Basic Coding With SHARPpy ¶

Written by: Greg Blumberg (OU/CIMMS)

This IPython Notebook tutorial is meant to teach the user how to directly interact with the SHARPpy libraries using the Python interpreter. This tutorial will cover reading in files into the the Profile object, plotting the data using Matplotlib, and computing various indices from the data. It is also a reference to the different functions and variables SHARPpy has available to the user.

In order to work with SHARPpy, you need to perform 3 steps before you can begin running routines such as CAPE/CIN on the data.

Step 1: Read in the data to work with.

1.) The Pilger, NE tornado proximity sounding from 19 UTC within the tutorial/ directory is an example of the SPC sounding file format that can be read in by the GUI. Here we'll read it in manually.

In [18]:
```
%matplotlib inline
spc_file = open('fnlsounding_CGS_092401.txt', 'r').read()
```

All of the SHARPpy routines (parcel lifting, composite indices, etc.) reside within the SHARPTAB module.

SHARPTAB contains 6 modules: params, winds, thermo, utils, interp, fire, constants, watch_type

Each module has different functions:

```
interp - interpolates different variables (temperature, dewpoint, wind, etc.) to a
 specified pressure
winds - functions used to compute different wind-related variables (shear, helicit
y, mean winds, storm relative vectors)
thermo - temperature unit conversions, theta-e, theta, wetbulb, lifting functions
utils - wind speed unit conversions, wind speed and direction to u and v conversion
s, QC
params - computation of different parameters, indices, etc. from the Profile object
fire - fire weather indices
```

Step 2: Load in the SHARPTAB module.

In [19]:
```
import sharppy
import sharppy.sharptab.profile as profile
import sharppy.sharptab.interp as interp
import sharppy.sharptab.winds as winds
import sharppy.sharptab.utils as utils
import sharppy.sharptab.params as params
import sharppy.sharptab.thermo as thermo
```

Step 3: Making a Profile object.

Before running any analysis routines on the data, we have to create a Profile object first. A Profile object describes the vertical thermodynamic and kinematic profiles and is the key object that all SHARPpy routines need to run. Any data source can be passed into a Profile object (i.e. radiosonde, RASS, satellite sounding retrievals, etc.) as long as it has these profiles:

- temperature (C)
- dewpoint (C)
- height (meters above mean sea level)
- pressure (millibars)
- wind speed (kts)
- wind direction (degrees)

or (optional)

- zonal wind component U (kts)
- meridional wind component V (kts)

For example, after reading in the data in the example above, a Profile object can be created. Since this file uses the value -9999 to indicate missing values, we need to tell SHARPpy to ignore these values in its calculations by including the missing field to be -9999. In addition, we tell SHARPpy we want to create a default BasicProfile object. Telling SHARPpy to create a "convective" profile object will generate a Profile object with all of the indices computed in the SHARPpy GUI. If you are only wanting to compute a few indices, you probably don't want to do that.

```
In [20]: import numpy as np
         from StringIO import StringIO

         def parseSPC(spc_file):
             ## read in the file
             data = np.array([l.strip() for l in spc_file.split('\n')])
             print(data)

             ## necessary index points
             title_idx = np.where( data == '%TITLE%')[0][0]
             start_idx = np.where( data == '%RAW%' )[0] + 1
             finish_idx = np.where( data == '%END%')[0]

             ## create the plot title
             data_header = data[title_idx + 1].split()
             location = data_header[0]
             time = data_header[1][:11]

             print(data[start_idx[0] : finish_idx[0]])

             ## put it all together for StringIO
             full_data = '\n'.join(data[start_idx[0] : finish_idx[0]][:])
             sound_data = StringIO( full_data )

             ## read the data into arrays
             p, h, T, Td, wdir, wspd = np.genfromtxt( sound_data, delimiter=',', commen
         ts="%", unpack=True )

             return p, h, T, Td, wdir, wspd

         pres, hght, tmpc, dwpc, wdir, wspd = parseSPC(spc_file)

         prof = profile.create_profile(profile='default', pres=pres, hght=hght, tmpc=tm
         pc, \
                                            dwpc=dwpc, wspd=wspd, wdir=wdir, missing=-
         9999, strictQC=True)
```

```
['%TITLE%' 'KCGS   010924/1800' ''
 'LEVEL       HGHT       TEMP       DWPT       WDIR       WSPD'
 '-------------------------------------------------------------------' '%RAW%'
 '1000.00,    110.39,\t   23.64,\t    20.78,     158.55,      11.69'
 '975.00,    322.74,\t   21.95,\t    17.94,\t   161.34,      15.79'
 '950.00,    540.14,\t   19.95,\t    16.78,\t   165.83,      20.64'
 '925.00,    761.32,\t   18.05,\t    15.67,    167.83,     23.06'
 '900.00,    988.56,\t   16.55,\t    14.55,\t   169.13,      24.74'
 '850.00,   1457.34,\t   13.95,\t    11.29,\t   176.13,      28.83'
 '800.00,   1949.27,\t   11.24,\t     6.39,\t   183.28,      33.87'
 '750.00,   2466.92,\t    7.95,\t     3.01,\t   187.67,      37.85'
 '700.00,   3013.51,\t    4.74,\t     1.06,\t   192.38,      40.79'
 '650.00,   3592.21,\t    1.35,\t    -0.41,\t   200.28,      43.72'
 '600.00,   4208.39,\t   -1.94,\t    -2.50,\t   208.30,      45.92'
 '550.00,   4868.29,\t   -5.04,\t    -6.14,\t   212.88,      45.83'
 '500.00,   5578.85,\t   -8.75,\t   -10.52,\t   218.14,      47.20'
 '450.00,   6349.19,   -13.54,\t   -15.38,\t   219.37,      46.26'
 '400.00,   7192.70,\t  -19.45,\t   -21.19,\t   215.62,      44.71'
 '350.00,   8126.89,\t  -26.34,\t   -27.99,\t   216.04,      45.91'
 '300.00,   9085.38,\t  -34.65,\t   -35.93,\t   221.18,      49.59'
 '250.00,  10376.20,\t  -45.84,\t   -46.41,\t   234.81,      66.11'
 '200.00,  11805.78,\t  -59.85,\t   -59.85,\t   244.40,      77.37'
 '150.00,  13637.45,\t  -63.85,\t   -67.96,\t   224.04,      65.98'
 '100.00,  16220.63,\t  -63.94,\t   -73.76,\t   228.91,      30.17' '%END%'
 '' '' '' '']
['1000.00,    110.39,\t   23.64,\t    20.78,     158.55,      11.69'
 '975.00,    322.74,\t   21.95,\t    17.94,\t   161.34,      15.79'
 '950.00,    540.14,\t   19.95,\t    16.78,\t   165.83,      20.64'
 '925.00,    761.32,\t   18.05,\t    15.67,    167.83,     23.06'
 '900.00,    988.56,\t   16.55,\t    14.55,\t   169.13,      24.74'
 '850.00,   1457.34,\t   13.95,\t    11.29,\t   176.13,      28.83'
 '800.00,   1949.27,\t   11.24,\t     6.39,\t   183.28,      33.87'
 '750.00,   2466.92,\t    7.95,\t     3.01,\t   187.67,      37.85'
 '700.00,   3013.51,\t    4.74,\t     1.06,\t   192.38,      40.79'
 '650.00,   3592.21,\t    1.35,\t    -0.41,\t   200.28,      43.72'
 '600.00,   4208.39,\t   -1.94,\t    -2.50,\t   208.30,      45.92'
 '550.00,   4868.29,\t   -5.04,\t    -6.14,\t   212.88,      45.83'
 '500.00,   5578.85,\t   -8.75,\t   -10.52,\t   218.14,      47.20'
 '450.00,   6349.19,   -13.54,\t   -15.38,\t   219.37,      46.26'
 '400.00,   7192.70,\t  -19.45,\t   -21.19,\t   215.62,      44.71'
 '350.00,   8126.89,\t  -26.34,\t   -27.99,\t   216.04,      45.91'
 '300.00,   9085.38,\t  -34.65,\t   -35.93,\t   221.18,      49.59'
 '250.00,  10376.20,\t  -45.84,\t   -46.41,\t   234.81,      66.11'
 '200.00,  11805.78,\t  -59.85,\t   -59.85,\t   244.40,      77.37'
 '150.00,  13637.45,\t  -63.85,\t   -67.96,\t   224.04,      65.98'
 '100.00,  16220.63,\t  -63.94,\t   -73.76,\t   228.91,      30.17']
```

