Jupyter notebook demonstrating the use of additional PmagPy functions

This Jupyter notebook demonstrates a number of PmagPy functions within a notebook environment running a Python 2.7 kernel. The benefits of working within these notebooks include: reproducibility, interactive code development, convenient workspace for projects, version control (when integrated with GitHub or other version control software) and ease of sharing.

The notebook can be viewed as html at the following link where the code and tables are better rendered than in this PDF: `http://pmagpy.github.io/Additional_PmagPy_Examples.html`

1 Contents of the notebook

1.1 Paleomagnetic Data Analysis Walkthrough

Basic Functions * The Dipole Equation * Get local geomagnetic field estimate from IGRF * Plotting Directional Data * Calculating the Angle Between Two Directions * Fisher-Distributed Directions * Flip Directional Data

Data Analysis * Test if Directions Are Fisher-Distributed * Simulating Inclination Error in Paleomagnetic Data * Correcting for Inclination Error in Paleomagnetic Data * Bootstrap Reversal Test * McFadden and McElhinny (1990) Reversal Test

Plotting Paleomagnetic Poles * Working with Poles * Calculate and Plot VGPs * Plotting APWPs

1.2 Rock Magnetism Data Analysis

- Working with Anisotropy Data
- Working with Curie Temperature Data
- Day Plots
- Hysteresis Loops
- Demagnetization Curves

1.3 Additional Features of the Jupyter Notebook

- Interactive Plotting

Note: This notebook makes use of pandas for reading, displaying, and using data with a dataframe structure. More information about the pandas module and its use within PmagPy can be found here within the documentation of the PmagPy Cookbook.

```
In [1]: # With the PmagPy folder in the PYTHONPATH,
        # the function modules from PmagPy can be imported
        import pmagpy.ipmag as ipmag
        import pmagpy.pmagplotlib as pmagplotlib
        import pmagpy.pmag as pmag

        from mpl_toolkits.basemap import Basemap
        import numpy as np
        import pandas as pd
```

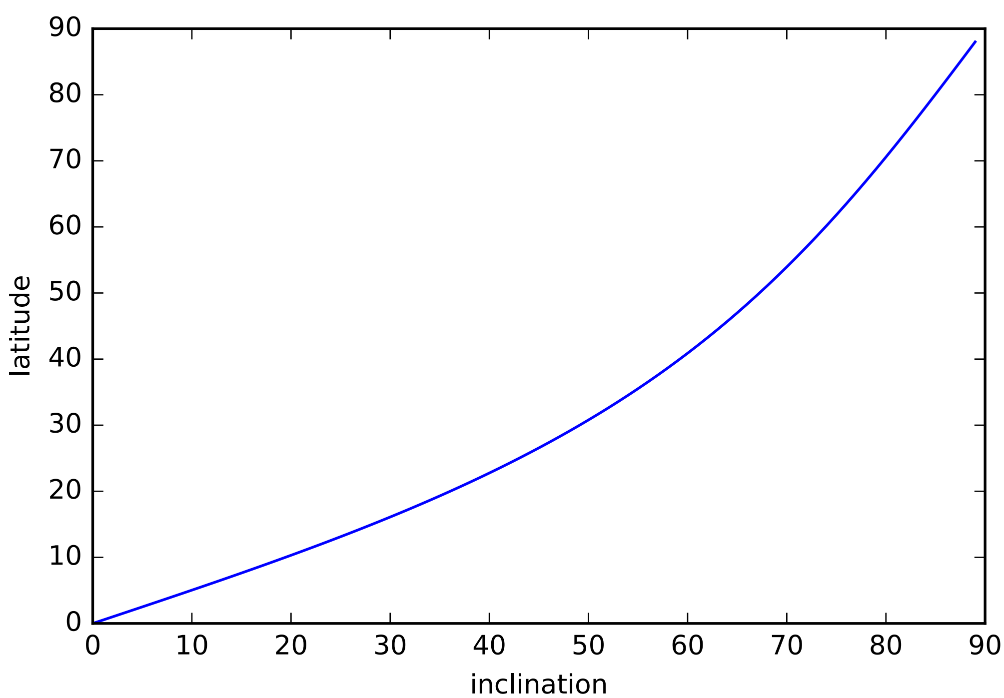
```
import matplotlib.pyplot as plt
import os
%matplotlib inline
%config InlineBackend.figure_formats = {'svg',}
```

2 The dipole equation

The following demonstrates the use of a simple function (**ipmag.lat_from_inc**) which uses the dipole equation to return expected latitude from inclination data as predicted by a pure geocentric axial dipole. The expected inclination for the geomagnetic field can be calculated from a specified latitude using **ipmag.inc_from_lat**.

```
In [2]: inclination = range(0,90,1)
        latitude = []
        for inc in inclination:
            lat = ipmag.lat_from_inc(inc)
            latitude.append(lat)
```

```
In [3]: plt.plot(inclination,latitude)
        plt.ylabel('latitude')
        plt.xlabel('inclination')
        plt.show()
```



Go to Top

3 Get local geomagnetic field estimate from IGRF

The function **ipmag.igrf** uses the International Geomagnetic Reference Field (IGRF) model to estimate the geomagnetic field direction at a particular location and time. Let's find the direction of the geomagnetic field in Berkeley, California (37.87° N, 122.27° W, elevation of 52 m) on August 27, 2013 (in decimal format, 2013.6544).

```
In [4]: berk_igrf = ipmag.igrf([2013.6544, .052, 37.871667, -122.272778])
        ipmag.igrf_print(berk_igrf)
```

```
Declination: 13.950
Inclination: 61.354
Intensity: 13.950 nT
```

Go to Top

4 Plotting Directions

We can plot this direction using **matplotlib** (**plt**) in conjunction with a few **ipmag** functions. To do this, we first initiate a figure (numbered as Fig. 0, with a size of 6x6) with the following syntax:

```
plt.figure(num=0,figsize=(6,6))
```

We then draw an equal area stereonet within the figure, specifying the figure number:

```
ipmag.plot_net(0)
```

Now we can plot the direction we just pulled from IGRF using **ipmag.plot_di()**:

```
ipmag.plot_di(berk_igrf[0],berk_igrf[1])
```

To label or color the plotted points, we would pass the same code as above but with a few extra arguments and one additional line of code:

```
ipmag.plot_di(berk_igrf[0],berk_igrf[1], color='r', label="Berkeley, CA -- August 27, 2013")
plt.legend()
```

We may wish to save the figure we just created. To do so, we would pass the following *save* function, specifying 1) the relative path to the folder where we want the figure to be saved and 2) the name of the file with the desired extension (.pdf in this example):

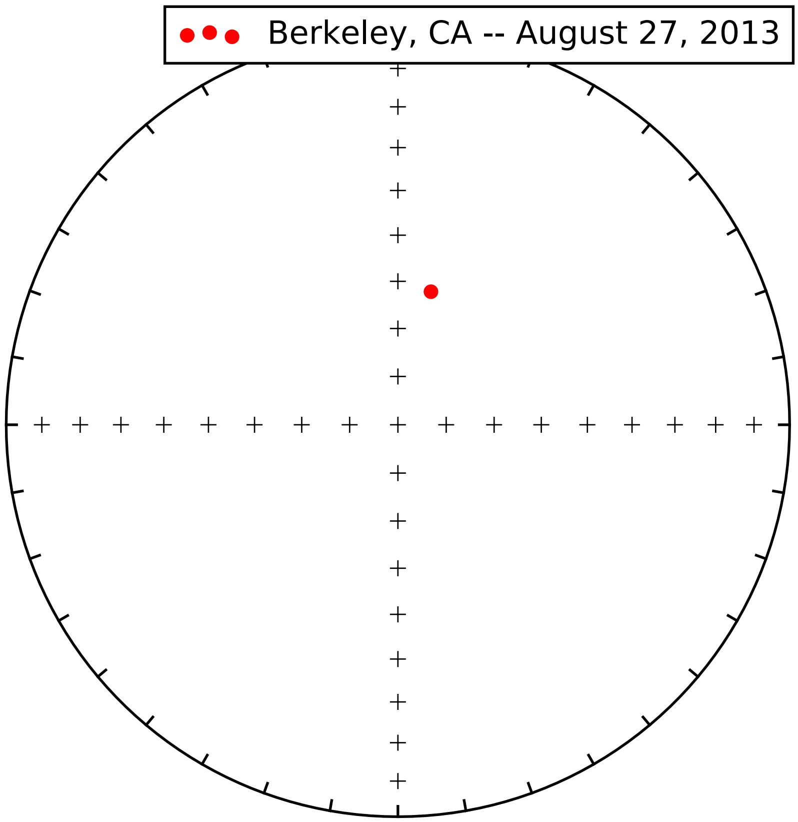
```
plt.savefig("./Additional_Notebook_Output/Berkeley_IGRF.pdf")
```

To ensure the figure is displayed properly and then cleared from the namespace, it is good practice to end such a code block with the following:

```
plt.show()
```

Now let's run the code we just developed.

```
In [5]: plt.figure(num=0,figsize=(5,5))
        ipmag.plot_net(0)
        ipmag.plot_di(berk_igrf[0],berk_igrf[1], color='r', label="Berkeley, CA -- August 27, 2013")
        plt.legend()
        plt.savefig("./Additional_Notebook_Output/Berkeley_IGRF.pdf")
        plt.show()
```



Let's see how this magnetic direction compares to the Geocentric Axial Dipole (GAD) model of the geomagnetic field. We can estimate the expected GAD inclination by passing Berkeley's latitude to the function **ipmag.inc_from_lat**.

We also demonstrate below how to manipulate the placement of the figure legend to ensure no data points are obscured. **plt.legend** uses the "best" location by default, but this can be changed with the following:

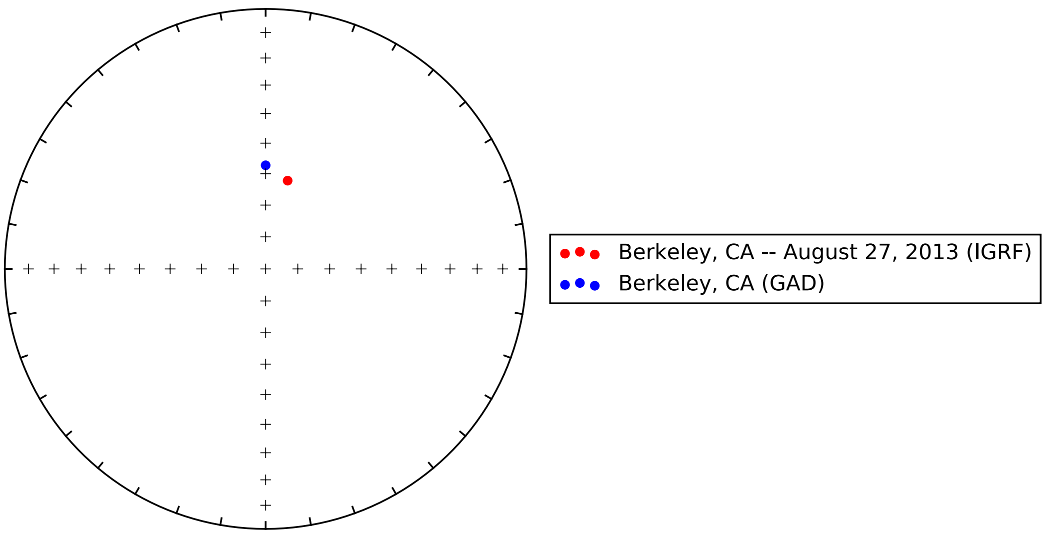
```
plt.legend(loc="upper right")
```

or

```
plt.legend(loc="lower center")
```

See the **plt.legend** documentation for the complete list of placement options. Alternatively, you can give `(x,y)` coordinates to the `loc=` keyword argument (with the origin `(0,0)` at the lower left of the figure). To manipulate placement even more precisely, use the keyword `bbox_to_anchor` in conjunction with `loc`. If this is done, `loc` becomes the anchor point on the legend, and `bbox_to_anchor` places this anchor point at the specified coordinates. The latter method is demonstrated below. Play around with the **plt.legend** arguments to see how this changes things.

```
In [6]: GAD_inc = ipmag.inc_from_lat(37.87)
        plt.figure(num=0,figsize=(5,5))
        ipmag.plot_net(0)
        ipmag.plot_di(berk_igrf[0],berk_igrf[1], color='r', label="Berkeley, CA -- August 27, 2013 (IGRI
        ipmag.plot_di(0,GAD_inc, color='b', label="Berkeley, CA (GAD)")
        plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
        plt.show()
```



Below, we calculate the angular difference between these two directions.

Go to Top

5 Calculate the Angle Between Directions

While **ipmag** functions have been optimized to preform tasks within an interactive computing environment such as the Jupyter notebook, the **pmag** functions which are used extensively within **ipmag** can also be directly called. Here is a demonstration of the function **pmag.angle**, which calculates the angle between two directions and outputs a **numpy** array. Continuing our comparison from the last section, let's calculate the angle between the IGRF and GAD-estimated magnetic directions calculated and plotted above.

In [7]:
```
direction1 = [berk_igrf[0],berk_igrf[1]]
direction2 = [0,GAD_inc]
print pmag.angle(direction1,direction2)[0]
```

```
8.18973048085
```

Go to Top

6 Generate and plot Fisher distributed unit vectors from a specified distribution

Let's use the function **ipmag.fishrot** to generate a set of 50 Fisher-distributed directions at a declination of 200° and inclination of 45°. These directions will serve as an example paleomagnetic dataset that will be used for the next several examples. The output from **ipmag.fishrot** is a nested list of lists of vectors (declination, inclination, intensity). Generally these vectors are unit vectors with an intensity of 1.0. We refer to this data structure as a di_block. In the code below the first two vectors are shown.

In [8]:
```
fisher_directions = ipmag.fishrot(k=40, n=50, dec=200, inc=50)
fisher_directions[0:2]
```

Out[8]:
```
[[183.99588954660211, 45.029054467016621, 1.0],
 [200.58837258558225, 41.811402826036407, 1.0]]
```

This di_block can be unpacked in separate lists of declination and inclination using the **ipmag.unpack_di_block** function.

