	P0	Pa	Reference pressure for potential temperature
poisson_exponent	kappa	None	Exponent in Poisson's equation (Rd/Cp_d)
dry_adiabatic_lapse_rate	gamma_d	K km^-1	The dry adiabatic lapse rate
molecular_weight_ratio	epsilon	None	Ratio of molecular weight of water to that of dry air

1.4.2 `metpy.units`

Module to provide unit support.

This makes use of the `pint` library and sets up the default settings for good temperature support.

`metpy.units.`**`units`**
    `pint.UnitRegistry`

The unit registry used throughout the package. Any use of units in MetPy should import this registry and use it to grab units.

metpy.units.**atleast_1d**(**arrs*)
Convert inputs to arrays with at least one dimension

Scalars are converted to 1-dimensional arrays, whilst other higher-dimensional inputs are preserved. This is a thin wrapper around *numpy.atleast_1d* to preserve units.

Parameters **arrs** (*arbitrary positional arguments*) – Input arrays to be converted if necessary

Returns *pint.Quantity* – A single quantity or a list of quantities, matching the number of inputs.

metpy.units.**atleast_2d**(**arrs*)
Convert inputs to arrays with at least two dimensions

Scalars and 1-dimensional arrays are converted to 2-dimensional arrays, whilst other higher-dimensional inputs are preserved. This is a thin wrapper around *numpy.atleast_2d* to preserve units.

Parameters **arrs** (*arbitrary positional arguments*) – Input arrays to be converted if necessary

Returns *pint.Quantity* – A single quantity or a list of quantities, matching the number of inputs.

metpy.units.**concatenate**(*arrs*, *axis=0*)
Concatenate multiple values into a new unitized object.

This is essentially a unit-aware version of *numpy.concatenate*. All items must be able to be converted to the same units. If an item has no units, it will be given those of the rest of the collection, without conversion. The first units found in the arguments is used as the final output units.

Parameters

- **arrs** (*Sequence of arrays*) – The items to be joined together
- **axis** (*integer, optional*) – The array axis along which to join the arrays. Defaults to 0 (the first dimension)

Returns *pint.Quantity* – New container with the value passed in and units corresponding to the first item.

1.4.3 metpy.io

MetPy's IO module contains classes for reading files. These classes are written to take both file names (for local files) or file-like objects; this allows reading files that are already in memory (using `io.StringIO`) or remote files (using `urlopen()`).

There are also classes to implement concepts from the Common Data Model (CDM). The purpose of these is to simplify data access by proving an interface similar to that of netcdf4-python.

CDM

metpy.io.cdm The Common Data Model (CDM) is a data model for representing a wide array of data. The goal is to be a simple, universal interface to different datasets. This API is a Python implementation in the spirit of the original Java interface in netCDF-Java.

class metpy.io.cdm.**AttributeContainer**
A class to handle maintaining a list of netCDF attributes. Implements the attribute handling for other CDM classes.

Initialize an *AttributeContainer*.

__init__()
Initialize an *AttributeContainer*.

ncattrs()
Get a list of the names of the netCDF attributes.

Returns *List[str]*

class metpy.io.cdm.**Dataset**
A Dataset represents a set of data using the Common Data Model (CDM).

This is currently only a wrapper around the root Group.

Initialize a Dataset.

__init__()
Initialize a Dataset.

class metpy.io.cdm.**Dimension**(*group*, *name*, *size=None*)
A Dimension is used to represent a shared dimension between different Variables. For instance, variables that are dependent upon a common set of times.

Initialize a Dimension. Instead of constructing a Dimension directly, you should use `Group.createDimension`.

Parameters

- **group** (*Group*) – The parent Group that owns this Variable.
- **name** (*str*) – The name of this Variable.
- **size** (*int or None, optional*) – The size of the Dimension. Defaults to None, which implies an empty dimension.

See also:

Group.createDimension

__init__(*group*, *name*, *size=None*)
Initialize a Dimension. Instead of constructing a Dimension directly, you should use `Group.createDimension`.

Parameters

- **group** (*Group*) – The parent Group that owns this Variable.
- **name** (*str*) – The name of this Variable.
- **size** (*int or None, optional*) – The size of the Dimension. Defaults to None, which implies an empty dimension.

See also:

Group.createDimension()

group()
Get the Group that owns this Dimension.

Returns *Group* – The parent Group.

name = None

Desc The name of the Dimension

Type str

size = None

Desc The size of this Dimension

Type int

class metpy.io.cdm.**Group**(*parent*, *name*)

A Group holds dimensions and variables. Every CDM dataset has at least a root group.

Initialize this *Group*. Instead of constructing a *Group* directly, you should use *createGroup()*.

Parameters

- **parent** (*Group or None*) – The parent Group for this one. Passing in None implies that this is the root *Group*.
- **name** (*str*) – The name of this group

See also:

Group.createGroup

__init__(*parent*, *name*)

Initialize this *Group*. Instead of constructing a *Group* directly, you should use *createGroup()*.

Parameters

- **parent** (*Group or None*) – The parent Group for this one. Passing in None implies that this is the root *Group*.
- **name** (*str*) – The name of this group

See also:

Group.createGroup()

createDimension(*name*, *size*)

Create a new *Dimension* in this *Group*.

Parameters

- **name** (*str*) – The name of the new Dimension.
- **size** (*int*) – The size of the Dimension

Returns *Dimension* – The newly created *Dimension*

createGroup(*name*)

Create a new Group as a descendant of this one.

Parameters **name** (*str*) – The name of the new Group.

Returns *Group* – The newly created *Group*

createVariable(*name*, *datatype*, *dimensions=()*, *fill_value=None*, *wrap_array=None*)

Create a new Variable in this Group.

Parameters

- **name** (*str*) – The name of the new Variable.
- **datatype** (*str or numpy.dtype*) – A valid Numpy dtype that describes the layout of the data within the Variable.
- **dimensions** (*tuple[str], optional*) – The dimensions of this Variable. Defaults to empty, which implies a scalar variable.

- **fill_value** (*number, optional*) – A scalar value that is used to fill the created storage. Defaults to None, which performs no filling, leaving the storage uninitialized.
- **wrap_array** (*numpy.ndarray, optional*) – Instead of creating an array, the Variable instance will assume ownership of the passed in array as its data storage. This is a performance optimization to avoid copying large data blocks. Defaults to None, which means a new array will be created.

Returns *Variable* – The newly created *Variable*

dimensions = None

Desc Dimensions contained within this group

Type dict[str, Dimension]

groups = None

Desc Any Groups nested within this one

Type dict[str, Group]

name = None

Desc The name of the *Group*

Type str

variables = None

Desc Variables contained within this group

Type dict[str, Variable]

class metpy.io.cdm.**Variable**(*group*, *name*, *datatype*, *dimensions*, *fill_value*, *wrap_array*)

A Variable holds typed data (using a numpy.ndarray), as well as any relevant attributes (e.g. units).

In addition to its various attributes, the Variable supports getting *and* setting data using the [] operator and indices or slices. Getting data returns numpy.ndarray instances.

Initialize a Variable. Instead of constructing a Variable directly, you should use *Group.createVariable()*.

Parameters

- **group** (Group) – The parent *Group* that owns this Variable.
- **name** (*str*) – The name of this Variable.
- **datatype** (*str or numpy.dtype*) – A valid Numpy dtype that describes the layout of each element of the data
- **dimensions** (*tuple[str], optional*) – The dimensions of this Variable. Defaults to empty, which implies a scalar variable.
- **fill_value** (*scalar, optional*) – A scalar value that is used to fill the created storage. Defaults to None, which performs no filling, leaving the storage uninitialized.
- **wrap_array** (*numpy.ndarray, optional*) – Instead of creating an array, the Variable instance will assume ownership of the passed in array as its data storage. This is a performance optimization to avoid copying large data blocks. Defaults to None, which means a new array will be created.