In SHARPpy, Profile objects have quality control checks built into them to alert the user to bad data and in order to prevent the program from crashing on computational routines. For example, upon construction of the Profile object, the SHARPpy will check for unrealistic values (i.e. dewpoint or temperature below absolute zero, negative wind speeds) and incorrect ordering of the height and pressure arrays. Height arrays must be increasing with array index, and pressure arrays must be decreasing with array index. Repeat values are not allowed.

If the user wishes to avoid these checks, set the "strictQC" flag to False when constructing an object.

Because Python is an interpreted language, it can be quite slow for certain processes. When working with soundings in SHARPpy, we recommend the profiles contain a maximum of 200-500 points. High resolution radiosonde profiles (i.e. 1 second profiles) contain thousands of points and some of the SHARPpy functions that involve lifting parcels (i.e. parcelx) may take a long time to run. To filter your data to make it easier for SHARPpy to work with, you can use a sounding filter such as the one found here:

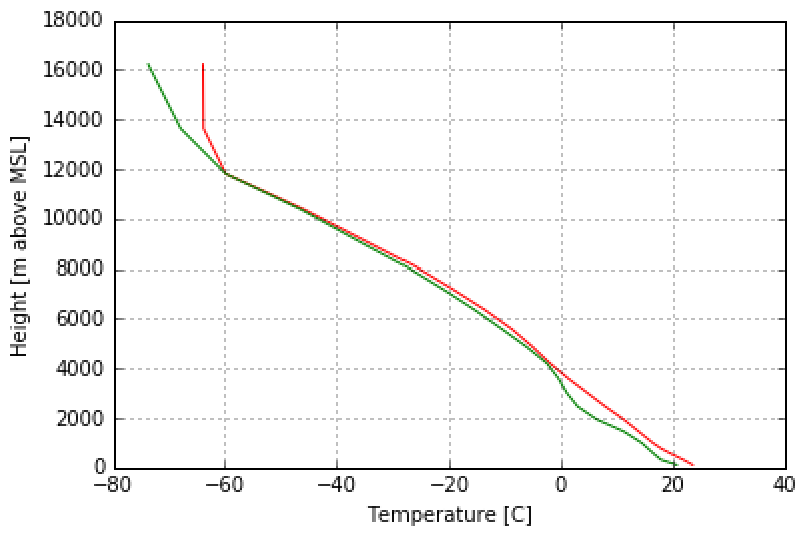
https://github.com/tsupinie/SoundingFilter (https://github.com/tsupinie/SoundingFilter)

Working with the data:

Once you have a Profile object, you can begin running analysis routines and plotting the data. The following sections show different examples of how to do this.