In [9]:
```
fisher_decs, fisher_incs = ipmag.unpack_di_block(fisher_directions)
print fisher_decs[0]
print fisher_incs[0]
```

```
183.995889547
45.029054467
```

Another way to deal with the di_block is to make it into a pandas dataframe which allows for the direction to be nicely displayed and analyzed. In the code below, a dataframe is made from the *fisher_directions* di_block and then the first 5 rows are displayed with .head().

In [10]:
```
directions = pd.DataFrame(fisher_directions,columns=['dec','inc','length'])
directions.head()
```

Out[10]:

	dec	inc	length
0	183.995890	45.029054	1
1	200.588373	41.811403	1
2	199.546318	36.865786	1
3	182.021565	57.114399	1
4	221.438217	38.160275	1

Now let's calculate the Fisher and Bingham means of these data.

```
In [11]: fisher_mean = ipmag.fisher_mean(directions.dec,directions.inc)
         bingham_mean = ipmag.bingham_mean(directions.dec,directions.inc)
```

Here's the raw output of the Fisher mean which is a dictionary containing the mean direction and associated statistics:

```
In [12]: fisher_mean
```

```
Out[12]: {'alpha95': 3.159239986865467,
          'csd': 12.553483567656313,
          'dec': 198.86339034639676,
          'inc': 49.196847025953076,
          'k': 41.633365663104868,
          'n': 50,
          'r': 48.823059360693883}
```

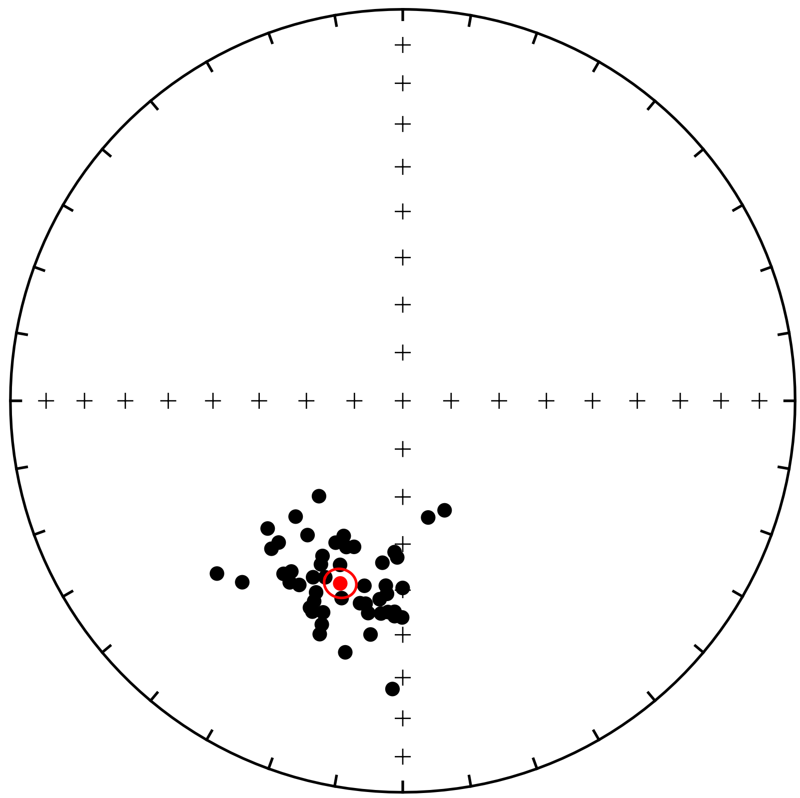
The function **ipmag.print_direction_mean** prints formatted output from this Fisher mean dictionary:

```
In [13]: ipmag.print_direction_mean(fisher_mean)
```

```
Dec: 198.9  Inc: 49.2
Number of directions in mean (n): 50
Angular radius of 95% confidence (a_95): 3.2
Precision parameter (k) estimate: 41.6
```

Now we can plot all of our data using the function **ipmag.plot_di**. We can also plot the Fisher mean with its angular radius of 95% confidence (α_{95}) using **ipmag.plot_di_mean**.

```
In [14]: declinations = directions.dec.tolist()
         inclinations = directions.inc.tolist()

         plt.figure(num=1,figsize=(5,5))
         ipmag.plot_net(1)
         ipmag.plot_di(declinations,inclinations)
         ipmag.plot_di_mean(fisher_mean['dec'],fisher_mean['inc'],fisher_mean['alpha95'],color='r')
```

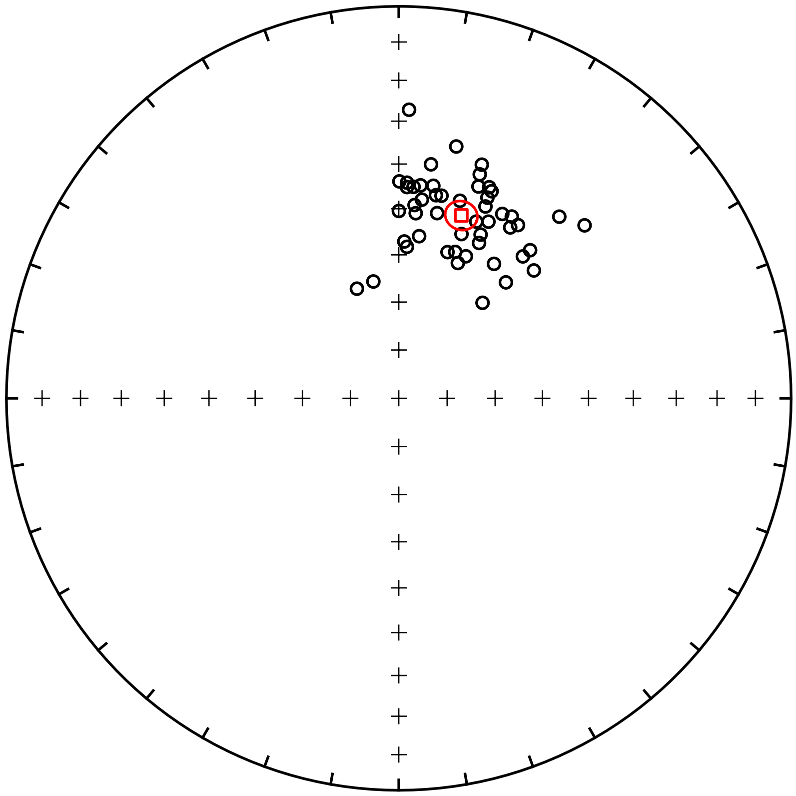


Go to Top

7 Flip polarity of directional data

Let's flip all the directions (find their antipodes) of the Fisher-distributed population using the function **ipmag.do_flip()** function and plot the resulting directions.

In [15]:
```
# get reversed directions
dec_reversed,inc_reversed = ipmag.do_flip(declinations,inclinations)

# take the Fisher mean of these reversed directions
rev_mean = ipmag.fisher_mean(dec_reversed,inc_reversed)

# plot the flipped directions
plt.figure(num=1,figsize=(5,5))
ipmag.plot_net(1)
ipmag.plot_di(dec_reversed, inc_reversed)
ipmag.plot_di_mean(rev_mean['dec'],rev_mean['inc'],rev_mean['alpha95'],color='r',marker='s')
```



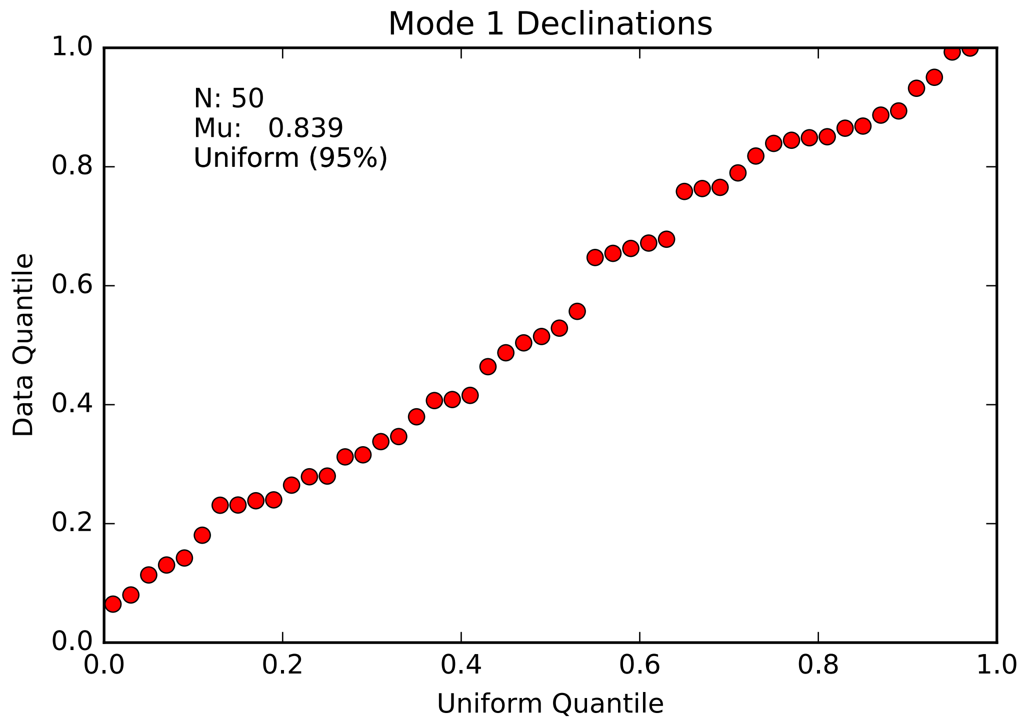
Go to Top

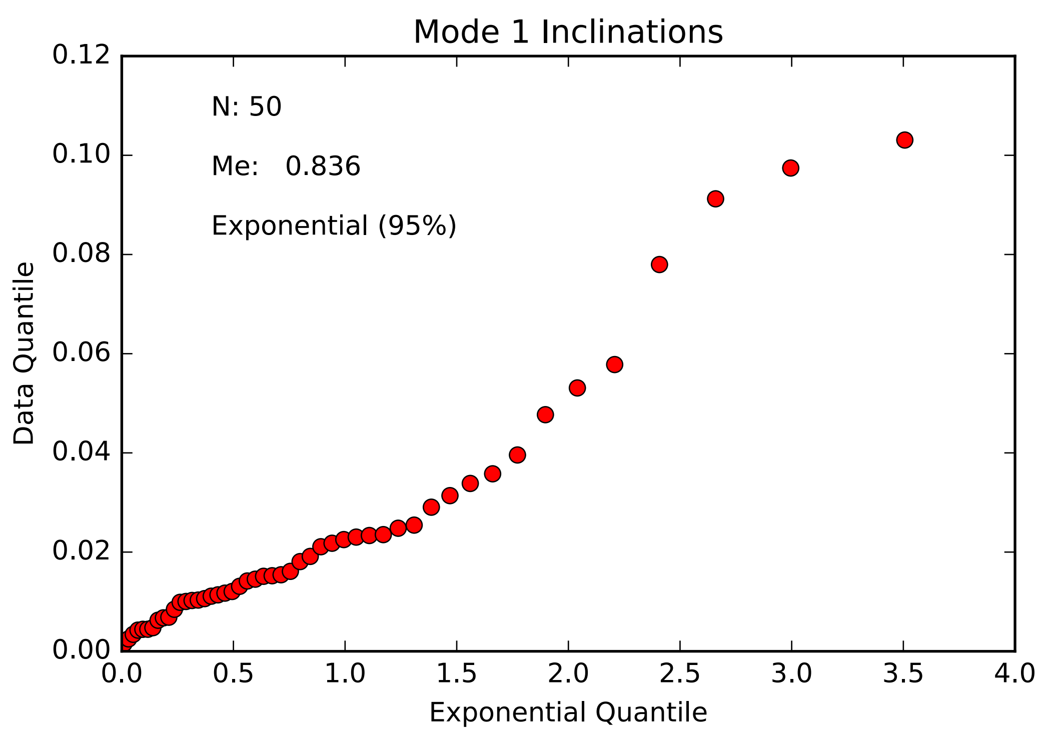
8 Test directional data for Fisher distribution

The function **ipmag.fishqq** tests whether directional data are Fisher-distributed. Let's use this test on the random Fisher-distributed directions we just created (it should pass!).

```
In [16]: ipmag.fishqq(declinations, inclinations)
```

```
Out[16]: {'Dec': 198.86305338122148,
          'Inc': 49.174480281318843,
          'Me': 0.83647984581601553,
          'Me_critical': 1.094,
          'Mode': 'Mode 1',
          'Mu': 0.83924082417308898,
          'Mu_critical': 1.207,
          'N': 50,
          'Test_result': 'consistent with Fisherian model'}
```


Mode 1 Declinations
N: 50
Mu: 0.839
Uniform (95%)
Data Quantile
Uniform Quantile
0.0
0.2
0.4
0.6
0.8
1.0