See also:

Group.createVariable

__init__(*group*, *name*, *datatype*, *dimensions*, *fill_value*, *wrap_array*)
Initialize a Variable. Instead of constructing a Variable directly, you should use *Group.createVariable()*.

Parameters

- **group** (Group) – The parent *Group* that owns this Variable.
- **name** (*str*) – The name of this Variable.
- **datatype** (*str or numpy.dtype*) – A valid Numpy dtype that describes the layout of each element of the data
- **dimensions** (*tuple[str], optional*) – The dimensions of this Variable. Defaults to empty, which implies a scalar variable.
- **fill_value** (*scalar, optional*) – A scalar value that is used to fill the created storage. Defaults to None, which performs no filling, leaving the storage uninitialized.
- **wrap_array** (*numpy.ndarray, optional*) – Instead of creating an array, the Variable instance will assume ownership of the passed in array as its data storage. This is a performance optimization to avoid copying large data blocks. Defaults to None, which means a new array will be created.

See also:

Group.createVariable()

datatype
numpy.dtype: a valid Numpy dtype that describes the layout of each element of the data

dimensions
tuple[str]: all the names of *Dimension* used by this *Variable*

dtype
numpy.dtype: a valid Numpy dtype that describes the layout of each element of the data

group()
Get the Group that owns this Variable.

Returns *Group* – The parent Group.

name
str: the name of the variable

ndim
int: the number of dimensions used by this variable

shape
tuple[int]: a tuple of integers describing the size of the Variable along each of its dimensions

size
int: the total number of elements

metpy.io.cdm.**cf_to_proj**(*var*)
Converts a Variable with projection information conforming to the Climate and Forecasting (CF) netCDF conventions to a Proj.4 Projection instance.

Parameters **var** (Variable) – The projection variable with appropriate attributes.

File Formats

These classes provide support for reading data from specific formats.

NEXRAD

class metpy.io.nexrad.**Level2File**(*filename*)

A class that handles reading the NEXRAD Level 2 data and the various messages that are contained within.

This class attempts to decode every byte that is in a given data file. It supports both external compression, as well as the internal BZ2 compression that is used.

stid
str

The ID of the radar station

dt
Datetime instance

The date and time of the data

vol_hdr
namedtuple

The unpacked volume header

sweeps
list of tuples

Data for each of the sweeps found in the file

rda_status
namedtuple, optional

Unpacked RDA status information, if found

maintenance_data
namedtuple, optional

Unpacked maintenance data information, if found

maintenance_data_desc
dict, optional

Descriptions of maintenance data fields, if maintenance data present

vcp_info
namedtuple, optional

Unpacked VCP information, if found

clutter_filter_bypass_map
dict, optional

Unpacked clutter filter bypass map, if present

rda
dict, optional

Unpacked RDA adaptation data, if present

rda_adaptation_desc
dict, optional

Descriptions of RDA adaptation data, if adaptation data present

Notes

The internal data structure that things are decoded into is still to be determined.

Create instance of *Level2File*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. Gzip-ed files are recognized with the extension '.gz', as are bzip2-ed files with the extension *.bz2* If *fname* is a file-like object, this will be read from directly.

__init__ (*filename*)
Create instance of *Level2File*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. Gzip-ed files are recognized with the extension '.gz', as are bzip2-ed files with the extension *.bz2* If *fname* is a file-like object, this will be read from directly.

msg1_data_hdr
alias of `Msg1DataHdr`

class `metpy.io.nexrad.`**Level3File** (*filename*)
A class that handles reading the wide array of NEXRAD Level 3 (NIDS) product files.

This class attempts to decode every byte that is in a given product file. It supports all of the various compression formats that exist for these products in the wild.

metadata
dict

Various general metadata available from the product

header
namedtuple

Decoded product header

prod_desc
namedtuple

Decoded product description block

siteID
str

ID of the site found in the header, empty string if none found

lat
float

Radar site latitude

lon
float

Radar site longitude

height
float

Radar site height AMSL

product_name
str

Name of the product contained in file

max_range
float

Maximum range of the product, taken from the NIDS ICD

map_data
Mapper

Class instance mapping data int values to proper floating point values

sym_block
list, optional

Any symbology block packets that were found

tab_pages
list, optional

Any tabular pages that were found

graph_pages
list, optional

Any graphical pages that were found

Notes

The internal data structure that things are decoded into is still to be determined.

Create instance of *Level3File*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. If file-like object, this will be read from directly.

__init__(*filename*)
Create instance of *Level3File*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. If file-like object, this will be read from directly.

metpy.io.nexrad.**is_precip_mode**(*vcp_num*)
Determine if the NEXRAD radar is operating in precipitation mode

Parameters **vcp_num** (*int*) – The NEXRAD volume coverage pattern (VCP) number

Returns *bool* – True if the VCP corresponds to precipitation mode, False otherwise

GINI

class metpy.io.gini.**GiniFile**(*filename*)
A class that handles reading the GINI format satellite images from the NWS.

This class attempts to decode every byte that is in a given GINI file.

Notes

The internal data structures that things are decoded into are subject to change. For a more stable interface, use the *to_dataset()* method.

See also:

GiniFile.to_dataset

Create instance of *GiniFile*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. Gzip-ed files are recognized with the extension `'.gz'`, as are bzip2-ed files with the extension `'.bz2'` If *filename* is a file-like object, this will be read from directly.

__init__ (*filename*)
Create instance of *GiniFile*.

Parameters **filename** (*str or file-like object*) – If str, the name of the file to be opened. Gzip-ed files are recognized with the extension `'.gz'`, as are bzip2-ed files with the extension `'.bz2'` If *filename* is a file-like object, this will be read from directly.

prod_desc = None

Desc Decoded first section of product description block

Type namedtuple

prod_desc2 = None

Desc Decoded second section of product description block

Type namedtuple

proj_info = None

Desc Decoded geographic projection information

Type namedtuple

to_dataset()
Convert to a CDM dataset.

Gives a representation of the data in a much more user-friendly manner, providing easy access to Variables and relevant attributes.

Returns *Dataset*

1.4.4 metpy.calc

This module contains a variety of meteorological calculations.

These are the major groups of calculations available in MetPy:

Basic Calculations

metpy.calc.basic.**get_wind_speed**(*u*, *v*)
Compute the wind speed from u and v-components.

Parameters

- **u** (*array_like*) – Wind component in the X (East-West) direction
- **v** (*array_like*) – Wind component in the Y (North-South) direction

Returns **wind speed** (*array_like*) – The speed of the wind

See also:

get_wind_components()

metpy.calc.basic.**get_wind_dir**(*u*, *v*)
Compute the wind direction from u and v-components.

Parameters

- **u** (*array_like*) – Wind component in the X (East-West) direction
- **v** (*array_like*) – Wind component in the Y (North-South) direction

Returns **wind direction** (*array_like*) – The direction of the wind in degrees, specified as the direction from which it is blowing

See also:

get_wind_components()

metpy.calc.basic.**get_wind_components**(*speed*, *wdir*)
Calculate the U, V wind vector components from the speed and direction.

Parameters

- **speed** (*array_like*) – The wind speed (magnitude)
- **wdir** (*array_like*) – The wind direction, specified as the direction from which the wind is blowing.

Returns **u, v** (*tuple of array_like*) – The wind components in the X (East-West) and Y (North-South) directions, respectively.

See also:

get_speed_dir()

Examples

```
>>> from metpy.units import units
>>> metpy.calc.get_wind_components(10. * units('m/s'), 225. * units.deg)
(<Quantity(7.071067811865475, 'meter / second')>,
 <Quantity(7.071067811865477, 'meter / second')>)
```

metpy.calc.basic.**windchill**(*temperature*, *speed*, *face_level_winds=False*, *mask_undefined=True*)
Calculate the Wind Chill Temperature Index (WCTI) from the current temperature and wind speed.