Plotting the data:

In [21]:
```
import matplotlib.pyplot as plt
plt.plot(prof.tmpc, prof.hght, 'r-')
plt.plot(prof.dwpc, prof.hght, 'g-')
#plt.barbs(40*np.ones(len(prof.hght)), prof.hght, prof.u, prof.v)
plt.xlabel("Temperature [C]")
plt.ylabel("Height [m above MSL]")
plt.grid()
plt.savefig("temp_dewpt_0924_1800.png",dpi=500)
plt.show()
```



SHARPpy Profile objects keep track of the height grid the profile lies on. Within the profile object, the height grid is assumed to be in meters above mean sea level.

In the example data provided, the profile can be converted to and from AGL from MSL:

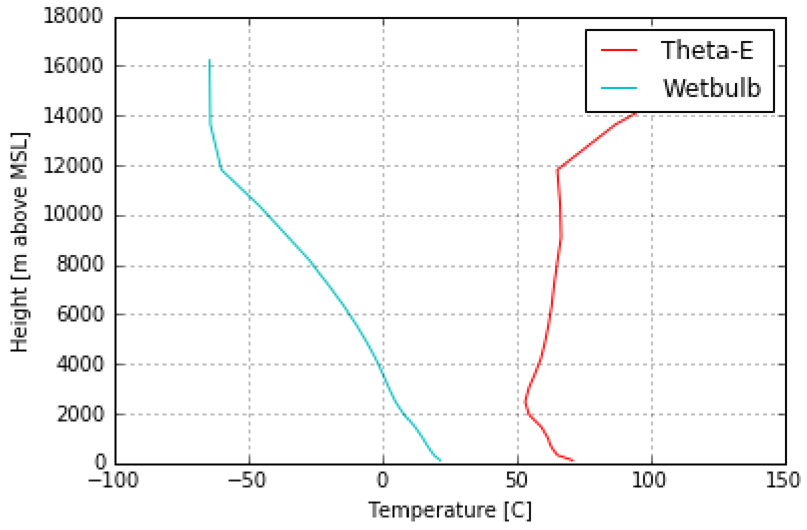
In [22]:
```
msl_hght = prof.hght[prof.sfc] # Grab the surface height value
print "SURFACE HEIGHT (m MSL):",msl_hght
agl_hght = interp.to_agl(prof, msl_hght) # Converts to AGL
print "SURFACE HEIGHT (m AGL):", agl_hght
msl_hght = interp.to_msl(prof, agl_hght) # Converts to MSL
print "SURFACE HEIGHT (m MSL):",msl_hght
```

```
SURFACE HEIGHT (m MSL): 110.39
SURFACE HEIGHT (m AGL): 0.0
SURFACE HEIGHT (m MSL): 110.39
```

Showing derived profiles:

By default, Profile objects also create derived profiles such as Theta-E and Wet-Bulb when they are constructed. These profiles are accessible to the user too.

In [23]:
```
plt.plot(thermo.ktoc(prof.thetae), prof.hght, 'r-', label='Theta-E')
plt.plot(prof.wetbulb, prof.hght, 'c-', label='Wetbulb')
plt.xlabel("Temperature [C]")
plt.ylabel("Height [m above MSL]")
plt.legend()
plt.grid()
plt.savefig("thetae_0924_1800.png",dpi=500)
plt.show()
```



Lifting Parcels:

In SHARPpy, parcels are lifted via the params.parcelx() routine. The parcelx() routine takes in the arguments of a Profile object and a flag to indicate what type of parcel you would like to be lifted. Additional arguments can allow for custom/user defined parcels to be passed to the parcelx() routine, however most users will likely be using only the Most-Unstable, Surface, 100 mb Mean Layer, and Forecast parcels.

The parcelx() routine by default utilizes the virtual temperature correction to compute variables such as CAPE and CIN. If the dewpoint profile contains missing data, parcelx() will disregard using the virtual temperature correction.

In [24]:
```
sfcpcl = params.parcelx( prof, flag=1 ) # Surface Parcel
fcstpcl = params.parcelx( prof, flag=2 ) # Forecast Parcel
mupcl = params.parcelx( prof, flag=3 ) # Most-Unstable Parcel
mlpcl = params.parcelx( prof, flag=4 ) # 100 mb Mean Layer Parcel
```

Once your parcel attributes are computed by params.parcelx(), you can extract information about the parcel such as CAPE, CIN, LFC height, LCL height, EL height, etc.

In [25]:
```
print "Most-Unstable CAPE:", mupcl.bplus # J/kg
print "Most-Unstable CIN:", mupcl.bminus # J/kg
print "Most-Unstable LCL:", mupcl.lclhght # meters AGL
print "Most-Unstable LFC:", mupcl.lfchght # meters AGL
print "Most-Unstable EL:", mupcl.elhght # meters AGL
print "Most-Unstable LI:", mupcl.li5 # C
print "MWPI1:", params.mburst1(prof)
print "MWPI2:", params.mburst2(prof)
print "DCAPE:", params.dcape(prof)
```

```
Most-Unstable CAPE: 1357.15624602
Most-Unstable CIN: -0.393441252522
Most-Unstable LCL: 353.216659786
Most-Unstable LFC: 396.979533941
Most-Unstable EL: 12191.7706589
Most-Unstable LI: -3.39776039866
MWPI1: 2.78988590082
MWPI2: 2.37046718598
DCAPE: (323.44724194674615, masked_array(data = [  5.17245307   5.17073048
7.86524825  10.35935297  12.69400525
  13.81197448  14.8908466   15.93826406  16.95844659],
             mask = False,
       fill_value = 1e+20)
, masked_array(data = [  750.   750.   800.   850.   900.   925.   950.   97
5.  1000.],
             mask = False,
       fill_value = 1e+20)
)
```