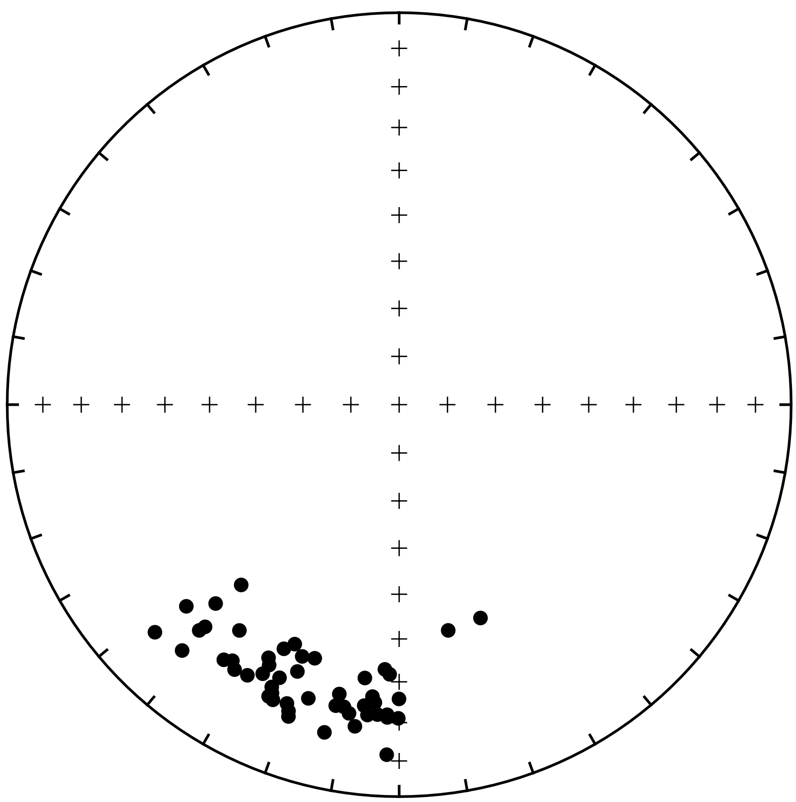

Mode 1 Inclinations
N: 50
Me: 0.836
Exponential (95%)
Data Quantile
Exponential Quantile
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Go to Top

9 Squish directional data

Inclination flattening can occur for magnetizations in sedimentary rocks. We can simulate inclination error of a specified "flattening factor" with the function **ipmag.squish**. Flattening factors range from 0 (completely flattened) to 1 (no flattening). Let's squish our directions with a 0.4 flattening factor.

```
In [17]: # squish all inclinations
         squished_incs = []
         for inclination in inclinations:
             squished_incs.append(ipmag.squish(inclination, 0.4))

         # plot the squished directional data
         plt.figure(num=1,figsize=(5,5))
         ipmag.plot_net(1)
         ipmag.plot_di(declinations,squished_incs)
         squished_DIs = np.array(zip(declinations,squished_incs))
```



In [18]:
```
ipmag.fisher_mean(di_block=squished_DIs)
```

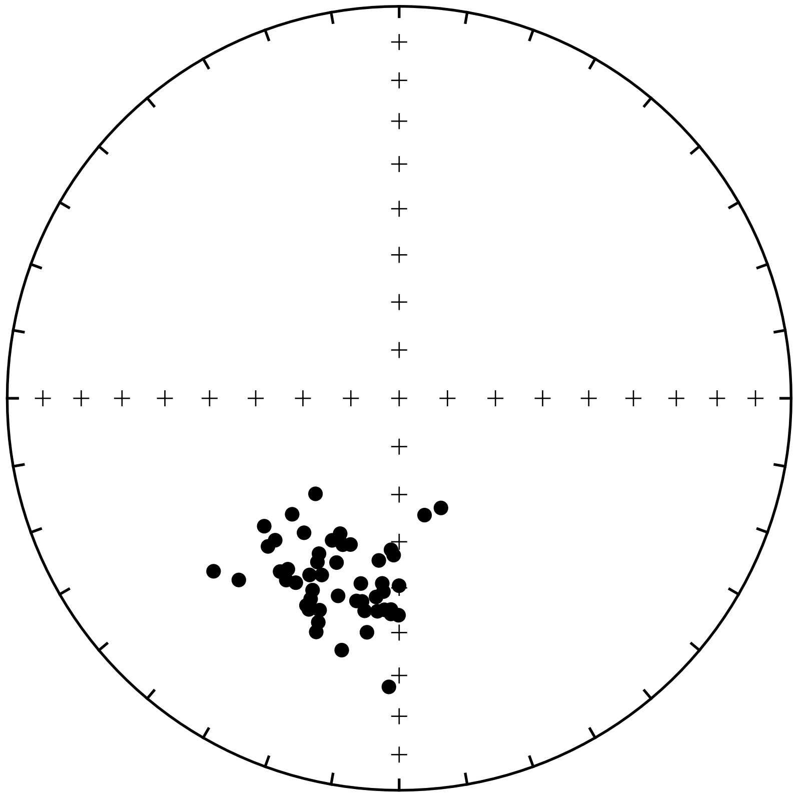
Out[18]:
```
{'alpha95': 3.7940228548357475,
 'csd': 14.997170953500827,
 'dec': 198.72106646496212,
 'inc': 25.567588455989114,
 'k': 29.171002445333873,
 'n': 50,
 'r': 48.320249703731456}
```

Go to Top

10 Unsquish directional data

We can also "unsquish" data by a specified flattening factor. Let's unsquish the data we squished above with the function **ipmag.unsquish**. Using a flattening factor of 0.4 will restore the data to its original state.

In [19]:
```
unsquished_incs = []
for squished_inc in squished_incs:
    unsquished_incs.append(ipmag.unsquish(squished_inc, 0.4))

# plot the squished directional data
plt.figure(num=1,figsize=(5,5))
ipmag.plot_net(1)
ipmag.plot_di(declinations,unsquished_incs)
```

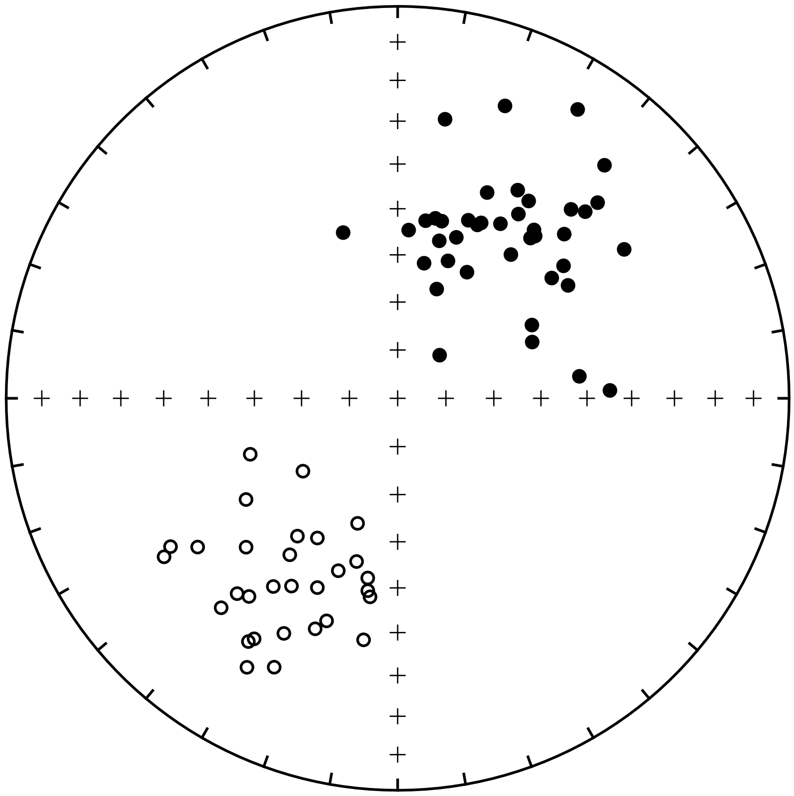


Go to Top

11 Bootstrap Reversal Test

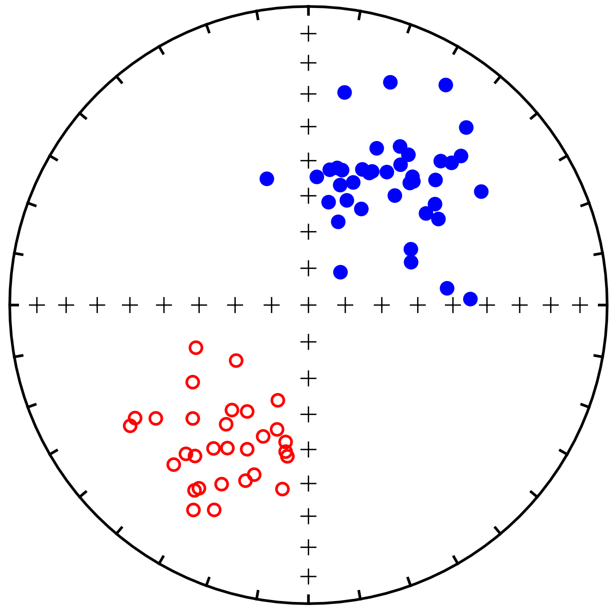
Here we carry out two types of reversal tests with **ipmag** to test if two populations are antipodal to one another: the bootstrap reversal test (Tauxe, 2010; **ipmag.reversal_test_bootstrap**) and the McFadden and McElhinny (1990) reversal test, which is an adaptation of the Watson V test for a common mean (**ipmag.reversal_test_MM1990**). The code below uses **ipmag.fishrot** to simulate normal directions and reversed directions from antipodal Fisher distributions.

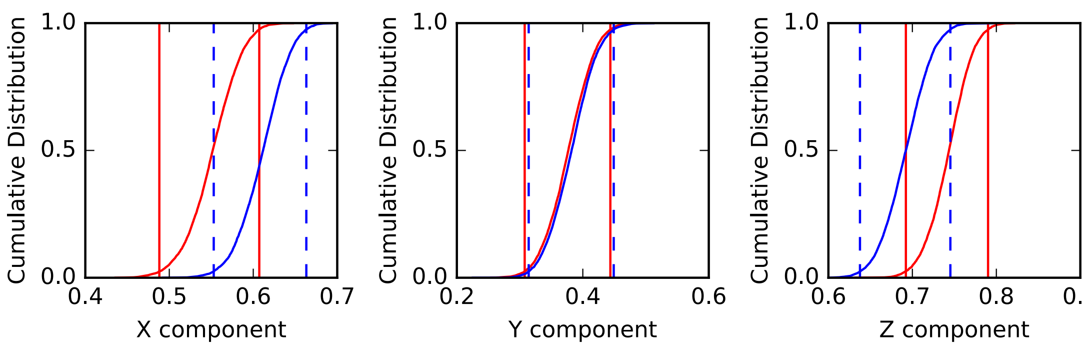
In [20]:
```
normal_directions = ipmag.fishrot(k=20,n=40,dec=30,inc=45)
reversed_directions = ipmag.fishrot(k=20,n=30,dec=210,inc=-45)
combined_directions = normal_directions + reversed_directions

plt.figure(num=1,figsize=(5,5))
ipmag.plot_net(1)
ipmag.plot_di(di_block=combined_directions)
```



```
In [21]: ipmag.reversal_test_bootstrap(di_block=combined_directions,
                                       plot_stereo=True, save=True,
                                       save_folder='./Additional_Notebook_Output/')
```

Here are the results of the bootstrap test for a common mean:





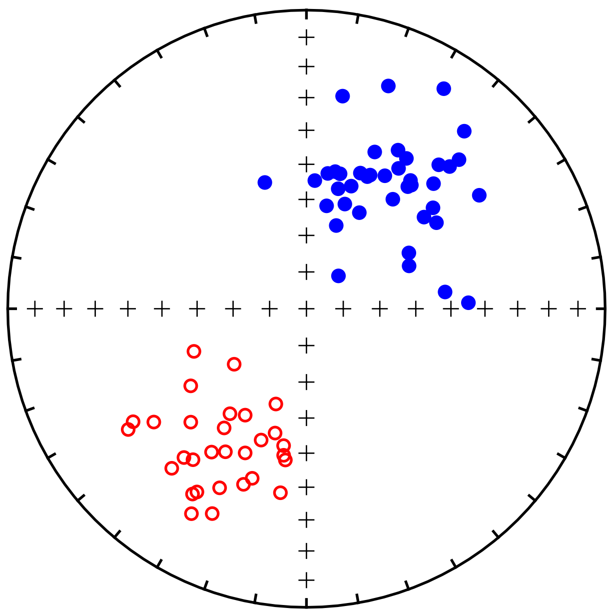
Go to Top

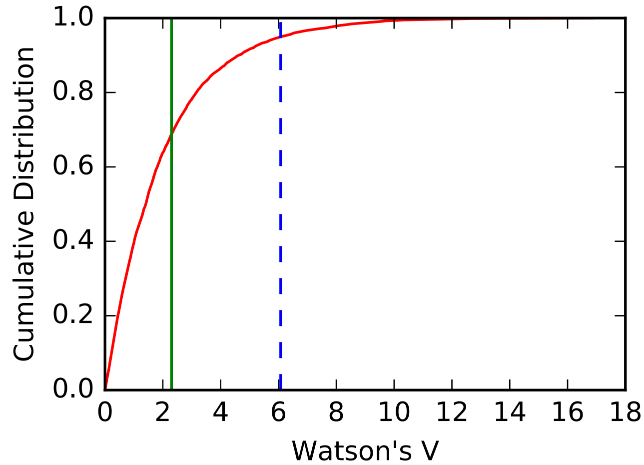
McFadden and McElhinny (1990) Reversal Test

```
In [22]: ipmag.reversal_test_MM1990(di_block=combined_directions,
                                    plot_CDF=True, plot_stereo=True,
                                    save=True, save_folder= './Additional_Notebook_Output/')
```

```
Results of Watson V test:
```

```
Watson's V:           2.3
Critical value of V:  6.1
"Pass": Since V is less than Vcrit, the null hypothesis
that the two populations are drawn from distributions
that share a common mean direction can not be rejected.

M&M1990 classification:

Angle between data set means: 4.6
Critical angle for M&M1990:   7.4
The McFadden and McElhinny (1990) classification for
this test is: 'B'
```





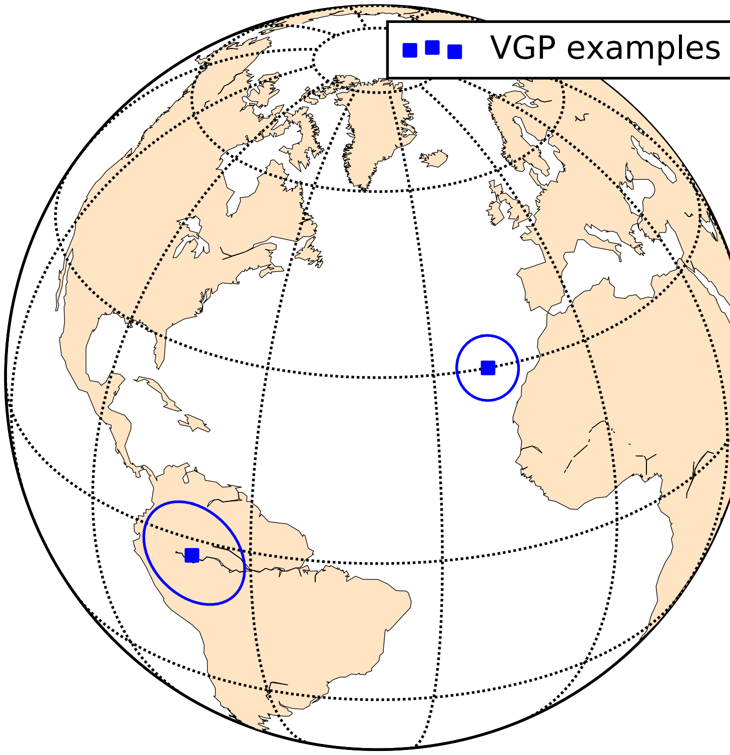
Go to Top

12 Working with Poles

A variety of plotting functions within PmagPy, together with the Basemap package of matplotlib, provide a great way to work with paleomagnetic poles, virtual geomagnetic poles, and polar wander paths.