Specifically, these formulas assume that wind speed is measured at 10m. If, instead, the speeds are measured at face level, the winds need to be multiplied by a factor of 1.5 (this can be done by specifying *face_level_winds* as True.)

Parameters

- **temperature** (*array_like*) – The air temperature
- **speed** (*array_like*) – The wind speed at 10m. If instead the winds are at face level, *face_level_winds* should be set to True and the 1.5 multiplicative correction will be applied automatically.

Returns *array_like* – The corresponding Wind Chill Temperature Index value(s)

Other Parameters

- **face_level_winds** (*bool, optional*) – A flag indicating whether the wind speeds were measured at facial level instead of 10m, thus requiring a correction. Defaults to False.

- **mask_undefined** (*bool, optional*) – A flag indicating whether a masked array should be returned with values where wind chill is undefined masked. These are values where the temperature > 50F or wind speed <= 3 miles per hour. Defaults to True.

See also:

heat_index()

References

metpy.calc.basic.**heat_index**(*temperature*, *rh*, *mask_undefined=True*)

Calculate the Heat Index from the current temperature and relative humidity.

The implementation uses the formula outlined in [6].

Parameters

- **temperature** (*array_like*) – Air temperature
- **rh** (*array_like*) – The relative humidity expressed as a unitless ratio in the range [0, 1]. Can also pass a percentage if proper units are attached.

Returns *array_like* – The corresponding Heat Index value(s)

Other Parameters **mask_undefined** (*bool, optional*) – A flag indicating whether a masked array should be returned with values where heat index is undefined masked. These are values where the temperature < 80F or relative humidity < 40 percent. Defaults to True.

See also:

windchill()

References

metpy.calc.basic.**pressure_to_height_std**(*pressure*)

Convert pressure data to heights using the U.S. standard atmosphere.

The implementation uses the formula outlined in [7].

Parameters **pressure** (*array_like*) – Atmospheric pressure

Returns *array_like* – The corresponding height value(s)

Notes

$$Z = \frac{T_0}{\Gamma}[1 - \frac{p}{p_0}^{\frac{R\Gamma}{g}}]$$

References

metpy.calc.basic.**coriolis_parameter**(*latitude*)

Calculate the coriolis parameter at each point.

The implementation uses the formula outlined in [8].

Parameters **latitude** (*array_like*) – Latitude at each point

Returns *array_like* – The corresponding coriolis force at each point

References

Kinematic Calculations

metpy.calc.kinematics.**v_vorticity**(*u*, *v*, *dx*, *dy*)
Calculate the vertical vorticity of the horizontal wind.

The grid must have a constant spacing in each direction.

Parameters

- **u** (*(X, Y) ndarray*) – x component of the wind
- **v** (*(X, Y) ndarray*) – y component of the wind
- **dx** (*float*) – The grid spacing in the x-direction
- **dy** (*float*) – The grid spacing in the y-direction

Returns *(X, Y) ndarray* – vertical vorticity

See also:

h_convergence(), *convergence_vorticity()*

metpy.calc.kinematics.**h_convergence**(*u*, *v*, *dx*, *dy*)
Calculate the horizontal convergence of the horizontal wind.

The grid must have a constant spacing in each direction.

Parameters

- **u** (*(X, Y) ndarray*) – x component of the wind
- **v** (*(X, Y) ndarray*) – y component of the wind
- **dx** (*float*) – The grid spacing in the x-direction
- **dy** (*float*) – The grid spacing in the y-direction

Returns *(X, Y) ndarray* – The horizontal convergence

See also:

v_vorticity(), *convergence_vorticity()*

metpy.calc.kinematics.**convergence_vorticity**(*u*, *v*, *dx*, *dy*)
Calculate the horizontal convergence and vertical vorticity of the horizontal wind.

The grid must have a constant spacing in each direction.

Parameters

- **u** (*(X, Y) ndarray*) – x component of the wind
- **v** (*(X, Y) ndarray*) – y component of the wind
- **dx** (*float*) – The grid spacing in the x-direction
- **dy** (*float*) – The grid spacing in the y-direction

Returns **convergence, vorticity** (*tuple of (X, Y) ndarrays*) – The horizontal convergence and vertical vorticity, respectively

See also:

v_vorticity(), *h_convergence()*

Notes

This is a convenience function that will do less work than calculating the horizontal convergence and vertical vorticity separately.

metpy.calc.kinematics.**advection**(*scalar*, *wind*, *deltas*)
Calculate the advection of a scalar field by the wind.

The order of the dimensions of the arrays must match the order in which the wind components are given. For example, if the winds are given [u, v], then the scalar and wind arrays must be indexed as x,y (which puts x as the rows, not columns).

Parameters

- **scalar** (*N-dimensional array*) – Array (with N-dimensions) with the quantity to be advected.
- **wind** (*sequence of arrays*) – Length N sequence of N-dimensional arrays. Represents the flow, with a component of the wind in each dimension. For example, for horizontal advection, this could be a list: [u, v], where u and v are each a 2-dimensional array.
- **deltas** (*sequence*) – A (length N) sequence containing the grid spacing in each dimension.

Returns *N-dimensional array* – An N-dimensional array containing the advection at all grid points.

metpy.calc.kinematics.**geostrophic_wind**(*heights*, *f*, *dx*, *dy*)
Calculate the geostrophic wind given from the heights or geopotential.

Parameters

- **heights** (*(x, y) ndarray*) – The height field, given with leading dimensions of x by y. There can be trailing dimensions on the array.
- **f** (*array_like*) – The coriolis parameter. This can be a scalar to be applied everywhere or an array of values.
- **dx** (*scalar*) – The grid spacing in the x-direction
- **dy** (*scalar*) – The grid spacing in the y-direction

Returns *A 2-item tuple of arrays* – A tuple of the u-component and v-component of the geostrophic wind.

Thermodynamic Calculations

metpy.calc.thermo.**potential_temperature**(*pressure*, *temperature*)
Calculate the potential temperature.

Uses the Poisson equation to calculation the potential temperature given *pressure* and *temperature*.

Parameters

- **pressure** (*pint.Quantity*) – The total atmospheric pressure
- **temperature** (*pint.Quantity*) – The temperature

Returns *pint.Quantity* – The potential temperature corresponding to the the temperature and pressure.

See also:

dry_lapse()

Notes

Formula:

$$\Theta = T(P_0/P)^\kappa$$

Examples

```
>>> from metpy.units import units
>>> metpy.calc.potential_temperature(800. * units.mbar, 273. * units.kelvin)
290.9814150577374
```

metpy.calc.thermo.**dry_lapse**(*pressure*, *temperature*)

Calculate the temperature at a level assuming only dry processes operating from the starting point.

This function lifts a parcel starting at *temperature*, conserving potential temperature. The starting pressure should be the first item in the *pressure* array.

Parameters

- **pressure** (*pint.Quantity*) – The atmospheric pressure level(s) of interest
- **temperature** (*pint.Quantity*) – The starting temperature

Returns *pint.Quantity* – The resulting parcel temperature at levels given by *pressure*

See also:

moist_lapse() Calculate parcel temperature assuming liquid saturation processes

parcel_profile() Calculate complete parcel profile

potential_temperature()

metpy.calc.thermo.**moist_lapse**(*pressure*, *temperature*)

Calculate the temperature at a level assuming liquid saturation processes operating from the starting point.

This function lifts a parcel starting at *temperature*. The starting pressure should be the first item in the *pressure* array. Essentially, this function is calculating moist pseudo-adiabats.

Parameters

- **pressure** (*pint.Quantity*) – The atmospheric pressure level(s) of interest
- **temperature** (*pint.Quantity*) – The starting temperature

Returns *pint.Quantity* – The temperature corresponding to the the starting temperature and pressure levels.