Other Parcel Object Attributes:

Here is a list of the attributes and their units contained in each parcel object (pcl):

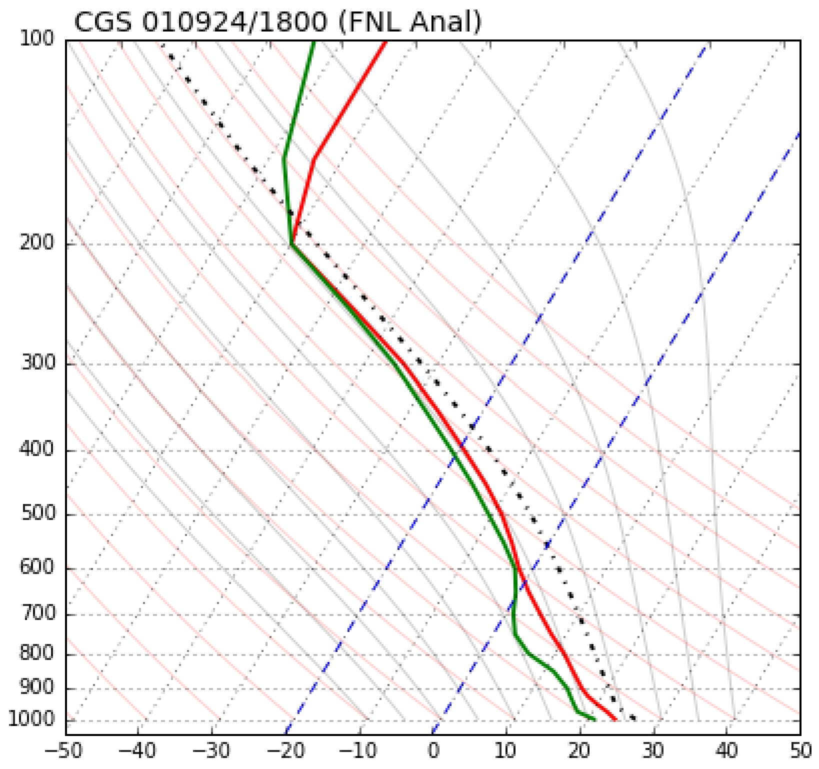
```
pcl.pres - Parcel beginning pressure (mb)
pcl.tmpc - Parcel beginning temperature (C)
pcl.dwpc - Parcel beginning dewpoint (C)
pcl.ptrace - Parcel trace pressure (mb)
pcl.ttrace - Parcel trace temperature (C)
pcl.blayer - Pressure of the bottom of the layer the parcel is lifted (mb)
pcl.tlayer - Pressure of the top of the layer the parcel is lifted (mb)
pcl.lclpres - Parcel LCL (lifted condensation level) pressure (mb)
pcl.lclhght - Parcel LCL height (m AGL)
pcl.lfcpres - Parcel LFC (level of free convection) pressure (mb)
pcl.lfchght - Parcel LFC height (m AGL)
pcl.elpres - Parcel EL (equilibrium level) pressure (mb)
pcl.elhght - Parcel EL height (m AGL)
pcl.mplpres - Maximum Parcel Level (mb)
pcl.mplhght - Maximum Parcel Level (m AGL)
pcl.bplus - Parcel CAPE (J/kg)
pcl.bminus - Parcel CIN (J/kg)
pcl.bfzl - Parcel CAPE up to freezing level (J/kg)
pcl.b3km - Parcel CAPE up to 3 km (J/kg)
pcl.b6km - Parcel CAPE up to 6 km (J/kg)
pcl.p0c - Pressure value at 0 C  (mb)
pcl.pm10c - Pressure value at -10 C (mb)
pcl.pm20c - Pressure value at -20 C (mb)
pcl.pm30c - Pressure value at -30 C (mb)
pcl.hght0c - Height value at 0 C (m AGL)
pcl.hghtm10c - Height value at -10 C (m AGL)
pcl.hghtm20c - Height value at -20 C (m AGL)
pcl.hghtm30c - Height value at -30 C (m AGL)
pcl.wm10c - Wet bulb velocity at -10 C
pcl.wm20c - Wet bulb velocity at -20 C
pcl.wm30c - Wet bulb at -30 C
pcl.li5 = - Lifted Index at 500 mb (C)
pcl.li3 = - Lifted Index at 300 mb (C)
pcl.brnshear - Bulk Richardson Number Shear
pcl.brnu - Bulk Richardson Number U (kts)
pcl.brnv - Bulk Richardson Number V (kts)
pcl.brn - Bulk Richardson Number (unitless)
pcl.limax - Maximum Lifted Index (C)
pcl.limaxpres - Pressure at Maximum Lifted Index (mb)
pcl.cap - Cap Strength (C)
pcl.cappres - Cap strength pressure (mb)
pcl.bmin - Buoyancy minimum in profile (C)
pcl.bminpres - Buoyancy minimum pressure (mb)
```

Adding a Parcel Trace and plotting Moist and Dry Adiabats:

In [26]:
```
# This serves as an intensive exercise of matplotlib's transforms
# and custom projection API. This example produces a so-called
# SkewT-logP diagram, which is a common plot in meteorology for
# displaying vertical profiles of temperature. As far as matplotlib is
# concerned, the complexity comes from having X and Y axes that are
# not orthogonal. This is handled by including a skew component to the
# basic Axes transforms. Additional complexity comes in handling the
# fact that the upper and lower X-axes have different data ranges, which
# necessitates a bunch of custom classes for ticks,spines, and the axis
# to handle this.

from matplotlib.axes import Axes
import matplotlib.transforms as transforms
import matplotlib.axis as maxis
import matplotlib.spines as mspines
import matplotlib.path as mpath
from matplotlib.projections import register_projection

# The sole purpose of this class is to look at the upper, lower, or total
# interval as appropriate and see what parts of the tick to draw, if any.
class SkewXTick(maxis.XTick):
    def draw(self, renderer):
        if not self.get_visible(): return
        renderer.open_group(self.__name__)

        lower_interval = self.axes.xaxis.lower_interval
        upper_interval = self.axes.xaxis.upper_interval

        if self.gridOn and transforms.interval_contains(
                self.axes.xaxis.get_view_interval(), self.get_loc()):
            self.gridline.draw(renderer)

        if transforms.interval_contains(lower_interval, self.get_loc()):
            if self.tick1On:
                self.tick1line.draw(renderer)
            if self.label1On:
                self.label1.draw(renderer)

        if transforms.interval_contains(upper_interval, self.get_loc()):
            if self.tick2On:
                self.tick2line.draw(renderer)
            if self.label2On:
                self.label2.draw(renderer)

        renderer.close_group(self.__name__)


# This class exists to provide two separate sets of intervals to the tick,
# as well as create instances of the custom tick
class SkewXAxis(maxis.XAxis):
    def __init__(self, *args, **kwargs):
        maxis.XAxis.__init__(self, *args, **kwargs)
        self.upper_interval = 0.0, 1.0

    def _get_tick(self, major):
        return SkewXTick(self.axes, 0, '', major=major)
```

```python
    @property
    def lower_interval(self):
        return self.axes.viewLim.intervalx

    def get_view_interval(self):
        return self.upper_interval[0], self.axes.viewLim.intervalx[1]


# This class exists to calculate the separate data range of the
# upper X-axis and draw the spine there. It also provides this range
# to the X-axis artist for ticking and gridlines
class SkewSpine(mspines.Spine):
    def _adjust_location(self):
        trans = self.axes.transDataToAxes.inverted()
        if self.spine_type == 'top':
            yloc = 1.0
        else:
            yloc = 0.0
        left = trans.transform_point((0.0, yloc))[0]
        right = trans.transform_point((1.0, yloc))[0]

        pts  = self._path.vertices
        pts[0, 0] = left
        pts[1, 0] = right
        self.axis.upper_interval = (left, right)


# This class handles registration of the skew-xaxes as a projection as well
# as setting up the appropriate transformations. It also overrides standard
# spines and axes instances as appropriate.
class SkewXAxes(Axes):
    # The projection must specify a name.  This will be used be the
    # user to select the projection, i.e. ``subplot(111,
    # projection='skewx')``.
    name = 'skewx'

    def _init_axis(self):
        #Taken from Axes and modified to use our modified X-axis
        self.xaxis = SkewXAxis(self)
        self.spines['top'].register_axis(self.xaxis)
        self.spines['bottom'].register_axis(self.xaxis)
        self.yaxis = maxis.YAxis(self)
        self.spines['left'].register_axis(self.yaxis)
        self.spines['right'].register_axis(self.yaxis)

    def _gen_axes_spines(self):
        spines = {'top':SkewSpine.linear_spine(self, 'top'),
                  'bottom':mspines.Spine.linear_spine(self, 'bottom'),
                  'left':mspines.Spine.linear_spine(self, 'left'),
                  'right':mspines.Spine.linear_spine(self, 'right')}
        return spines

    def _set_lim_and_transforms(self):
        """
        This is called once when the plot is created to set up all the
        transforms for the data, text and grids.
        """
        rot = 30

        #Get the standard transform setup from the Axes base class
        Axes._set_lim_and_transforms(self)

        # Need to put the skew in the middle, after the scale and limits,
        # but before the transAxes. This way, the skew is done in Axes
        # coordinates thus performing the transform around the proper origin
        # We keep the pre-transAxes transform around for other users, like the
        # spines for finding bounds
        self.transDataToAxes = self.transScale + (self.transLimits +
                transforms.Affine2D().skew_deg(rot, 0))

        # Create the full transform from Data to Pixels
        self.transData = self.transDataToAxes + self.transAxes

        # Blended transforms like this need to have the skewing applied using
        # both axes, in axes coords like before.
        self._xaxis_transform = (transforms.blended_transform_factory(
                    self.transScale + self.transLimits,
                    transforms.IdentityTransform()) +
                transforms.Affine2D().skew_deg(rot, 0)) + self.transAxes

# Now register the projection with matplotlib so the user can select
# it.
register_projection(SkewXAxes)

pcl = mupcl
# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(6.5875, 6.2125))
ax = fig.add_subplot(111, projection='skewx')
ax.grid(True)

pmax = 1000
pmin = 10
dp = -10
presvals = np.arange(int(pmax), int(pmin)+dp, dp)

# plot the moist-adiabats
for t in np.arange(-10,45,5):
    tw = []
    for p in presvals:
        tw.append(thermo.wetlift(1000., t, p))
    ax.semilogy(tw, presvals, 'k-', alpha=.2)

def thetas(theta, presvals):
    return ((theta + thermo.ZEROCNK) / (np.power((1000. / presvals),thermo.ROC
P))) - thermo.ZEROCNK

# plot the dry adiabats
for t in np.arange(-50,110,10):
    ax.semilogy(thetas(t, presvals), presvals, 'r-', alpha=.2)

plt.title(' CGS 010924/1800 (FNL Anal)', fontsize=14, loc='left')
# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dicatated by the typical meteorological plot
```

```
ax.semilogy(prof.tmpc, prof.pres, 'r', lw=2)
ax.semilogy(prof.dwpc, prof.pres, 'g', lw=2)
ax.semilogy(pcl.ttrace, pcl.ptrace, 'k-.', lw=2)

# An example of a slanted line at constant X
l = ax.axvline(0, color='b', linestyle='--')
l = ax.axvline(-20, color='b', linestyle='--')

# Disables the log-formatting that comes with semilogy
ax.yaxis.set_major_formatter(plt.ScalarFormatter())
ax.set_yticks(np.linspace(100,1000,10))
ax.set_ylim(1050,100)

ax.xaxis.set_major_locator(plt.MultipleLocator(10))
ax.set_xlim(-50,50)
plt.savefig("CGS_1800.png",dpi=500)
plt.show()
```