In [23]:
```
# initiate figure and specify figure size
plt.figure(figsize=(5, 5))

# initiate a Basemap projection, specifying the latitude and
# longitude (lat_0 and lon_0) at which our figure is centered.
pmap = Basemap(projection='ortho',lat_0=30,lon_0=320,
               resolution='c',area_thresh=50000)
# other optional modifications to the globe figure
pmap.drawcoastlines(linewidth=0.25)
pmap.fillcontinents(color='bisque',lake_color='white',zorder=1)
pmap.drawmapboundary(fill_color='white')
pmap.drawmeridians(np.arange(0,360,30))
pmap.drawparallels(np.arange(-90,90,30))


# Here we plot a pole at 340 E longitude, 30 N latitude with an
# alpha 95 error angle of 5 degrees. Keyword arguments allow us
# to specify the label, shape, and color of this data.
ipmag.plot_pole(pmap,340,30,5,label='VGP examples',
               marker='s',color='Blue')

# We can plot multiple poles sequentially on the same globe using
# the same plot_pole function.
ipmag.plot_pole(pmap,290,-3,9,marker='s',color='Blue')
```

```
plt.legend()
# Optional save (uncomment to save the figure)
#plt.savefig('Code_output/VGP_example.pdf')
plt.show()
```



Go to Top

13 Calculate and Plot VGPs

Using the function **ipmag.vgp_calc**, we can calculate virtual geomagnetic poles (VGPs) of our fFisher-distributed directions. We'll need to first assign a location to these magnetic directions - let's assume they are from Berkeley, CA (37.87° N, 122.27° W).

In [24]:
```
# plug in site latitude and longitude to the "directions" dataframe
directions['site_lat'] = 37.97
directions['site_lon'] = -122.27

# calculate VGPs (this automatically adds VGP data to the dataframe)
ipmag.vgp_calc(directions, dec_tc = 'dec', inc_tc = 'inc')
directions.head()
```

Out[24]:
```
          dec        inc  length  site_lat  site_lon  paleolatitude  \
0  183.995890  45.029054       1     37.97   -122.27      26.588302
```

```
1  200.588373  41.811403        1      37.97    -122.27        24.095659
2  199.546318  36.865786        1      37.97    -122.27        20.553229
3  182.021565  57.114399        1      37.97    -122.27        37.715082
4  221.438217  38.160275        1      37.97    -122.27        21.449872

      vgp_lat      vgp_lon   vgp_lat_rev  vgp_lon_rev
0 -25.333013  233.776578    25.333013    53.776578
1 -24.992308  216.987184    24.992308    36.987184
2 -28.660199  216.813085    28.660199    36.813085
3 -14.291968  236.079855    14.291968    56.079855
4 -18.969679  197.086706    18.969679    17.086706
```

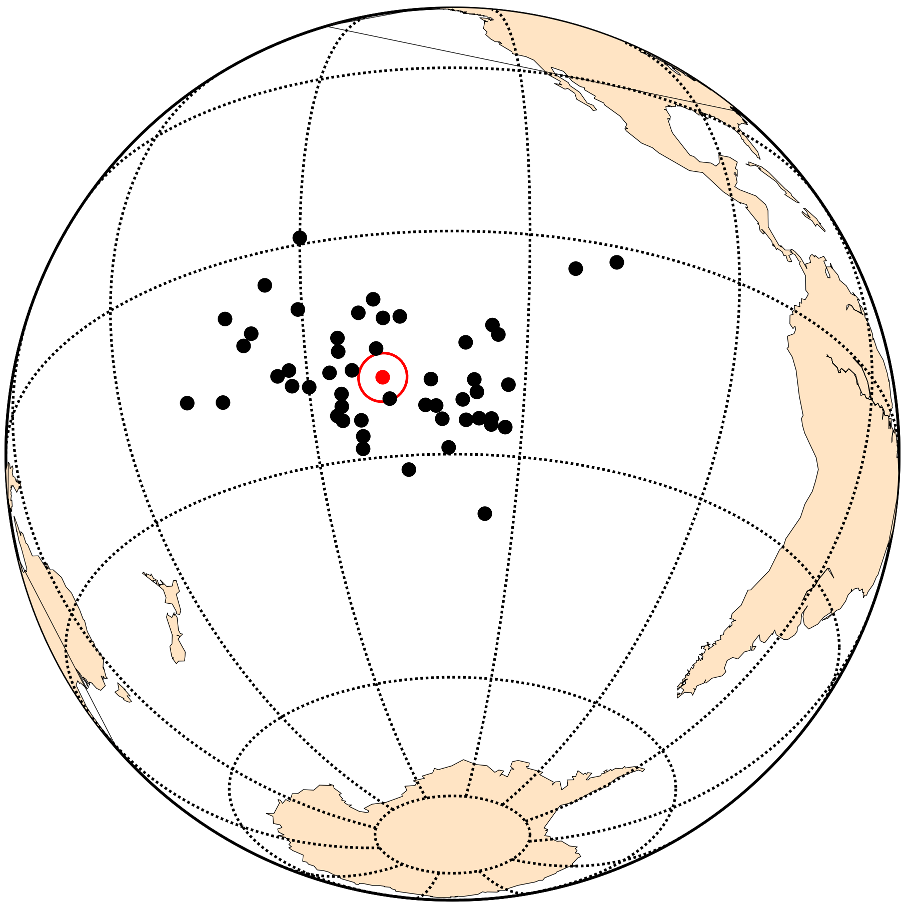
We have already calculated the Fisher mean of this data, so let's translate it to a VGP too. For a one-line dataset, we plug the Fisher mean data into a **pandas** *Series* instead of a *DataFrame* (a *DataFrame* can be considered a sequence of concatenated *Series*).

In [25]:
```
mean_pole = pd.Series(fisher_mean)
mean_pole['site_lat'] = 37.97
mean_pole['site_lon'] = -122.27
ipmag.vgp_calc(mean_pole, dec_tc = 'dec', inc_tc = 'inc')
mean_pole
```

Out[25]:
```
alpha95                 3.15924
csd                     12.5535
dec                     198.863
inc                     49.1968
k                       41.6334
n                            50
r                       48.8231
site_lat                  37.97
site_lon                -122.27
paleolatitude            30.079
vgp_lat                -19.7048
vgp_lon            220.44194226
vgp_lat_rev             19.7048
vgp_lon_rev             40.4419
dtype: object
```

In [26]:
```
plt.figure(figsize=(6, 6))
pmap = Basemap(projection='ortho',lat_0=-30,lon_0=-130,
               resolution='c',area_thresh=50000)
pmap.drawcoastlines(linewidth=0.25)
pmap.fillcontinents(color='bisque',lake_color='white',zorder=1)
pmap.drawmapboundary(fill_color='white')
pmap.drawmeridians(np.arange(0,360,30))
pmap.drawparallels(np.arange(-90,90,30))

# use the print_pole_mean function to print the mean data above the globe
ipmag.print_pole_mean(mean_pole)
for n in range(len(directions)):
    ipmag.plot_vgp(pmap, directions['vgp_lon'][n],
                   directions['vgp_lat'][n])
ipmag.plot_pole(pmap, mean_pole['vgp_lon'], mean_pole['vgp_lat'],
                mean_pole['alpha95'], color='r')
```

```
Plong: 198.9  Plat: 49.2
Number of directions in mean (n): 50.0
Angular radius of 95% confidence (A_95): 3.2
Precision parameter (k) estimate: 41.6
```



Go to Top

14 Plotting APWPs

The capability to plot multiple poles in sequence provides a good way to visualize polar wander paths. Here we use the Phanerozoic APWP of Laurentia *(Torsvik, 2012)* to demonstrate the plot_pole_colorbar function.

We first upload the Torsvik (2012) data using the pandas function *read_csv*.

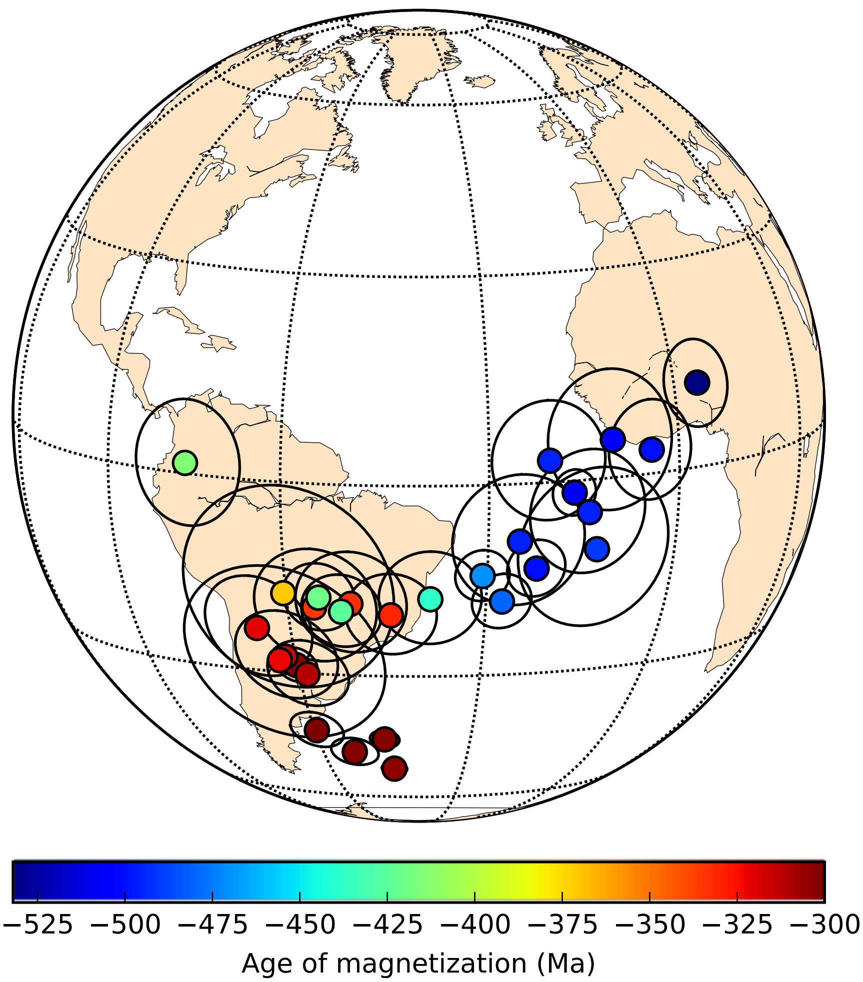
In [27]:
```
Laurentia_Pole_Compilation = pd.read_csv('./Additional_Data/Torsvik2012/Laurentia_Pole_Compila
Laurentia_Pole_Compilation.head()
```

Out[27]:

	Q	A95	Com	Formation	Lat	Lon	CLat	CLon	RLat	\
0	5	3.9	NaN	Dunkard Formation	-44.1	301.5	-41.5	300.4	-38.0	

```
1  5  2.1  NaN      Laborcita  Formation -42.1  312.1 -43.0  313.4 -32.7
2  5  3.4    #        Wescogame Formation -44.1  303.9 -46.3  306.8 -38.2
3  6  3.1    I         Glenshaw Formation -28.6  299.9 -28.6  299.9 -28.6
4  5  1.8  NaN  Lower  Casper  Formation -45.7  308.6 -50.5  314.6 -37.6

   RLon              EULER  Age GPDB RefNo/Reference
0  43.0  (63.2/_ 13.9/79.9)  300                302, T
1  52.9  (63.2/_ 13.9/79.9)  301               1311,  T
2  51.4  (63.2/_ 13.9/79.9)  301               1311,  T
3  32.4  (63.2/_ 13.9/79.9)  303        Kodama (2009)
4  59.8  (63.2/_ 13.9/79.9)  303               1455,  T
```

In [28]:
```
# initiate the figure as in the plot_pole example
plt.figure(figsize=(6, 6))
pmap = Basemap(projection='ortho',lat_0=10,lon_0=320,
               resolution='c',area_thresh=50000)
pmap.drawcoastlines(linewidth=0.25)
pmap.fillcontinents(color='bisque',lake_color='white',zorder=1)
pmap.drawmapboundary(fill_color='white')
pmap.drawmeridians(np.arange(0,360,30))
pmap.drawparallels(np.arange(-90,90,30))

# Loop through the uploaded data and use the plot_pole_colorbar function
# (instead of plot_pole) to plot the individual poles. The input of this
# function is very similar to that of plot_pole but has the additional
# arguments of (1)AGE, (2)MINIMUM AND (3)MAXIMUM AGES OF PLOTTED POLES.
# Note that the ages are treated as negative numbers -- this just determines
# the direction of the colorbar.
for n in xrange (0, len(Laurentia_Pole_Compilation)):
     m = ipmag.plot_pole_colorbar(pmap, Laurentia_Pole_Compilation['CLon'][n],
                                  Laurentia_Pole_Compilation['CLat'][n],
                                  Laurentia_Pole_Compilation['A95'][n],
                                  -Laurentia_Pole_Compilation['Age'][n],
                                  -532,
                                  -300,
                                  markersize=80, color="k", alpha=1)

pmap.colorbar(m,location='bottom',pad="5%",label='Age of magnetization (Ma)')

# Optional save (uncomment to save the figure)
#plt.savefig('Additional_Notebook_Output/plot_pole_colorbar_example.pdf')

plt.show()
```



Go to Top

15 Working with anisotropy data

The following code demonstrates reading magnetic anisotropy data into a pandas DataFrame.