See also:

dry_lapse() Calculate parcel temperature assuming dry adiabatic processes

parcel_profile() Calculate complete parcel profile

Notes

This function is implemented by integrating the following differential equation:

$$\frac{dT}{dP} = \frac{1}{P}\frac{R_d T + L_v r_s}{C_{pd} + \frac{L_v^2 r_s \epsilon}{R_d T^2}}$$

This equation comes from [1].

References

metpy.calc.thermo.**lcl**(*pressure*, *temperature*, *dewpt*, *max_iters=50*, *eps=0.01*)

Calculate the lifted condensation level (LCL) using from the starting point.

The starting state for the parcel is defined by *temperature*, *dewpt*, and *pressure*.

Parameters

- **pressure** (*pint.Quantity*) – The starting atmospheric pressure
- **temperature** (*pint.Quantity*) – The starting temperature
- **dewpt** (*pint.Quantity*) – The starting dew point

Returns *pint.Quantity* – The LCL

Other Parameters

- **max_iters** (*int, optional*) – The maximum number of iterations to use in calculation, defaults to 50.
- **eps** (*float, optional*) – The desired absolute error in the calculated value, defaults to 1e-2.

See also:

parcel_profile()

Notes

This function is implemented using an iterative approach to solve for the LCL. The basic algorithm is: 1. Find the dew point from the LCL pressure and starting mixing ratio 2. Find the LCL pressure from the starting temperature and dewpoint 3. Iterate until convergence

The function is guaranteed to finish by virtue of the *maxIters* counter.

metpy.calc.thermo.**lfc**(*pressure*, *temperature*, *dewpt*)

Calculate the level of free convection (LFC) by finding the first intersection of the ideal parcel path and the measured parcel temperature.

Parameters

- **pressure** (*pint.Quantity*) – The atmospheric pressure
- **temperature** (*pint.Quantity*) – The temperature at the levels given by *pressure*
- **dewpt** (*pint.Quantity*) – The dew point at the levels given by *pressure*

Returns *pint.Quantity* – The LFC

[1] Bakhshaii, A. and R. Stull, 2013: Saturated Pseudoadiabats–A Noniterative Approximation. J. Appl. Meteor. Clim., 52, 5-15.

See also:

parcel_profile()

metpy.calc.thermo.**parcel_profile**(*pressure*, *temperature*, *dewpt*)
Calculate the profile a parcel takes through the atmosphere, lifting from the starting point.

The parcel starts at *temperature*, and *dewpt*, lifted up dry adiabatically to the LCL, and then moist adiabatically from there. *pressure* specifies the pressure levels for the profile.

Parameters

- **pressure** (*pint.Quantity*) – The atmospheric pressure level(s) of interest. The first entry should be the starting point pressure.
- **temperature** (*pint.Quantity*) – The starting temperature
- **dewpt** (*pint.Quantity*) – The starting dew point

Returns *pint.Quantity* – The parcel temperatures at the specified pressure levels.

See also:

lcl(), *moist_lapse()*, *dry_lapse()*

metpy.calc.thermo.**vapor_pressure**(*pressure*, *mixing*)
Calculate water vapor (partial) pressure

Given total *pressure* and water vapor *mixing* ratio, calculates the partial pressure of water vapor.

Parameters

- **pressure** (*pint.Quantity*) – total atmospheric pressure
- **mixing** (*pint.Quantity*) – dimensionless mass mixing ratio

Returns *pint.Quantity* – The ambient water vapor (partial) pressure in the same units as *pressure*.

See also:

saturation_vapor_pressure(), *dewpoint()*

metpy.calc.thermo.**saturation_vapor_pressure**(*temperature*)
Calculate the saturation water vapor (partial) pressure

Parameters **temperature** (*pint.Quantity*) – The temperature

Returns *pint.Quantity* – The saturation water vapor (partial) pressure

See also:

vapor_pressure(), *dewpoint()*

Notes

Instead of temperature, dewpoint may be used in order to calculate the actual (ambient) water vapor (partial) pressure.

The formula used is that from Bolton 1980 [2] for T in degrees Celsius:

$$6.112e^{\frac{17.67T}{T+243.5}}$$

References

metpy.calc.thermo.**dewpoint_rh**(*temperature*, *rh*)

Calculate the ambient dewpoint given air temperature and relative humidity.

Parameters

- **temperature** (*pint.Quantity*) – Air temperature
- **rh** (*pint.Quantity*) – Relative humidity expressed as a ratio in the range [0, 1]

Returns *pint.Quantity* – The dew point temperature

See also:

dewpoint(), *saturation_vapor_pressure()*

metpy.calc.thermo.**dewpoint**(*e*)

Calculate the ambient dewpoint given the vapor pressure.

Parameters **e** (*pint.Quantity*) – Water vapor partial pressure

Returns *pint.Quantity* – Dew point temperature

See also:

dewpoint_rh(), *saturation_vapor_pressure()*, *vapor_pressure()*

Notes

This function inverts the Bolton 1980 [3] formula for saturation vapor pressure to instead calculate the temperature. This yield the following formula for dewpoint in degrees Celsius:

$$T = \frac{243.5 log(e/6.112)}{17.67 - log(e/6.112)}$$

References

metpy.calc.thermo.**mixing_ratio**(*part_press*, *tot_press*)

Calculates the mixing ratio of gas given its partial pressure and the total pressure of the air.

There are no required units for the input arrays, other than that they have the same units.

Parameters

- **part_press** (*pint.Quantity*) – Partial pressure of the constituent gas
- **tot_press** (*pint.Quantity*) – Total air pressure

Returns *pint.Quantity* – The (mass) mixing ratio, dimensionless (e.g. Kg/Kg or g/g)

See also:

vapor_pressure()

metpy.calc.thermo.**saturation_mixing_ratio**(*tot_press*, *temperature*)

Calculates the saturation mixing ratio given total pressure and the temperature.

The implementation uses the formula outlined in [4]

Parameters

- **tot_press** (*pint.Quantity*) – Total atmospheric pressure

- **temperature** (*pint.Quantity*) – The temperature

Returns *pint.Quantity* – The saturation mixing ratio, dimensionless

References

metpy.calc.thermo.**equivalent_potential_temperature**(*pressure*, *temperature*)
Calculates equivalent potential temperature given an air parcel's pressure and temperature.

The implementation uses the formula outlined in [5]

Parameters

- **pressure** (*pint.Quantity*) – Total atmospheric pressure
- **temperature** (*pint.Quantity*) – The temperature

Returns *pint.Quantity* – The corresponding equivalent potential temperature of the parcel

Notes

$$\Theta_e = \Theta e^{\frac{L_v r_s}{C_{pd} T}}$$

References

Turbulence Time Series Calculations

metpy.calc.turbulence This module contains calculations related to turbulence and time series perturbations.

metpy.calc.turbulence.**get_perturbation**(*ts*, *axis=-1*)
Compute the perturbation from the mean of a time series.

Parameters **ts** (*array_like*) – The time series from which you wish to find the perturbation time series (perturbation from the mean).

Returns *array_like* – The perturbation time series.

Other Parameters **axis** (*int*) – The index of the time axis. Default is -1

Notes

The perturbation time series produced by this function is defined as the perturbations about the mean:

$$x(t)' = x(t) - \overline{x(t)}$$

metpy.calc.turbulence.**tke**(*u*, *v*, *w*, *perturbation=False*, *axis=-1*)
Compute the turbulence kinetic energy (e) from the time series of the velocity components.

Parameters

- **u** (*array_like*) – The wind component along the x-axis
- **v** (*array_like*) – The wind component along the y-axis

- **w** (*array_like*) – The wind component along the z-axis
- **perturbation** (*{False, True}, optional*) – True if the *u*, *v*, and *w* components of wind speed supplied to the function are perturbation velocities. If False, perturbation velocities will be calculated by removing the mean value from each component.