Calculating Kinematic Variables:

SHARPpy also allows the user to compute kinematic variables such as shear, mean-winds, and storm relative helicity. SHARPpy will also compute storm motion vectors based off of the work by Stephen Corfidi and Matthew Bunkers. Below is some example code to compute the following:

1.) 0-3 km Pressure-Weighted Mean Wind

2.) 0-6 km Shear (kts)

3.) Bunker's Storm Motion (right-mover) (Bunkers et al. 2014 version)

4.) Bunker's Storm Motion (left-mover) (Bunkers et al. 2014 version)

5.) 0-3 Storm Relative Helicity

```
In [27]: sfc = prof.pres[prof.sfc]
         p3km = interp.pres(prof, interp.to_msl(prof, 3000.))
         p6km = interp.pres(prof, interp.to_msl(prof, 6000.))
         p1km = interp.pres(prof, interp.to_msl(prof, 1000.))
         mean_3km = winds.mean_wind(prof, pbot=sfc, ptop=p3km)
         sfc_6km_shear = winds.wind_shear(prof, pbot=sfc, ptop=p6km)
         sfc_3km_shear = winds.wind_shear(prof, pbot=sfc, ptop=p3km)
         sfc_1km_shear = winds.wind_shear(prof, pbot=sfc, ptop=p1km)
         print "0-3 km Pressure-Weighted Mean Wind (kt):", utils.comp2vec(mean_3km[0],
         mean_3km[1])[1]
         print "0-6 km Shear (kt):", utils.comp2vec(sfc_6km_shear[0], sfc_6km_shear[1])
         [1]
         srwind = params.bunkers_storm_motion(prof)
         print "Bunker's Storm Motion (right-mover) [deg,kts]:", utils.comp2vec(srwind[
         0], srwind[1])
         print "Bunker's Storm Motion (left-mover) [deg,kts]:", utils.comp2vec(srwind[2
         ], srwind[3])
         srh3km = winds.helicity(prof, 0, 3000., stu = srwind[0], stv = srwind[1])
         srh1km = winds.helicity(prof, 0, 1000., stu = srwind[0], stv = srwind[1])
         print "0-3 km Storm Relative Helicity [m2/s2]:",srh3km[0]
```

```
0-3 km Pressure-Weighted Mean Wind (kt): 27.7420278513
0-6 km Shear (kt): 42.0292670293
Bunker's Storm Motion (right-mover) [deg,kts]: (masked_array(data = 221.06664
4628,
             mask = False,
       fill_value = -9999.0)
, 27.715783425191876)
Bunker's Storm Motion (left-mover) [deg,kts]: (masked_array(data = 179.294620
303,
             mask = False,
       fill_value = -9999.0)
, 43.237551079794827)
0-3 km Storm Relative Helicity [m2/s2]: 205.062270304
```

Calculating variables based off of the effective inflow layer:

The effective inflow layer concept is used to obtain the layer of buoyant parcels that feed a storm's inflow. Here are a few examples of how to compute variables that require the effective inflow layer in order to calculate them:

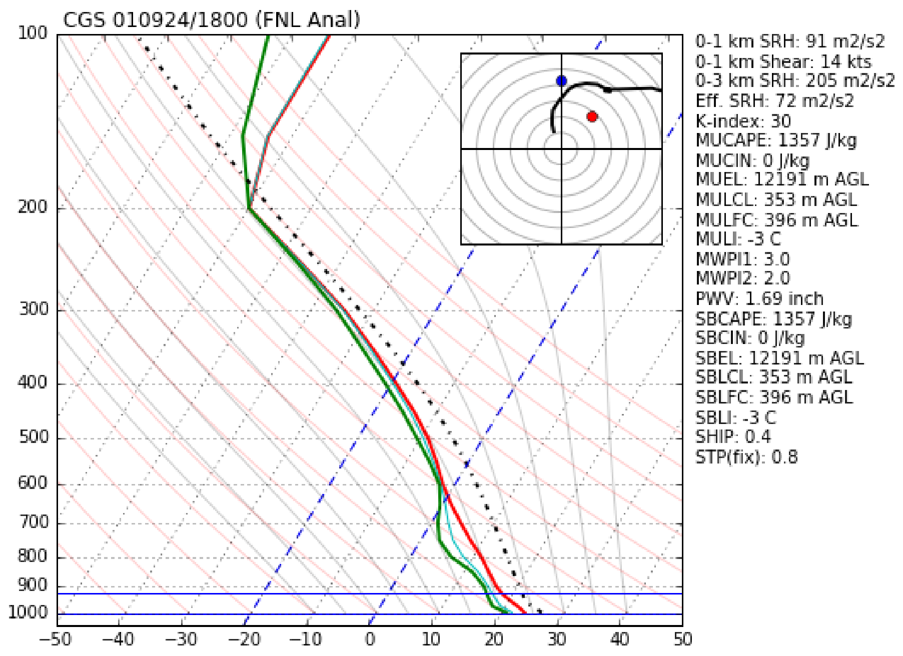
In [28]:
```
stp_fixed = params.stp_fixed(sfcpcl.bplus, sfcpcl.lclhght, srh1km[0], utils.co
mp2vec(sfc_6km_shear[0], sfc_6km_shear[1])[1])
ship = params.ship(prof)
eff_inflow = params.effective_inflow_layer(prof)
ebot_hght = interp.to_agl(prof, interp.hght(prof, eff_inflow[0]))
etop_hght = interp.to_agl(prof, interp.hght(prof, eff_inflow[1]))
print "Effective Inflow Layer Bottom Height (m AGL):", ebot_hght
print "Effective Inflow Layer Top Height (m AGL):", etop_hght
effective_srh = winds.helicity(prof, ebot_hght, etop_hght, stu = srwind[0], st
v = srwind[1])
print "Effective Inflow Layer SRH (m2/s2):", effective_srh[0]
ebwd = winds.wind_shear(prof, pbot=eff_inflow[0], ptop=eff_inflow[1])
ebwspd = utils.mag( ebwd[0], ebwd[1] )
print "Effective Bulk Wind Difference:", ebwspd
scp = params.scp(mupcl.bplus, effective_srh[0], ebwspd)
stp_cin = params.stp_cin(mlpcl.bplus, effective_srh[0], ebwspd, mlpcl.lclhght,
mlpcl.bminus)
print "Supercell Composite Parameter:", scp
print "Significant Tornado Parameter (w/CIN):", stp_cin
print "Significant Tornado Parameter (fixed):", stp_fixed
```

```
Effective Inflow Layer Bottom Height (m AGL): 0.0
Effective Inflow Layer Top Height (m AGL): 650.93
Effective Inflow Layer SRH (m2/s2): 72.9618823232
Effective Bulk Wind Difference: 11.6761793849
Supercell Composite Parameter: 1.15618315613
Significant Tornado Parameter (w/CIN): 0.0
Significant Tornado Parameter (fixed): 0.82717009895
```

Putting it all together into one plot:

In [32]:
```python
indices = {'SBCAPE': [int(sfcpcl.bplus), 'J/kg'],\
           'SBCIN': [int(sfcpcl.bminus), 'J/kg'],\
           'SBLCL': [int(sfcpcl.lclhght), 'm AGL'],\
           'SBLFC': [int(sfcpcl.lfchght), 'm AGL'],\
           'SBEL': [int(sfcpcl.elhght), 'm AGL'],\
           'SBLI': [int(sfcpcl.li5), 'C'],\
           'MUCAPE': [int(mupcl.bplus), 'J/kg'],\
           'MUCIN': [int(mupcl.bminus), 'J/kg'],\
           'MULCL': [int(mupcl.lclhght), 'm AGL'],\
           'MULFC': [int(mupcl.lfchght), 'm AGL'],\
           'MUEL': [int(mupcl.elhght), 'm AGL'],\
           'MULI': [int(mupcl.li5), 'C'],\
           '0-1 km SRH': [int(srh1km[0]), 'm2/s2'],\
           '0-1 km Shear': [int(utils.comp2vec(sfc_1km_shear[0], sfc_1km_shear
[1])[1]), 'kts'],\
           '0-3 km SRH': [int(srh3km[0]), 'm2/s2'],\
           'Eff. SRH': [int(effective_srh[0]), 'm2/s2'],\
           'PWV': [round(params.precip_water(prof), 2), 'inch'],\
           'K-index': [int(params.k_index(prof)), ''],\
           'MWPI1': [round(params.mburst1(prof)), ''],\
           'MWPI2': [round(params.mburst2(prof)), ''],\
           'STP(fix)': [round(stp_fixed, 1), ''],\
           'SHIP': [round(ship, 1), '']}

# Set the parcel trace to be plotted as the Most-Unstable parcel.
mupcl = params.parcelx( prof, flag=3 ) # Most-Unstable Parcel
pcl = mupcl

# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(6.5875, 6.2125))
ax = fig.add_subplot(111, projection='skewx')
ax.grid(True)

pmax = 1000
pmin = 10
dp = -10
presvals = np.arange(int(pmax), int(pmin)+dp, dp)

# plot the moist-adiabats
for t in np.arange(-10,45,5):
    tw = []
    for p in presvals:
        tw.append(thermo.wetlift(1000., t, p))
    ax.semilogy(tw, presvals, 'k-', alpha=.2)

def thetas(theta, presvals):
    return ((theta + thermo.ZEROCNK) / (np.power((1000. / presvals),thermo.ROC
P))) - thermo.ZEROCNK

# plot the dry adiabats
for t in np.arange(-50,110,10):
    ax.semilogy(thetas(t, presvals), presvals, 'r-', alpha=.2)

plt.title(' CGS 010924/1800 (FNL Anal)', fontsize=12, loc='left')
# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dicatated by the typical meteorological plot
```

```
ax.semilogy(prof.tmpc, prof.pres, 'r', lw=2) # Plot the temperature profile
ax.semilogy(prof.wetbulb, prof.pres, 'c-') # Plot the wetbulb profile
ax.semilogy(prof.dwpc, prof.pres, 'g', lw=2) # plot the dewpoint profile
ax.semilogy(pcl.ttrace, pcl.ptrace, 'k-.', lw=2) # plot the parcel trace
# An example of a slanted line at constant X
l = ax.axvline(0, color='b', linestyle='--')
l = ax.axvline(-20, color='b', linestyle='--')

# Plot the effective inflow layer using blue horizontal lines
ax.axhline(eff_inflow[0], color='b')
ax.axhline(eff_inflow[1], color='b')

#plt.barbs(10*np.ones(len(prof.pres)), prof.pres, prof.u, prof.v)
# Disables the log-formatting that comes with semilogy
ax.yaxis.set_major_formatter(plt.ScalarFormatter())
ax.set_yticks(np.linspace(100,1000,10))
ax.set_ylim(1050,100)
ax.xaxis.set_major_locator(plt.MultipleLocator(10))
ax.set_xlim(-50,50)

# List the indices within the indices dictionary on the side of the plot.
string = ''
for key in np.sort(indices.keys()):
    string = string + key + ': ' + str(indices[key][0]) + ' ' + indices[key][1
] + '\n'
plt.text(1.02, 1, string, verticalalignment='top', transform=plt.gca().transAx
es)

# Draw the hodograph on the Skew-T.
# TAS 2015-4-16: hodograph doesn't plot for some reason ...
ax2 = plt.axes([.625,.625,.25,.25])
below_12km = np.where(interp.to_agl(prof, prof.hght) < 12000)[0]
u_prof = prof.u[below_12km]
v_prof = prof.v[below_12km]
ax2.plot(u_prof[~u_prof.mask], v_prof[~u_prof.mask], 'k-', lw=2)
ax2.get_xaxis().set_visible(False)
ax2.get_yaxis().set_visible(False)
for i in range(10,90,10):
    # Draw the range rings around the hodograph.
    circle = plt.Circle((0,0),i,color='k',alpha=.3, fill=False)
    ax2.add_artist(circle)
ax2.plot(srwind[0], srwind[1], 'ro') # Plot Bunker's Storm motion right mover
 as a red dot
ax2.plot(srwind[2], srwind[3], 'bo') # Plot Bunker's Storm motion left mover a
s a blue dot

ax2.set_xlim(-60,60)
ax2.set_ylim(-60,60)
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
plt.savefig("CGS_1800.png",dpi=500, bbox_inches="tight")
plt.show()
```