In [29]:
```
aniso_data = pd.read_csv('./Additional_Data/ani_depthplot/rmag_anisotropy.txt',
                         delimiter='\t',skiprows=1)
aniso_data.head()
```

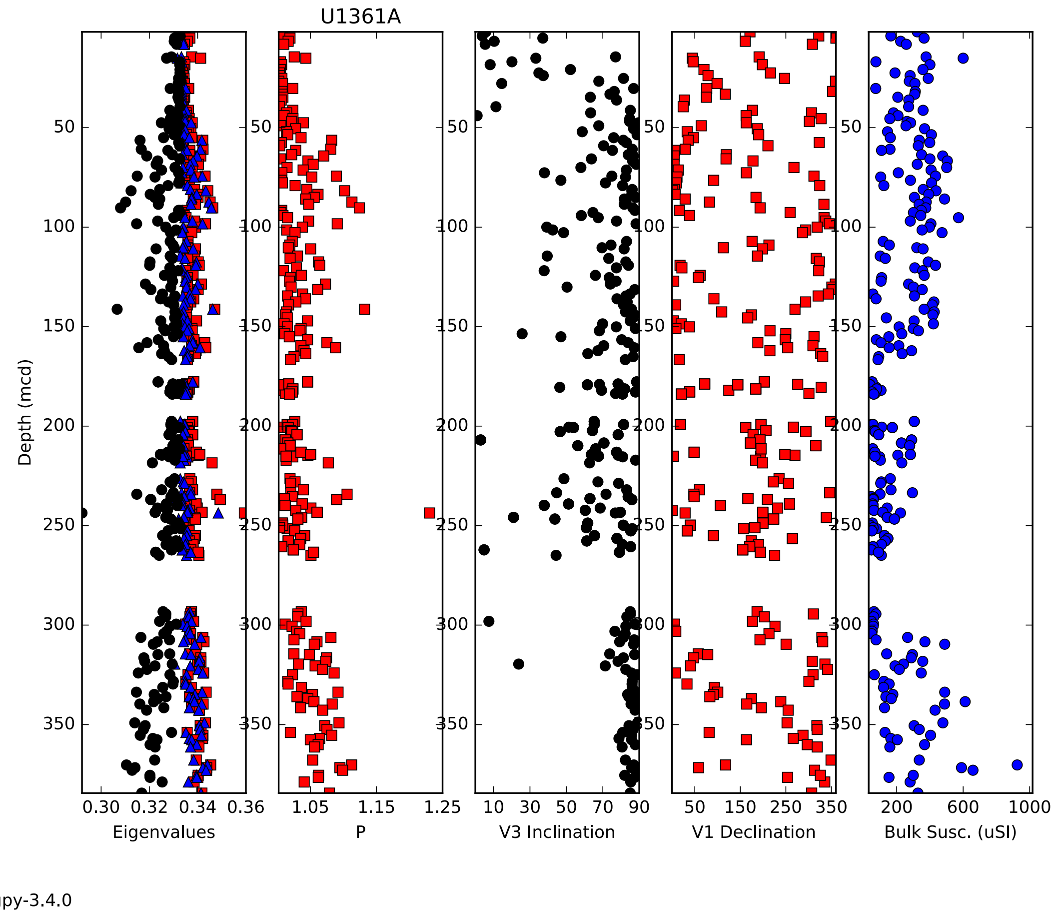
Out[29]:

	anisotropy_n	anisotropy_s1	anisotropy_s2	anisotropy_s3	anisotropy_s4 \
0	192	0.332294	0.332862	0.334844	-0.000048
1	192	0.333086	0.332999	0.333916	-0.000262
2	192	0.333750	0.332208	0.334041	-0.000699
3	192	0.330565	0.333928	0.335507	0.000603
4	192	0.332747	0.332939	0.334314	-0.001516

```
   anisotropy_s5  anisotropy_s6  anisotropy_sigma  anisotropy_tilt_correction  \
0       0.000027      -0.000263          0.000122                          -1
1      -0.000322       0.000440          0.000259                          -1
2       0.000663       0.002888          0.000093                          -1
3       0.000212      -0.000932          0.000198                          -1
4      -0.000311      -0.000099          0.000162                          -1

  anisotropy_type      anisotropy_unit  er_analyst_mail_names  \
0             AMS  Normalized by trace                    NaN
1             AMS  Normalized by trace                    NaN
2             AMS  Normalized by trace                    NaN
3             AMS  Normalized by trace                    NaN
4             AMS  Normalized by trace                    NaN

  er_citation_names er_location_name          er_sample_name  \
0        This study           U1361A  318-U1361A-001H-2-W-35
1        This study           U1361A  318-U1361A-001H-3-W-90
2        This study           U1361A  318-U1361A-001H-4-W-50
3        This study           U1361A  318-U1361A-001H-5-W-59
4        This study           U1361A  318-U1361A-001H-6-W-60

             er_site_name        er_specimen_name      magic_method_codes
0  318-U1361A-001H-2-W-35  318-U1361A-001H-2-W-35  LP-X:AE-H:LP-AN-MS:SO-V
1  318-U1361A-001H-3-W-90  318-U1361A-001H-3-W-90  LP-X:AE-H:LP-AN-MS:SO-V
2  318-U1361A-001H-4-W-50  318-U1361A-001H-4-W-50  LP-X:AE-H:LP-AN-MS:SO-V
3  318-U1361A-001H-5-W-59  318-U1361A-001H-5-W-59  LP-X:AE-H:LP-AN-MS:SO-V
4  318-U1361A-001H-6-W-60  318-U1361A-001H-6-W-60  LP-X:AE-H:LP-AN-MS:SO-V
```

The function **ipmag.aniso_depthplot** is one example of how PmagPy works with such data to generate plots.

```
In [30]: ipmag.aniso_depthplot(dir_path='./Additional_Data/ani_depthplot/');
```

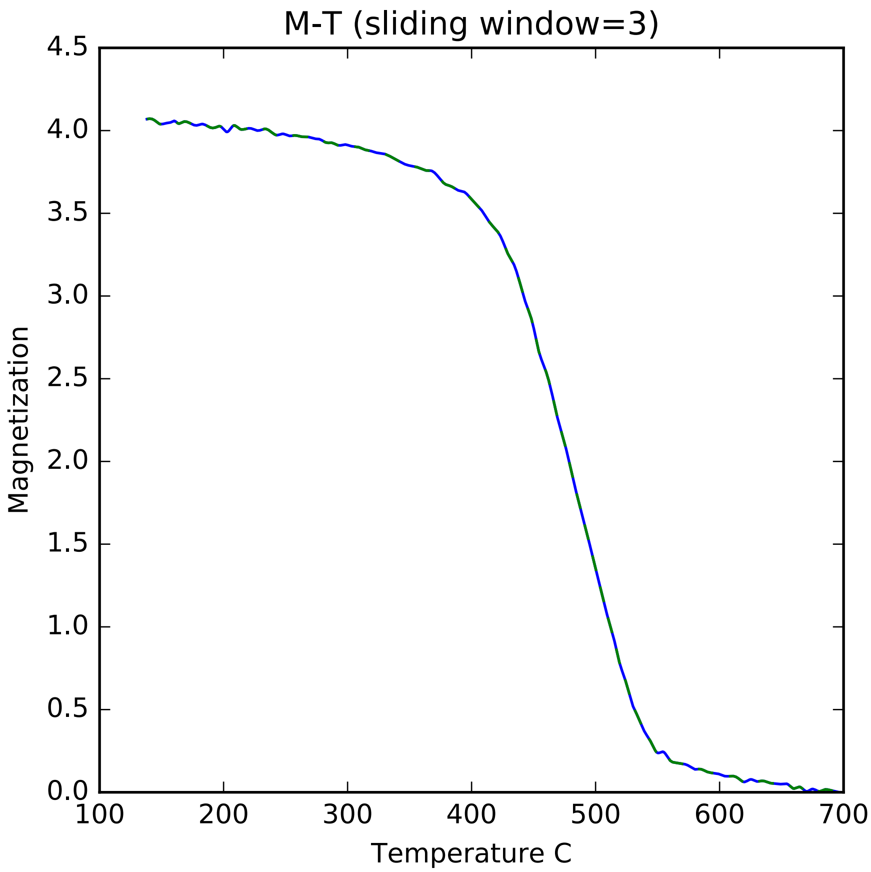


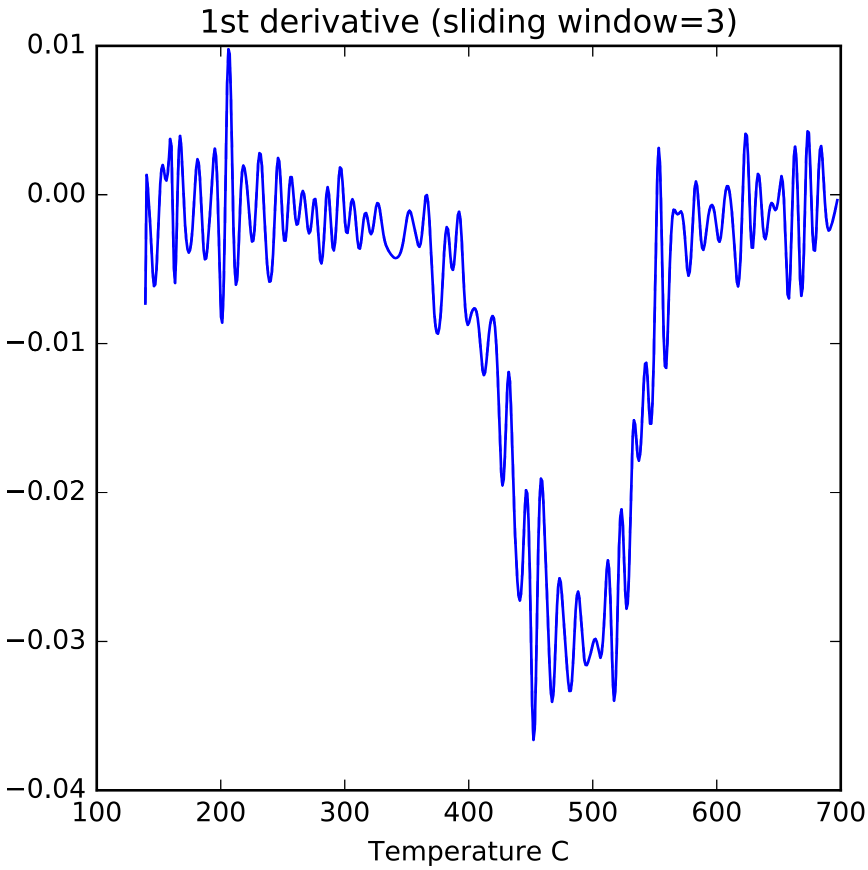
Go to Top

16 Curie temperature data

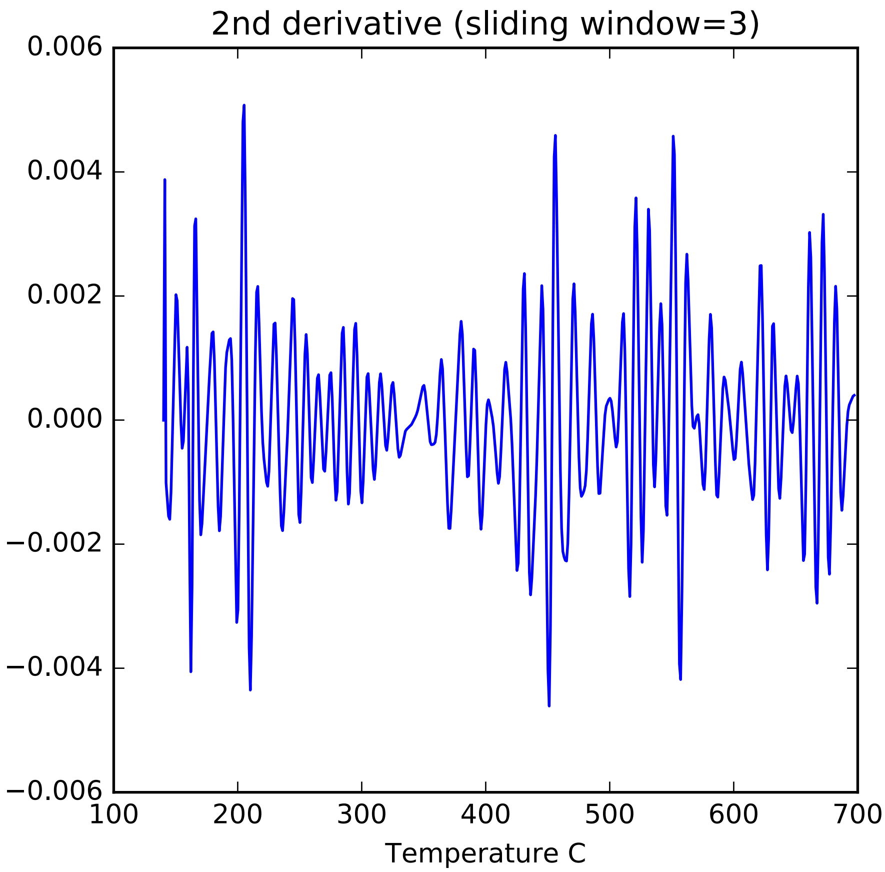
```
In [31]: ipmag.curie(path_to_file='./Additional_Data/curie/',
                     file_name='curie_example.dat', save=True,
                     save_folder='Additional_Notebook_Output/curie/')
```