Returns *array_like* – The corresponding turbulence kinetic energy value

Other Parameters **axis** (*int*) – The index of the time axis. Default is -1

See also:

get_perturbation() Used to compute perturbations if *perturbation* is False.

Notes

Turbulence Kinetic Energy is computed as:

$$e = 0.5\sqrt{\overline{u'^2} + \overline{v'^2} + \overline{w'^2}},$$

where the velocity components

$$u', v', u'$$

are perturbation velocities. For more information on the subject, please see [1].

References

metpy.calc.turbulence.**kinematic_flux**(*vel*, *b*, *perturbation=False*, *axis=-1*)

Compute the kinematic flux from the time series of two variables *vel* and b. Note that to be a kinematic flux, at least one variable must be a component of velocity.

Parameters

- **vel** (*array_like*) – A component of velocity
- **b** (*array_like*) – May be a component of velocity or a scalar variable (e.g. Temperature)
- **perturbation** (*{False, True}, optional*) – True if the *vel* and *b* variables are perturbations. If False, perturbations will be calculated by removing the mean value from each variable.

Returns *array_like* – The corresponding kinematic flux

Other Parameters **axis** (*int*) – The index of the time axis. Default is -1

Notes

A kinematic flux is computed as

$$\overline{u's'}$$

where at the prime notation denotes perturbation variables, and at least one variable is perturbation velocity. For example, the vertical kinematic momentum flux (two velocity components):

$$\overline{u'w'}$$

[1] Garratt, J.R., 1994: The Atmospheric Boundary Layer. Cambridge University Press, 316 pp.

or the the vertical kinematic heat flux (one velocity component, and one scalar):

$$\overline{w'T'}$$

If perturbation variables are passed into this function (i.e. *perturbation* is True), the kinematic flux is computed using the equation above.

However, the equation above can be rewritten as

$$\overline{us} - \overline{u}\,\overline{s}$$

which is computationally more efficient. This is how the kinematic flux is computed in this function if *perturbation* is False.

For more information on the subject, please see [2].

References

metpy.calc.turbulence.**friction_velocity**(*u*, *w*, *v=None*, *perturbation=False*, *axis=-1*)

Compute the friction velocity from the time series of the x, z, and optionally y, velocity components.

Parameters

- **u** (*array_like*) – The wind component along the x-axis
- **w** (*array_like*) – The wind component along the z-axis
- **v** (*array_like, optional*) – The wind component along the y-axis.
- **perturbation** (*{False, True}, optional*) – True if the *u*, *w*, and *v* components of wind speed supplied to the function are perturbation velocities. If False, perturbation velocities will be calculated by removing the mean value from each component.

Returns *array_like* – The corresponding friction velocity

Other Parameters **axis** (*int*) – The index of the time axis. Default is -1

See also:

kinematic_flux() Used to compute the x-component and y-component vertical kinematic momentum flux(es) used in the computation of the friction velocity.

Notes

The Friction Velocity is computed as:

$$u_* = \sqrt[4]{\left(\overline{u'w'}\right)^2 + \left(\overline{v'w'}\right)^2},$$

where :math: overline{u^{prime}w^{prime}} and :math: overline{v^{prime}w^{prime}} are the x-component and y-components of the vertical kinematic momentum flux, respectively. If the optional v component of velocity is not supplied to the function, the computation of the friction velocity is reduced to

$$u_* = \sqrt[4]{\left(\overline{u'w'}\right)^2}$$

For more information on the subject, please see [3].

[2] Garratt, J.R., 1994: The Atmospheric Boundary Layer. Cambridge University Press, 316 pp.
[3] Garratt, J.R., 1994: The Atmospheric Boundary Layer. Cambridge University Press, 316 pp.

References

1.4.5 metpy.plots

Skew-T

class metpy.plots.skewt.**SkewT**(*fig=None*, *rotation=30*, *subplot=(1, 1, 1)*)
Make Skew-T log-P plots of data

This class simplifies the process of creating Skew-T log-P plots in using matplotlib. It handles requesting the appropriate skewed projection, and provides simplified wrappers to make it easy to plot data, add wind barbs, and add other lines to the plots (e.g. dry adiabats)

ax
matplotlib.axes.Axes

The underlying Axes instance, which can be used for calling additional plot functions (e.g. *axvline*)

Creates SkewT - logP plots.

Parameters

- **fig** (*matplotlib.figure.Figure, optional*) – Source figure to use for plotting. If none is given, a new matplotlib.figure.Figure instance will be created.
- **rotation** (*float or int, optional*) – Controls the rotation of temperature relative to horizontal. Given in degrees counterclockwise from x-axis. Defaults to 30 degrees.
- **subplot** (tuple[int, int, int] or *matplotlib.gridspec.SubplotSpec* instance, optional) – Controls the size/position of the created subplot. This allows creating the skewT as part of a collection of subplots. If subplot is a tuple, it should conform to the specification used for matplotlib.figure.Figure.add_subplot(). The matplotlib.gridspec.SubplotSpec can be created by using matplotlib.gridspec.GridSpec.

plot(*p*, *t*, **args*, ***kwargs*)
Plot data.

Simple wrapper around plot so that pressure is the first (independent) input. This is essentially a wrapper around *semilogy*. It also sets some appropriate ticking and plot ranges.

Parameters

- **p** (*array_like*) – pressure values
- **t** (*array_like*) – temperature values, can also be used for things like dew point
- **args** – Other positional arguments to pass to semilogy()
- **kwargs** – Other keyword arguments to pass to semilogy()

Returns *list[matplotlib.lines.Line2D]* – lines plotted

See also:

matplotlib.pyplot.semilogy()

plot_barbs(*p*, *u*, *v*, *xloc=1.0*, *x_clip_radius=0.08*, *y_clip_radius=0.08*, ***kwargs*)
Plot wind barbs.

Adds wind barbs to the skew-T plot. This is a wrapper around the *barbs* command that adds to appropriate transform to place the barbs in a vertical line, located as a function of pressure.

Parameters

- **p** (*array_like*) – pressure values
- **u** (*array_like*) – U (East-West) component of wind
- **v** (*array_like*) – V (North-South) component of wind
- **xloc** (*float, optional*) – Position for the barbs, in normalized axes coordinates, where 0.0 denotes far left and 1.0 denotes far right. Defaults to far right.
- **x_clip_radius** (*float, optional*) – Space, in normalized axes coordinates, to leave before clipping wind barbs in the x-direction. Defaults to 0.08.
- **y_clip_radius** (*float, optional*) – Space, in normalized axes coordinates, to leave above/below plot before clipping wind barbs in the y-direction. Defaults to 0.08.
- **kwargs** – Other keyword arguments to pass to `barbs()`

Returns *matplotlib.quiver.Barbs* – instance created

See also:

`matplotlib.pyplot.barbs()`

plot_dry_adiabats(*t0=None*, *p=None*, ***kwargs*)
Plot dry adiabats.

Adds dry adiabats (lines of constant potential temperature) to the plot. The default style of these lines is dashed red lines with an alpha value of 0.5. These can be overridden using keyword arguments.

Parameters

- **t0** (*array_like, optional*) – Starting temperature values in Kelvin. If none are given, they will be generated using the current temperature range at the bottom of the plot.
- **p** (*array_like, optional*) – Pressure values to be included in the dry adiabats. If not specified, they will be linearly distributed across the current plotted pressure range.
- **kwargs** – Other keyword arguments to pass to `matplotlib.collections.LineCollection`

Returns *matplotlib.collections.LineCollection* – instance created

See also:

dry_lapse() *plot_moist_adiabats()* `matplotlib.collections.LineCollection`

plot_mixing_lines(*w=None*, *p=None*, ***kwargs*)
Plot lines of constant mixing ratio.