List of functions in each module:

This tutorial cannot cover all of the functions in SHARPpy. Below is a list of all of the functions accessible through SHARPTAB. In order to learn more about the function in this IPython Notebook, open up a new "In[]:" field and type in the path to the function (for example):

```
params.dcape()
```

Documentation should appear below the cursor describing the function itself, the function's arguments, its output values, and any references to meteorological literature the function was based on.

```
In [15]: print "Functions within params.py:"
         for key in params.__all__:
             print "\tparams." + key + "()"
         print "\nFunctions within winds.py:"
         for key in winds.__all__:
             print "\twinds." + key + "()"
         print "\nFunctions within thermo.py:"
         for key in thermo.__all__:
             print "\tthermo." + key + "()"
         print "\nFunctions within interp.py:"
         for key in interp.__all__:
             print "\tinterp." + key + "()"
         print "\nFunctions within utils.py:"
         for key in utils.__all__:
             print "\tutils." + key + "()"
```

```
Functions within params.py:
        params.DefineParcel()
        params.Parcel()
        params.inferred_temp_advection()
        params.k_index()
        params.t_totals()
        params.c_totals()
        params.v_totals()
        params.precip_water()
        params.temp_lvl()
        params.max_temp()
        params.mean_mixratio()
        params.mean_theta()
        params.mean_thetae()
        params.mean_relh()
        params.lapse_rate()
        params.most_unstable_level()
        params.parcelx()
        params.bulk_rich()
        params.bunkers_storm_motion()
        params.effective_inflow_layer()
        params.convective_temp()
        params.esp()
        params.pbl_top()
        params.precip_eff()
        params.dcape()
        params.sig_severe()
        params.dgz()
        params.ship()
        params.stp_cin()
        params.stp_fixed()
        params.scp()
        params.mmp()
        params.wndg()
        params.sherb()
        params.tei()
        params.cape()
        params.mburst1()
        params.mburst2()
        params.dcp()
        params.ehi()
        params.sweat()
        params.hgz()
        params.lhp()

Functions within winds.py:
        winds.mean_wind()
        winds.mean_wind_npw()
        winds.mean_wind_old()
        winds.mean_wind_npw_old()
        winds.sr_wind()
        winds.sr_wind_npw()
        winds.wind_shear()
        winds.helicity()
        winds.max_wind()
        winds.non_parcel_bunkers_motion()
        winds.corfidi_mcs_motion()
        winds.mbe_vectors()
        winds.non_parcel_bunkers_motion_experimental()
        winds.critical_angle()

Functions within thermo.py:
        thermo.drylift()
        thermo.thalvl()
        thermo.lcltemp()
        thermo.theta()
        thermo.wobf()
        thermo.satlift()
        thermo.wetlift()
        thermo.lifted()
        thermo.vappres()
        thermo.mixratio()
        thermo.temp_at_mixrat()
        thermo.wetbulb()
        thermo.thetaw()
        thermo.thetae()
        thermo.virtemp()
        thermo.relh()
        thermo.ftoc()
        thermo.ctof()
        thermo.ctok()
        thermo.ktoc()
        thermo.ftok()
        thermo.ktof()

Functions within interp.py:
        interp.pres()
        interp.hght()
        interp.temp()
        interp.dwpt()
        interp.vtmp()
        interp.components()
        interp.vec()
        interp.to_agl()
        interp.to_msl()

Functions within utils.py:
        utils.INT2STR()
        utils.FLOAT2STR()
        utils.MS2KTS()
        utils.KTS2MS()
        utils.MS2MPH()
        utils.MPH2MS()
        utils.MPH2KTS()
        utils.KTS2MPH()
        utils.M2FT()
        utils.FT2M()
        utils.vec2comp()
        utils.comp2vec()
        utils.mag()
        utils.QC()
```