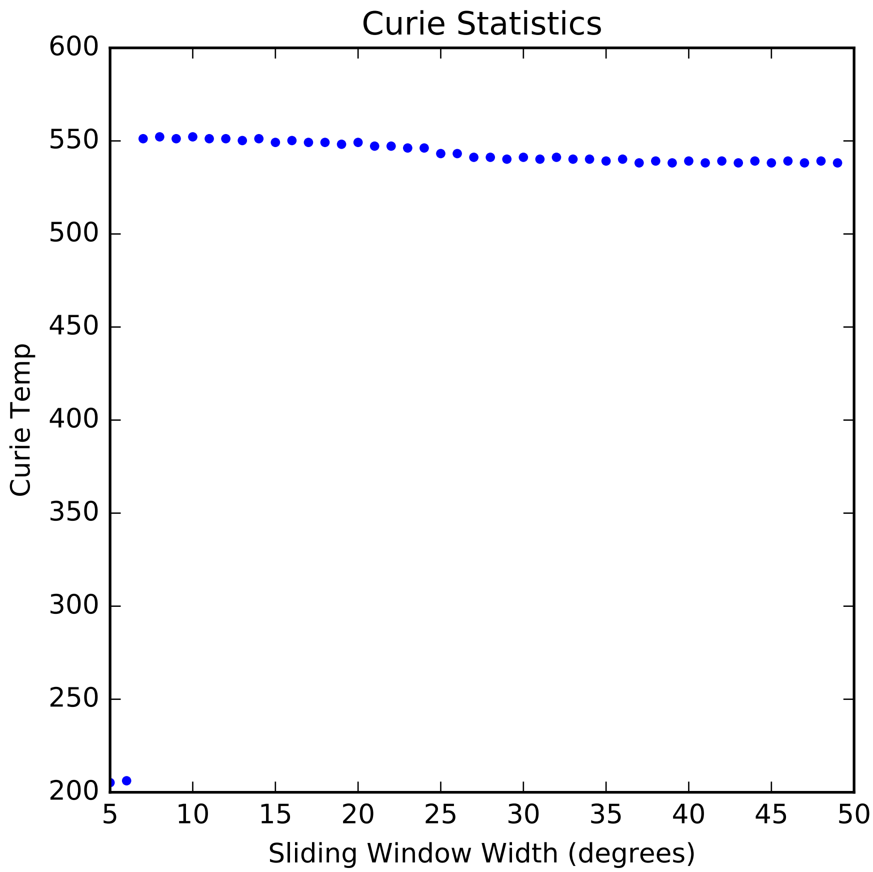
```
second deriative maximum is at T=205
```


M-T (sliding window=3)
Magnetization
Temperature C
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
100
200
300
400
500
600
700



1st derivative (sliding window=3)
0.01
0.00
−0.01
−0.02
−0.03
−0.04
100
200
300
400
500
600
700
Temperature C


2nd derivative (sliding window=3)
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
100
200
300
400
500
600
700
Temperature C

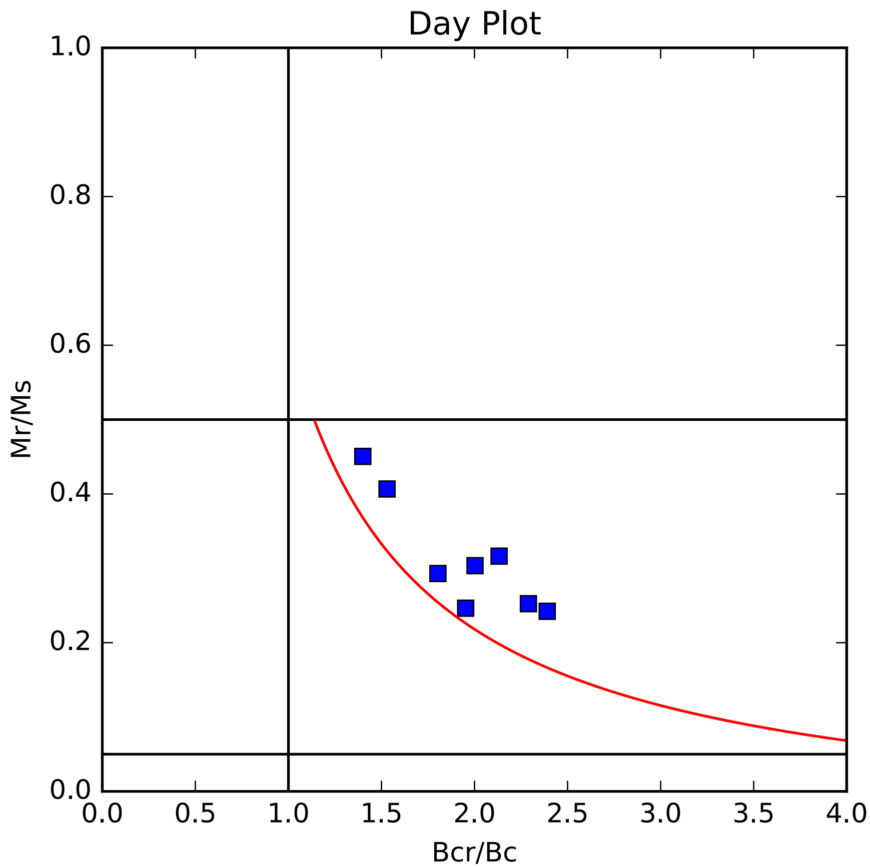


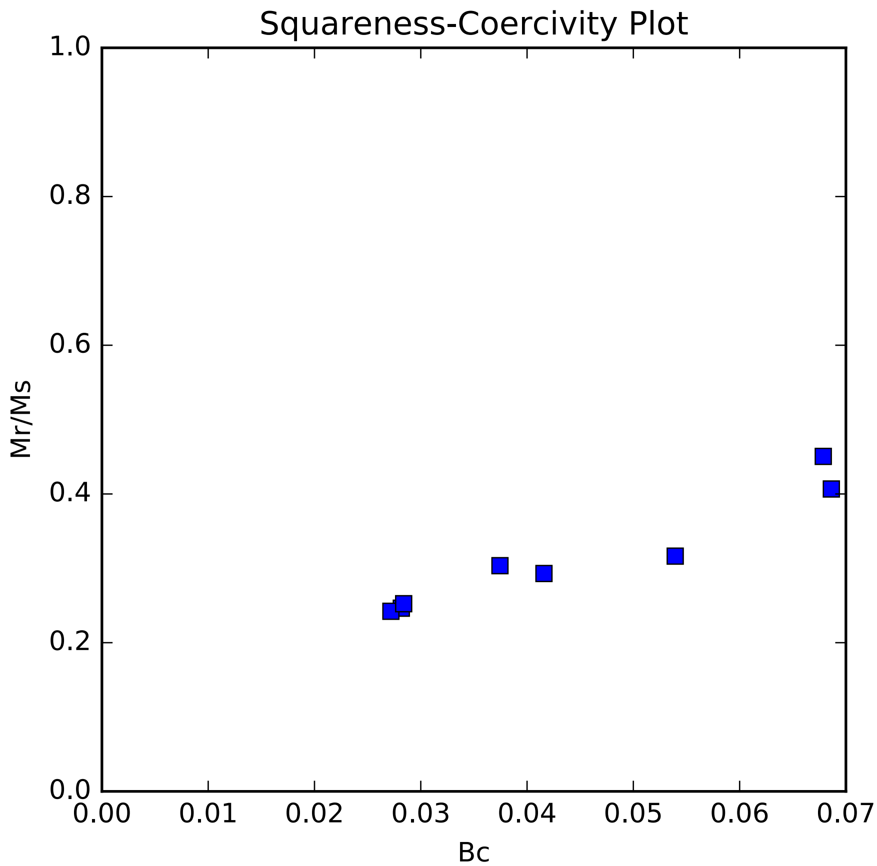
Go to Top

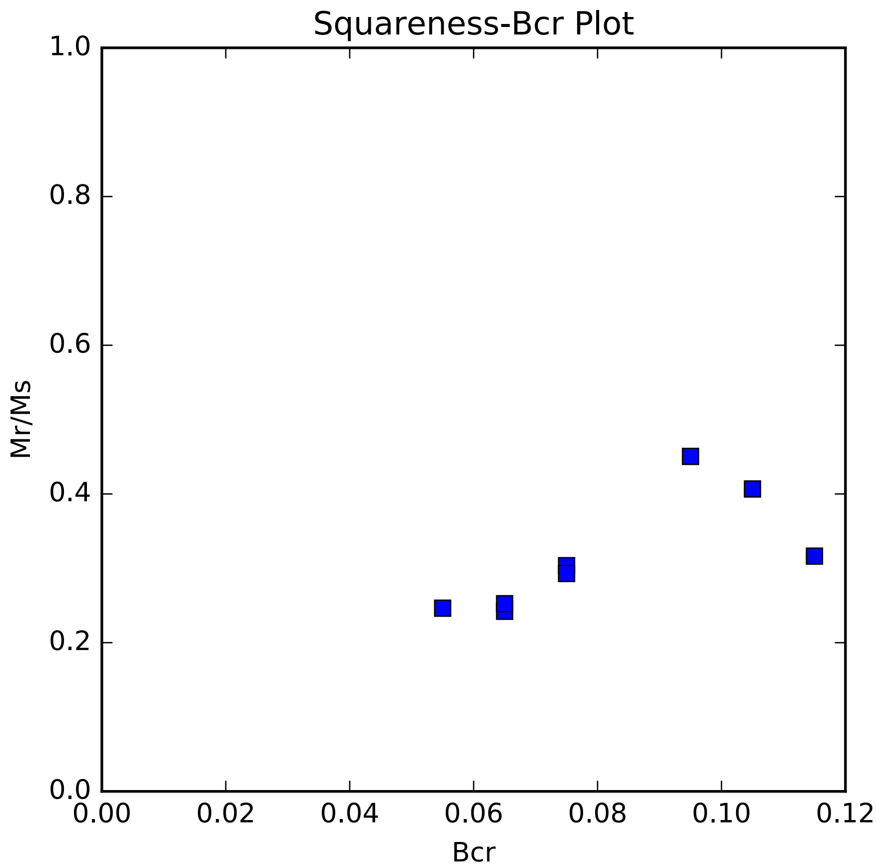
17 Day plots

Here we demonstrate the function **ipmag.dayplot**, which creates Day plots, squareness/coercivity and squareness/coercivity of remanence diagrams using hysteresis data.

```
In [32]: ipmag.dayplot(path_to_file='./Additional_Data/dayplot_magic/',
                       hyst_file='dayplot_magic_example.dat',
                       save=True,save_folder='Additional_Notebook_Output/day_plots/');
```


Day Plot
Mr/Ms
Bcr/Bc
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Squareness-Coercivity Plot
Mr/Ms
Bc
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07




```
<matplotlib.figure.Figure at 0x114da9290>
```