Adds lines of constant mixing ratio (isohumes) to the plot. The default style of these lines is dashed green lines with an alpha value of 0.8. These can be overridden using keyword arguments.

Parameters

- **w** (*array_like, optional*) – Unitless mixing ratio values to plot. If none are given, default values are used.
- **p** (*array_like, optional*) – Pressure values to be included in the isohumes. If not specified, they will be linearly distributed across the current plotted pressure range up to 600 mb.
- **kwargs** – Other keyword arguments to pass to `matplotlib.collections.LineCollection`

Returns *matplotlib.collections.LineCollection* – instance created

See also:

matplotlib.collections.LineCollection

plot_moist_adiabats(*t0=None*, *p=None*, ***kwargs*)
Plot moist adiabats.

Adds saturated pseudo-adiabats (lines of constant equivalent potential temperature) to the plot. The default style of these lines is dashed blue lines with an alpha value of 0.5. These can be overridden using keyword arguments.

Parameters

- **t0** (*array_like, optional*) – Starting temperature values in Kelvin. If none are given, they will be generated using the current temperature range at the bottom of the plot.
- **p** (*array_like, optional*) – Pressure values to be included in the moist adiabats. If not specified, they will be linearly distributed across the current plotted pressure range.
- **kwargs** – Other keyword arguments to pass to matplotlib.collections.LineCollection

Returns *matplotlib.collections.LineCollection* – instance created

See also:

moist_lapse() *plot_dry_adiabats()* matplotlib.collections.LineCollection

class metpy.plots.skewt.**Hodograph**(*ax=None*, *component_range=80*)
Make a hodograph of wind data–plots the u and v components of the wind along the x and y axes, respectively.

This class simplifies the process of creating a hodograph using matplotlib. It provides helpers for creating a circular grid and for plotting the wind as a line colored by another value (such as wind speed).

ax
matplotlib.axes.Axes

The underlying Axes instance used for all plotting

Create a Hodograph instance.

Parameters

- **ax** (*matplotlib.axes.Axes*, optional) – The *Axes* instance used for plotting
- **component_range** (*value*) – The maximum range of the plot. Used to set plot bounds and control the maximum number of grid rings needed.

add_grid(*increment=10.0*, ***kwargs*)
Add grid lines to hodograph.

Creates lines for the x- and y-axes, as well as circles denoting wind speed values.

Parameters

- **increment** (*value, optional*) – The value increment between rings
- **kwargs** – Other kwargs to control appearance of lines

See also:

matplotlib.patches.Circle, matplotlib.axes.Axes.axhline(), matplotlib.axes.Axes.axvline()

plot(*u*, *v*, ***kwargs*)
Plot u, v data.

Plots the wind data on the hodograph.

Parameters

- **u** (*array_like*) – u-component of wind
- **v** (*array_like*) – v-component of wind
- **kwargs** – Other keyword arguments to pass to `matplotlib.axes.Axes.plot()`

Returns *list[matplotlib.lines.Line2D]* – lines plotted

See also:

Hodograph.plot_colormapped()

plot_colormapped(*u*, *v*, *c*, ***kwargs*)

Plot u, v data, with line colored based on a third set of data.

Plots the wind data on the hodograph, but

Simple wrapper around plot so that pressure is the first (independent) input. This is essentially a wrapper around *semilogy*. It also sets some appropriate ticking and plot ranges.

Parameters

- **u** (*array_like*) – u-component of wind
- **v** (*array_like*) – v-component of wind
- **c** (*array_like*) – data to use for colormapping
- **kwargs** – Other keyword arguments to pass to `matplotlib.collections.LineCollection`

Returns *matplotlib.collections.LineCollection* – instance created

See also:

Hodograph.plot()

Station Plots

class metpy.plots.station_plot.**StationPlot**(*ax*, *x*, *y*, *fontsize=10*, *spacing=None*, *transform=None*)

Make a standard meteorological station plot.

Plots values, symbols, or text spaced around a central location. Can also plot wind barbs as the center of the location.

Initialize the StationPlot with items that do not change.

This sets up the axes and station locations. The *fontsize* and *spacing* are also specified here to ensure that they are consistent between individual station elements.

Parameters

- **ax** (*matplotlib.axes.Axes*) – The `Axes` for plotting
- **x** (*array_like*) – The x location of the stations in the plot
- **y** (*array_like*) – The y location of the stations in the plot
- **fontsize** (*int*) – The fontsize to use for drawing text
- **spacing** (*int*) – The spacing, in points, that corresponds to a single increment between station plot elements.

- **transform** (*matplotlib.transforms.Transform (or compatible)*) – The default transform to apply to the x and y positions when plotting.

location_names = {'C': (0, 0), 'E': (1, 0), 'SW': (-1, -1), 'NE': (1, 1), 'N': (0, 1), 'S': (0, -1), 'W': (-1, 0), 'SE': (1, -1), 'N

plot_barb(*u*, *v*, ***kwargs*)

At the center of the station model plot wind barbs.

Additional keyword arguments given will be passed onto matplotlib's `barbs()` function; this is useful for specifying things like color or line width.

Parameters

- **u** (*array-like*) – The data to use for the u-component of the barbs.
- **v** (*array-like*) – The data to use for the v-component of the barbs.
- **kwargs** – Additional keyword arguments to pass to matplotlib's `barbs()` function.

See also:

plot_parameter(), *plot_symbol()*, *plot_text()*

plot_parameter(*location*, *parameter*, *formatter='.0f'*, ***kwargs*)

At the specified location in the station model plot a set of values.

This specifies that at the offset *location*, the data in *parameter* should be plotted. The conversion of the data values to a string is controlled by *formatter*.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

If something has already been plotted at this location, it will be replaced.

Parameters

- **location** (*str or tuple[float, float]*) – The offset (relative to center) to plot this parameter. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions; increments are multiplied by *spacing* to give offsets in x and y relative to the center.
- **parameter** (*array_like*) – The numeric values that should be plotted
- **formatter** (*str or callable, optional*) – How to format the data as a string for plotting. If a string, it should be compatible with the `format()` builtin. If a callable, this should take a value and return a string. Defaults to '0.f'.
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

plot_barb(), *plot_symbol()*, *plot_text()*

plot_symbol(*location*, *codes*, *symbol_mapper*, ***kwargs*)

At the specified location in the station model plot a set of symbols.

This specifies that at the offset *location*, the data in *codes* should be converted to unicode characters (for our `wx_symbol_font`) using *symbol_mapper*, and plotted.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

If something has already been plotted at this location, it will be replaced.

Parameters

- **location** (*str or tuple[float, float]*) – The offset (relative to center) to plot this parameter. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions; increments are multiplied by *spacing* to give offsets in x and y relative to the center.
- **codes** (*array_like*) – The numeric values that should be converted to unicode characters for plotting.
- **symbol_mapper** (*callable*) – Controls converting data values to unicode code points for the `wx_symbol_font` font. This should take a value and return a single unicode character. See `metpy.plots.wx_symbols` for included mappers.
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

plot_barb(), *plot_parameter()*, *plot_text()*

plot_text (*location*, *text*, ***kwargs*)

At the specified location in the station model plot a collection of text.

This specifies that at the offset *location*, the strings in *text* should be plotted.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

If something has already been plotted at this location, it will be replaced.

Parameters

- **location** (*str or tuple[float, float]*) – The offset (relative to center) to plot this parameter. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions; increments are multiplied by *spacing* to give offsets in x and y relative to the center.
- **text** (*list (or array) of strings*) – The strings that should be plotted
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

plot_barb(), *plot_parameter()*, *plot_symbol()*

class `metpy.plots.station_plot.`**StationPlotLayout**

Encapsulates a standard layout for plotting using *StationPlot*.

This class keeps a collection of offsets, plot formats, etc. for a parameter based on its name. This then allows a dictionary of data (or any object that allows looking up of arrays based on a name) to be passed to *plot()* to plot the data all at once.