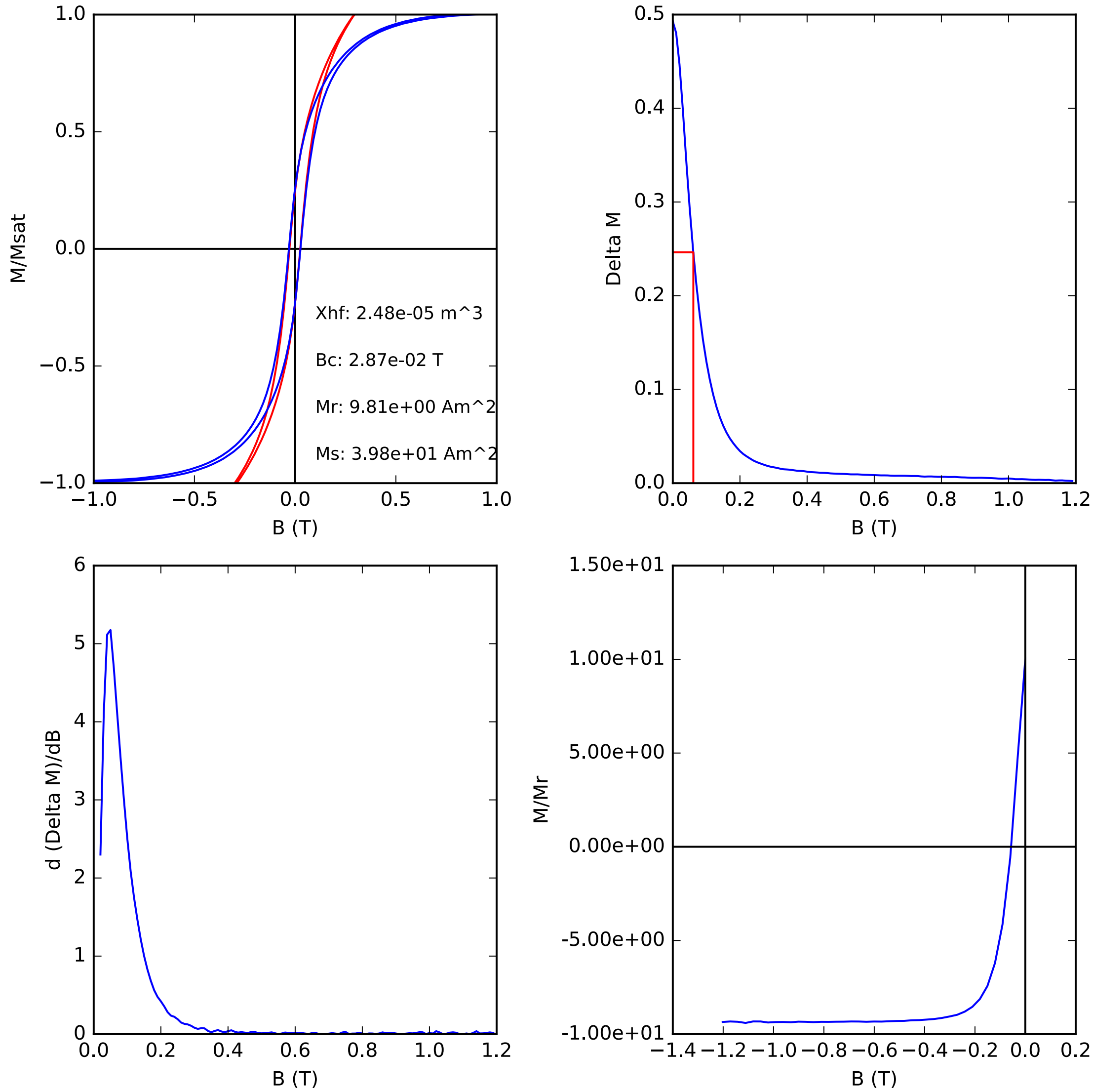
Go to Top

18 Hysteresis Loops

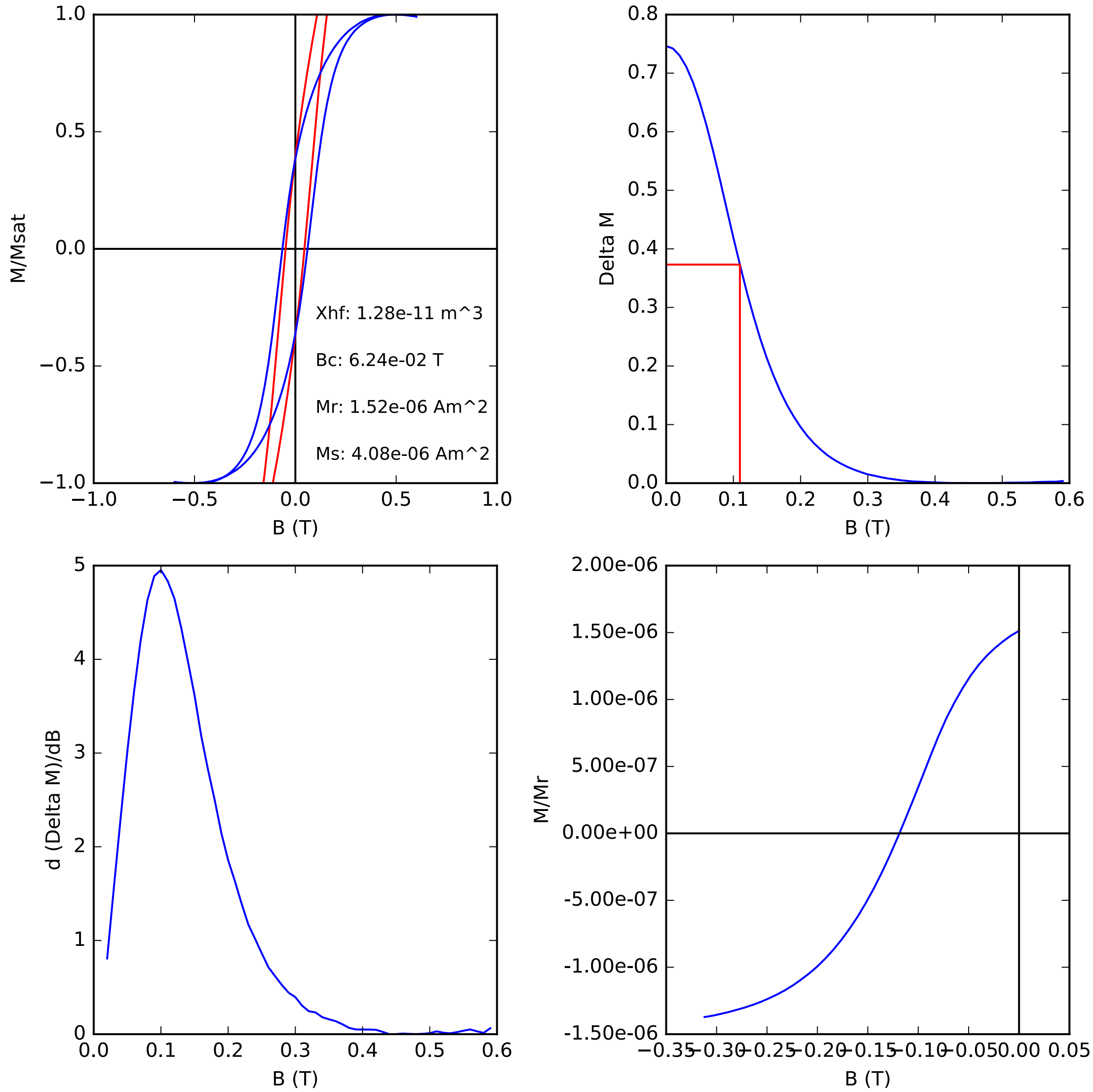
The function **ipmag.hysteresis_magic** also generates a set of hysteresis plots with data from a *magic_measurements* file.

```
In [33]: ipmag.hysteresis_magic(path_to_file='./Additional_Data/hysteresis_magic/',
                          hyst_file='hysteresis_magic_example.dat', save=True,
                         save_folder='./Additional_Notebook_Output/hysteresis')
```