See also:

StationPlot

class PlotTypes

Different plotting types for the layout.

Controls how items are displayed (e.g. converting values to symbols).

barb = <PlotTypes.barb: 4>

symbol = <PlotTypes.symbol: 2>

text = <PlotTypes.text: 3>

StationPlotLayout.**__repr__**()
Return string representation of layout

StationPlotLayout.**add_barb**(*u_name*, *v_name*, *units=None*, ***kwargs*)
Add a wind barb to the center of the station layout.

This specifies that u- and v-component data should be pulled from the data container using the keys *u_name* and *v_name*, respectively, and plotted as a wind barb at the center of the station plot. If *units* are given, both components will be converted to these units.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or line width.

Parameters

- **u_name** (*str*) – The name of the parameter for the u-component for *barbs*, which is used as a key to pull data out of the data container passed to *plot()*.
- **v_name** (*str*) – The name of the parameter for the v-component for *barbs*, which is used as a key to pull data out of the data container passed to *plot()*.
- **units** (*pint-compatible unit, optional*) – The units to use for plotting. Data will be converted to this unit before conversion to a string. If not specified, no conversion is done.
- **kwargs** – Additional keyword arguments to use for matplotlib's barbs() function.

See also:

add_symbol(), *add_text()*, *add_value()*

StationPlotLayout.**add_symbol**(*location*, *name*, *symbol_mapper*, ***kwargs*)
Add a symbol to the station layout.

This specifies that at the offset *location*, data should be pulled from the data container using the key *name* and plotted. Data values will converted to glyphs appropriate for MetPy's symbol font using the callable *symbol_mapper*.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

Parameters

- **location** (*str or tuple[float, float]*) – The offset (relative to center) to plot this value. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions.
- **name** (*str*) – The name of the parameter, which is used as a key to pull data out of the data container passed to *plot()*.
- **symbol_mapper** (*callable*) – Controls converting data values to unicode code points for the wx_symbol_font font. This should take a value and return a single unicode character. See metpy.plots.wx_symbols for included mappers.
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

add_barb(), *add_text()*, *add_value()*

StationPlotLayout.**add_text**(*location*, *name*, ***kwargs*)
Add a text field to the station layout.

This specifies that at the offset *location*, data should be pulled from the data container using the key *name* and plotted directly as text with no conversion applied.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

Parameters

- **location** (*str or tuple(float, float)*) – The offset (relative to center) to plot this value. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions.
- **name** (*str*) – The name of the parameter, which is used as a key to pull data out of the data container passed to *plot()*.
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

add_barb(), *add_symbol()*, *add_value()*

StationPlotLayout.**add_value**(*location*, *name*, *fmt='.0f'*, *units=None*, ***kwargs*)

Add a numeric value to the station layout.

This specifies that at the offset *location*, data should be pulled from the data container using the key *name* and plotted. The conversion of the data values to a string is controlled by *fmt*. The units required for plotting can also be passed in using *units*, which will cause the data to be converted before plotting.

Additional keyword arguments given will be passed onto the actual plotting code; this is useful for specifying things like color or font properties.

Parameters

- **location** (*str or tuple[float, float]*) – The offset (relative to center) to plot this value. If str, should be one of 'C', 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', or 'NW'. Otherwise, should be a tuple specifying the number of increments in the x and y directions.
- **name** (*str*) – The name of the parameter, which is used as a key to pull data out of the data container passed to *plot()*.
- **fmt** (*str or callable, optional*) – How to format the data as a string for plotting. If a string, it should be compatible with the `format()` builtin. If a callable, this should take a value and return a string. Defaults to '0.f'.
- **units** (*pint-compatible unit, optional*) – The units to use for plotting. Data will be converted to this unit before conversion to a string. If not specified, no conversion is done.
- **kwargs** – Additional keyword arguments to use for matplotlib's plotting functions.

See also:

add_barb(), *add_symbol()*, *add_text()*

StationPlotLayout.**names**()

Get the list of names used by the layout.

Returns *list[str]* – the list of names of variables used by the layout

StationPlotLayout.**plot**(*plotter*, *data_dict*)

Plot a collection of data using this layout for a station plot.

This function iterates through the entire specified layout, pulling the fields named in the layout from *data_dict* and plotting them using *plotter* as specified in the layout. Fields present in the layout, but not in *data_dict*, are ignored.

Parameters

- **plotter** (*StationPlot*) – *StationPlot* to use to plot the data. This controls the axes, spacing, station locations, etc.
- **data_dict** (*dict[str, array-like]*) – Data container that maps a name to an array of data. Data from this object will be used to fill out the station plot.

metpy.plots.station_plot.**nws_layout = {barb: (barb, ('eastward_wind', 'northward_wind'), ...), (-2, 0): (value, visi**
Full NWS station plot layout

metpy.plots.station_plot.**simple_layout = {C: (symbol, cloud_coverage, ...), NE: (value, air_pressure_at_sea_level**

Desc Simple station plot layout

Colortables

metpy.plots.ctables

class metpy.plots.ctables.**ColortableRegistry**
Manages the collection of colortables.

Provides access to colortables, read collections of files, and generates matplotlib's Normalize instances to go with the colortable.

add_colortable(*fobj*, *name*)
Add a colortable from a file to the registry

Parameters

- **fobj** (*file-like object*) – The file to read the colortable from
- **name** (*str*) – The name under which the colortable will be stored

get_colortable(*name*)
Get a colortable from the registry

Parameters **name** (*str*) – The name under which the colortable will be stored

Returns *matplotlib.colors.ListedColormap* – The colortable corresponding to *name*

get_with_boundaries(*name*, *boundaries*)
Get a colortable from the registry with a corresponding norm.

Builds a *matplotlib.colors.BoundaryNorm* using *boundaries*.

Parameters

- **name** (*str*) – The name under which the colortable will be stored
- **boundaries** (*array_like*) – The list of boundaries for the norm

Returns *matplotlib.colors.BoundaryNorm*, *matplotlib.colors.ListedColormap* – The boundary norm based on *boundaries*, and the colortable itself.

get_with_steps(*name*, *start*, *step*)
Get a colortable from the registry with a corresponding norm.

Builds a *matplotlib.colors.BoundaryNorm* using *start*, *step*, and the number of colors, based on the colortable obtained from *name*.

Parameters

- **name** (*str*) – The name under which the colortable will be stored
- **start** (*float*) – The starting boundary
- **step** (*float*) – The step between boundaries

Returns *matplotlib.colors.BoundaryNorm*, *matplotlib.colors.ListedColormap* – The boundary norm based on *start* and *step* with the number of colors from the number of entries matching the colortable, and the colortable itself.

scan_dir(*path*)
Scan a directory on disk for colortable files and add them to the registry

Parameters **path** (*str*) – The path to the directory with the colortables

scan_resource(*pkg*, *path*)
Scan a resource directory for colortable files and add them to the registry

Parameters

- **pkg** (*str*) – The package containing the resource directory
- **path** (*str*) – The path to the directory with the colortables

metpy.plots.ctables.**convert_gempak_table**(*infile*, *outfile*)
Convert a GEMPAK colortable to one MetPy can read.

Reads lines from a GEMPAK-style color table file, and writes them to another file in a format that MetPy can parse.

Parameters

- **infile** (*file-like object*) – The file-like object to read from
- **outfile** (*file-like object*) – The file-like object to write to

metpy.plots.ctables.**read_colortable**(*fobj*)
Read colortable information from a file.

Reads a colortable, which consists of one color per line of the file, where a color can be one of: a tuple of 3 floats, a string with a HTML color name, or a string with a HTML hex color.

Parameters **fobj** (*a file-like object*) – A file-like object to read the colors from

Returns *List of tuples* – A list of the RGB color values, where each RGB color is a tuple of 3 floats in the range of [0, 1].

- modindex
- genindex

1.5 Developer's Guide

1.5.1 Requirements

- pytest
- flake8
- sphinx >= 1.3
- sphinx-rtd-theme >= 0.1.7
- nbconvert>=4.1

Conda

Settings up a development environment in MetPy is as easy as (from the base of the repository):

```
conda env create
conda develop -n devel .
```

The `environment.yml` contains all of the configuration needed to easily set up the environment, called `devel`. The second line sets up conda to run directly out of the git repository.

1.5.2 Making Changes

The changes to the MetPy source (and documentation) should be made via GitHub pull requests against `master`, even for those with administration rights. While it's tempting to make changes directly to `master` and push them up, it is better to make a pull request so that others can give feedback. If nothing else, this gives a chance for the automated tests to run on the PR. This can eliminate "brown paper bag" moments with buggy commits on the master branch.

During the Pull Request process, before the final merge, it's a good idea to rebase the branch and squash together smaller commits. It's not necessary to flatten the entire branch, but it can be nice to eliminate small fixes and get the merge down to logically arranged commit. This can also be used to hide sins from history–this is the only chance, since once it hits `master`, it's there forever!

1.5.3 Versioning

To manage identifying the version of the code, MetPy relies upon versioneer. `versioneer` takes the current version of the source from git tags and any additional commits. For development, the version will have a string like `0.1.1+76.g136e37b.dirty`, which comes from `git describe`. This version means that the current code is 76 commits past the 0.1.1 tag, on git hash `136e37b`, with local changes on top (indicated by `dirty`). For a release, or non-git repo source dir, the version will just come from the most recent tag (i.e. `v0.1.1`).

To make a new version, simply add a new tag with a name like `vMajor.Minor.Bugfix` and push to GitHub. Github will add a new release with a source archive.zip file. Running

```
python setup.py sdist
```

will build a new source distribution with the appropriately generated version file as well. This will also create a new stable set of documentation.

`versioneer` is installed in the base of the repository. To update, install the latest copy using `pip install versioneer`. Then recreate the `_version.py` file using:

```
python setup.py versioneer
```

1.5.4 Testing

Unit tests are the lifeblood of the project, as it ensures that we can continue to add and change the code and stay confident that things have not broken. Running the tests requires `pytest`, which is easily available through `conda` or `pip`. Running the tests can be done via either:

```
python setup.py test
```

or

```
py.test
```

Using `py.test` also gives you the option of passing a path to the directory with tests to run, which can speed running only the tests of interest when doing development. For instance, to only run the tests in the `metpy/calc` directory, use:

```
py.test metpy/calc
```

Some tests (for matplotlib plotting code) are done through an image comparison, using the pytest-mpl plugin. To run these tests, use:

```
py.test --mpl
```

When adding new image comparison tests, start by creating the baseline images for the tests:

```
py.test --mpl-generate-path=baseline
```

That command runs the tests and saves the images in the `baseline` directory. Once the images are reviewed and determined to be correct, they should be moved to a `baseline` directory in the same directory as the test script (e.g. `metpy/plots/tests`) For more information, see the docs for mpl-test.

1.5.5 Code Style

MetPy uses the Python code style outlined in PEP8. For better or worse, this is what the majority of the Python world uses. The one deviation is that line length limit is 95 characters. 80 is a good target, but some times longer lines are needed.

While the authors are no fans of blind adherence to style and so-called project "clean-ups" that go through and correct code style, MetPy has adopted this style from the outset. Therefore, it makes sense to enforce this style as code is added to keep everything clean and uniform. To this end, part of the automated testing for MetPy checks style. To check style locally within the source directory you can use the `flake8` tool. Running it from the root of the source directory is as easy as:

```
flake8 metpy
```

1.5.6 Documentation

MetPy's documentation is built using sphinx >= 1.3. API documentation is automatically generated from docstrings, written using the NumPy docstring standard. There are also example IPython notebooks in the `examples/notebooks` directory. Using IPython's API, these are automatically converted to restructured text for inclusion in the documentation. The examples can also be converted to standalone scripts using:

```
python setup.py examples
```

The documentation is hosted by Read the Docs. The docs are built automatically from `master` as well as for the tagged versions on github. `master` is used for the `latest` documentation, and the latest tagged version is used for the `stable` documentation. To see what the docs will look like on RTD, you also need to install the `sphinx-rtd-theme` package.

1.5.7 Other Tools

Continuous integration is performed by Travis CI and AppVeyor. Travis runs the unit tests on Linux for all supported versions of Python, as well as runs against the minimum package versions. `flake8` (with the `pep8-naming` and `flake8-quotes` plugins) is also run against the code to check formatting. Travis is also used to build the documentation and to run the examples to ensure they stay working. AppVeyor is a similar service; here the tests and examples are run against Python 2 and 3 for both 32- and 64-bit versions of Windows.

Test coverage is monitored by codecov.io.

The following services are used to track code quality:

- QuantifiedCode
- Codacy
- Code Climate
- Scrutinizer
- Landscape.io

1.5.8 Releasing

To create a new release, go to the GitHub page and make a new release. The tag should be a sensible version number, like v1.0.0. Add a name (can just be the version) and add some release notes on what the big changes are. It's also possible to use loghub to get information on all the issues and PRs that were closed for the relevant milestone. Tagging a new version on GitHub should also update the stable docs on Read the Docs.

PyPI

Once the new release is published on GitHub, this will create the tag, which will trigger new builds on Travis (and AppVeyor, but that's not relevant). When the main test build on Travis (currently Python 3 tests) succeeds, Travis will handle building the source distribution and wheels, and upload them to PyPI.

To build and upload manually (if for some reason it is necessary):

1. Do a pull locally to grab the new tag. This will ensure that `versioneer` will give you the proper version.
2. (optional) Perform a `git clean -f -x -d` from the root of the repository. This will **delete** everything not tracked by git, but will also ensure clean source distribution. `MANIFEST.in` is set to include/exclude mostly correctly, but could miss some things.
3. Run `python setup.py sdist bdist_wheel` (this requires that `wheel` is installed).
4. Upload using `twine`: `twine upload dist/*`, assuming the `dist/` directory contains only files for this release. This upload process will include any changes to the `README` as well as any updated flags from `setup.py`.

Conda

MetPy conda packages are automatically produced and uploaded to Anaconda.org thanks to conda-forge. Once the release is built and uploaded to PyPI, then a Pull Request should be made against the MetPy feedstock, which contains the recipe for building MetPy's conda packages. The Pull Request should:

1. Update the version
2. Update the hash to match that of the new source distribution **uploaded to PyPI**
3. Reset the build number to 0 (if necessary)
4. Update the dependencies (and their versions) as necessary

The Pull Request will test building the packages on all the platforms. Once this succeeds, the Pull Request can be merged, which will trigger the final build and upload of the packages to anaconda.org.

CHAPTER 2

Contact Us

- For questions and discussion about MetPy, join Unidata's python-users mailing list
- The source code is available on GitHub
- Bug reports and feature requests should be directed to the GitHub issue tracker
- MetPy has a Gitter chatroom for more "live" communication
- MetPy can also be found on Twitter

CHAPTER 3

Presentations

- Presentation on MetPy's build infrastructure by Ryan May at SciPy 2016.
- MetPy was included in tools presented at the SSEC/Wisconsin AOS Python Workshop.
- Presentation on MetPy at the 2016 AMS Annual Meeting by Ryan May.
- Ryan May's talk and tutorial on MetPy at the 2015 Unidata Users Workshop.

CHAPTER 4

License

MetPy is available under the terms of the open source BSD 3 Clause license.

CHAPTER 5

Related Projects

- netCDF4-python is the officially blessed Python API for netCDF
- siphon is an API for accessing remote data on THREDDS Data Server

m

Symbols

A

B

C

D

E

F

G

S

T

U

V

W