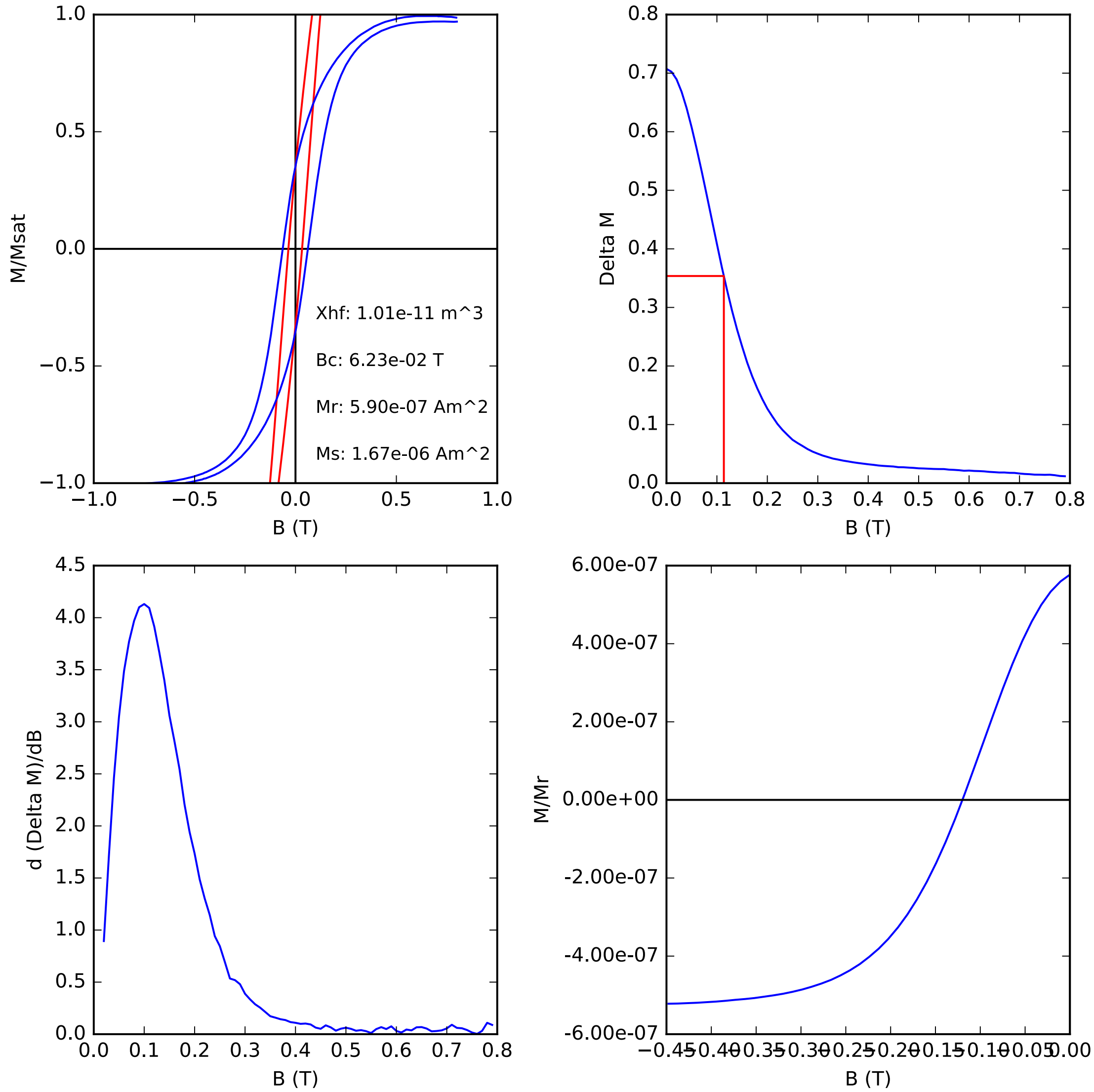
```
IS06a-1 1 out of  8
```



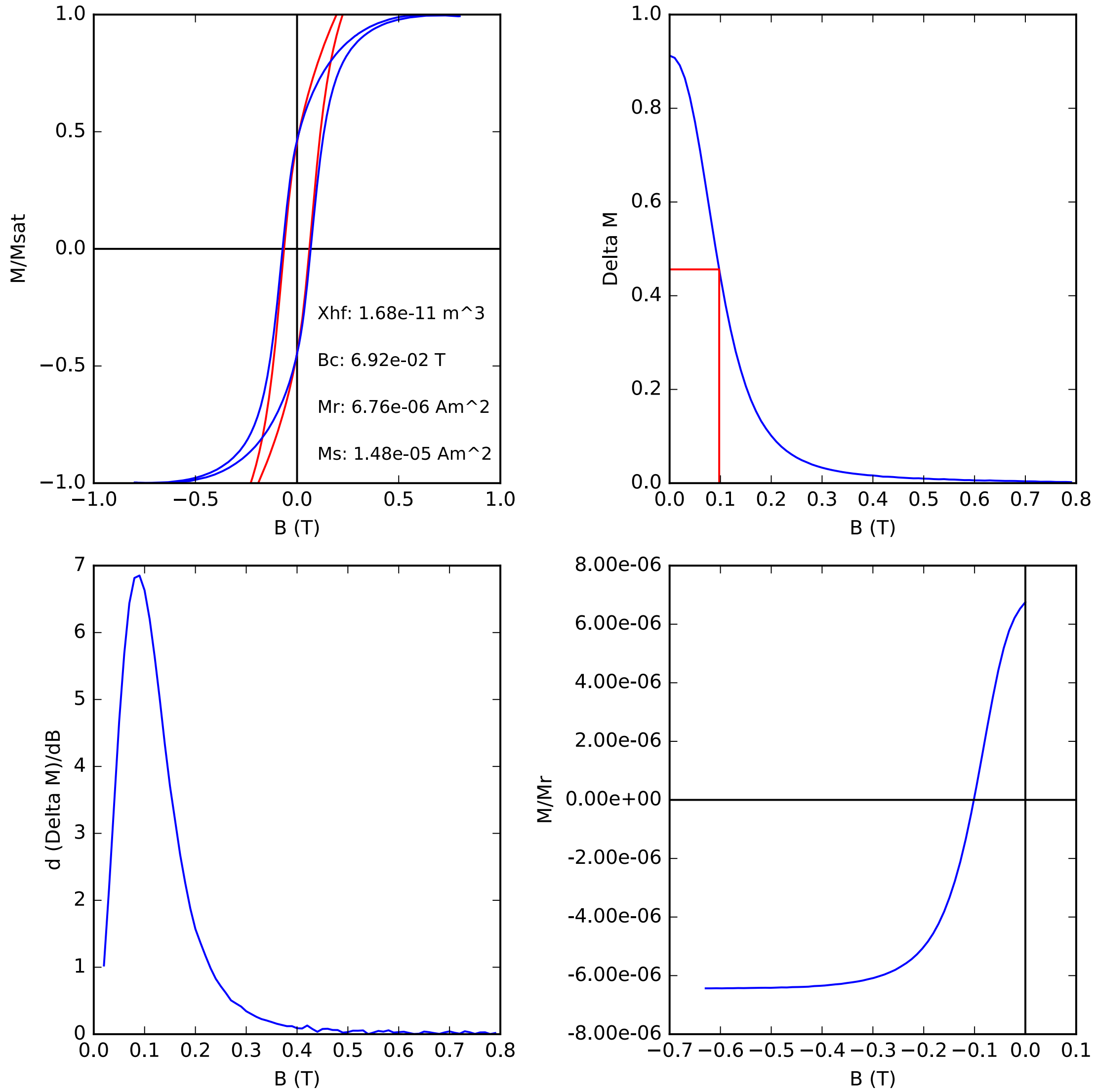
IS06a-2 2 out of 8



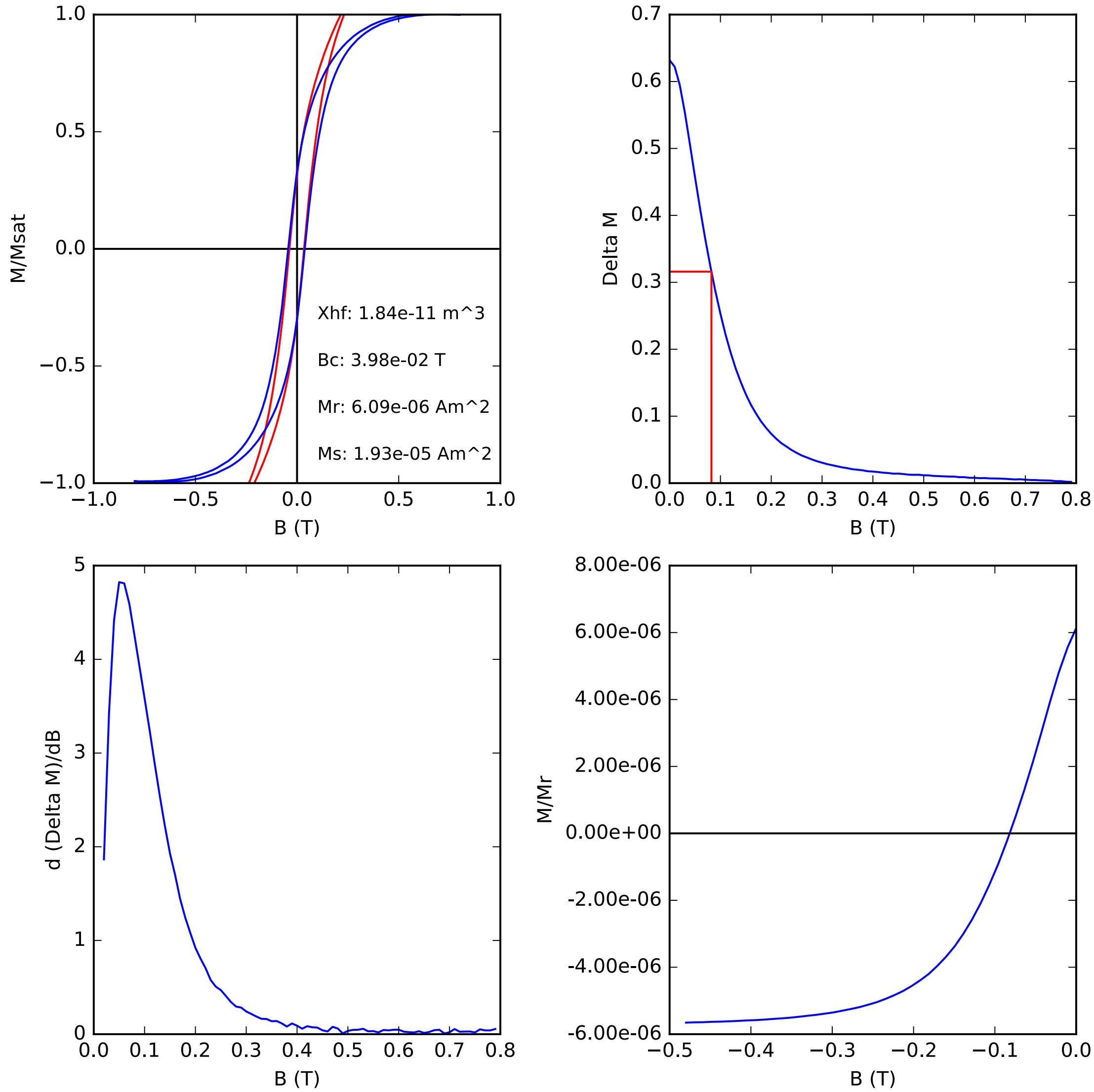
IS06a-3 3 out of 8



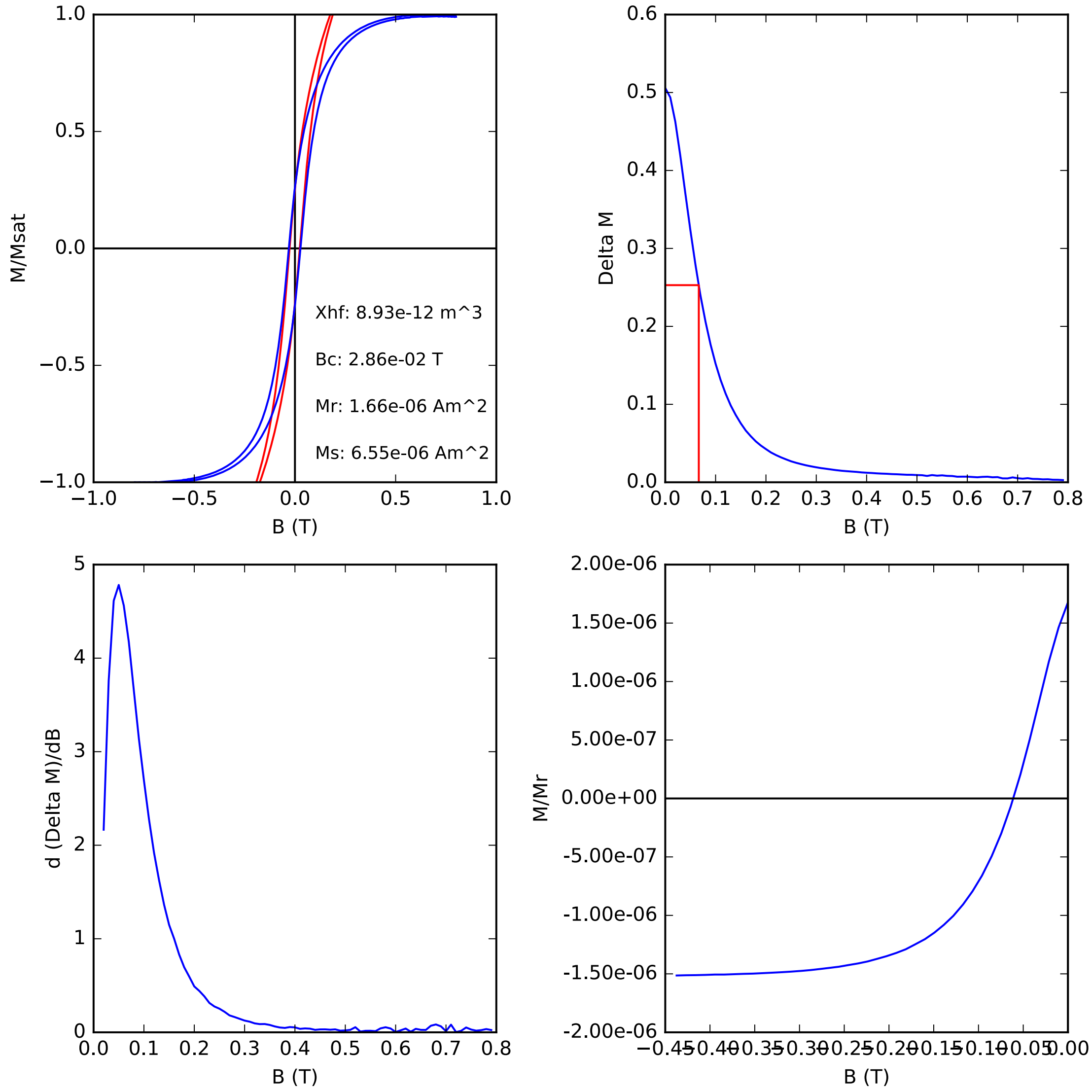
IS06a-4 4 out of 8



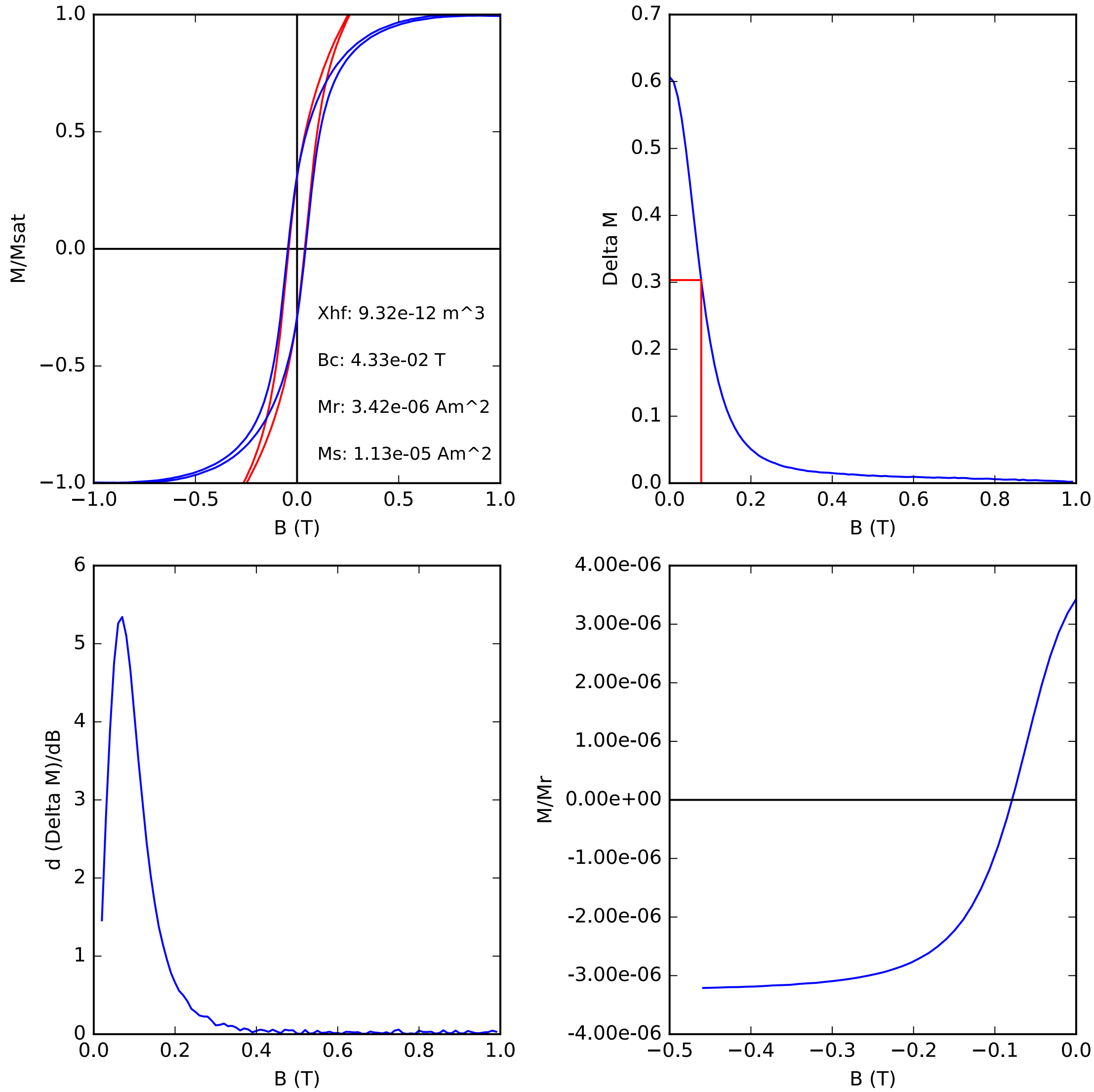
IS06a-5 5 out of 8



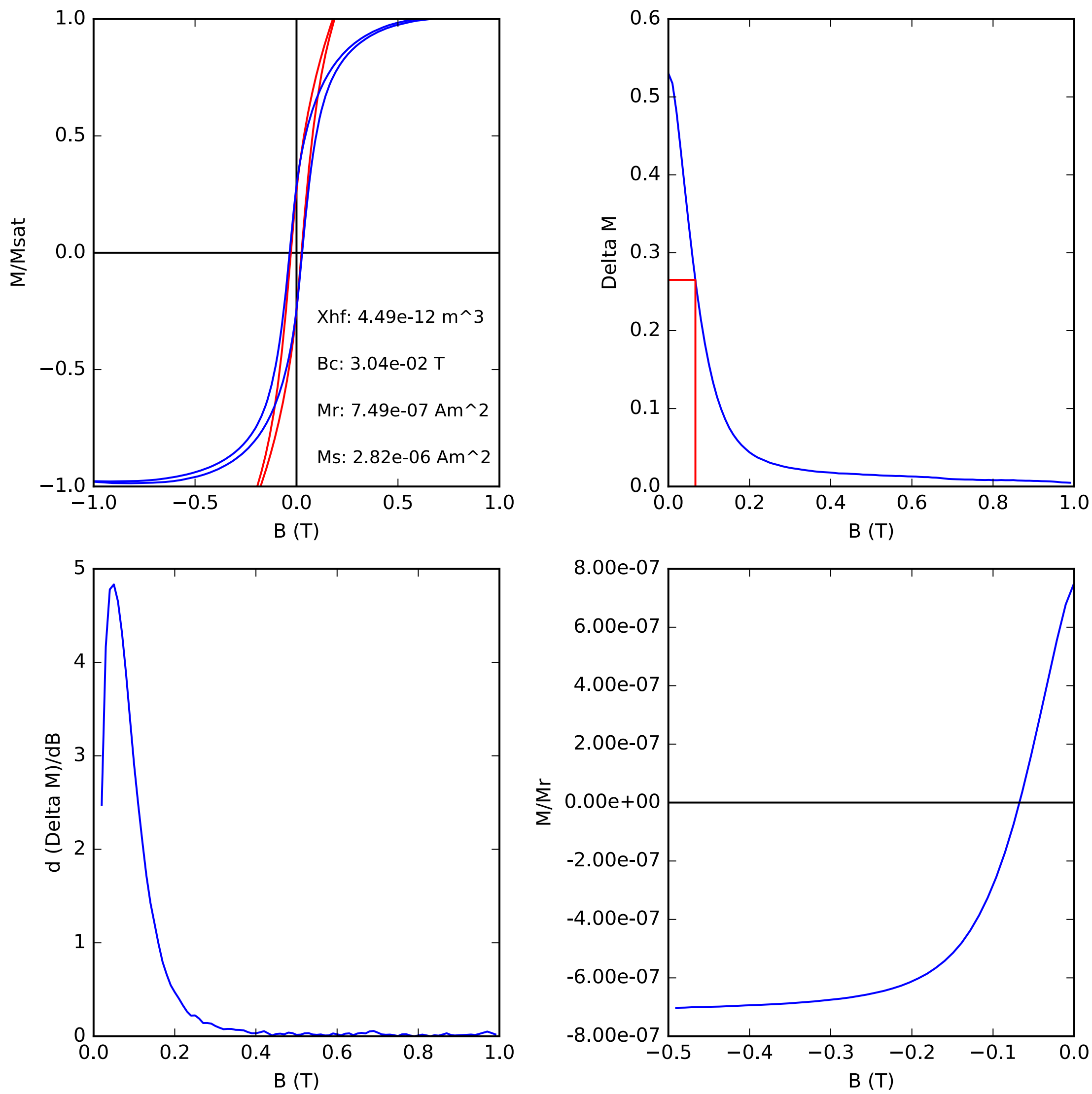
IS06a-6 6 out of 8



IS06a-8 7 out of 8



IS06a-9 8 out of 8



Go to Top

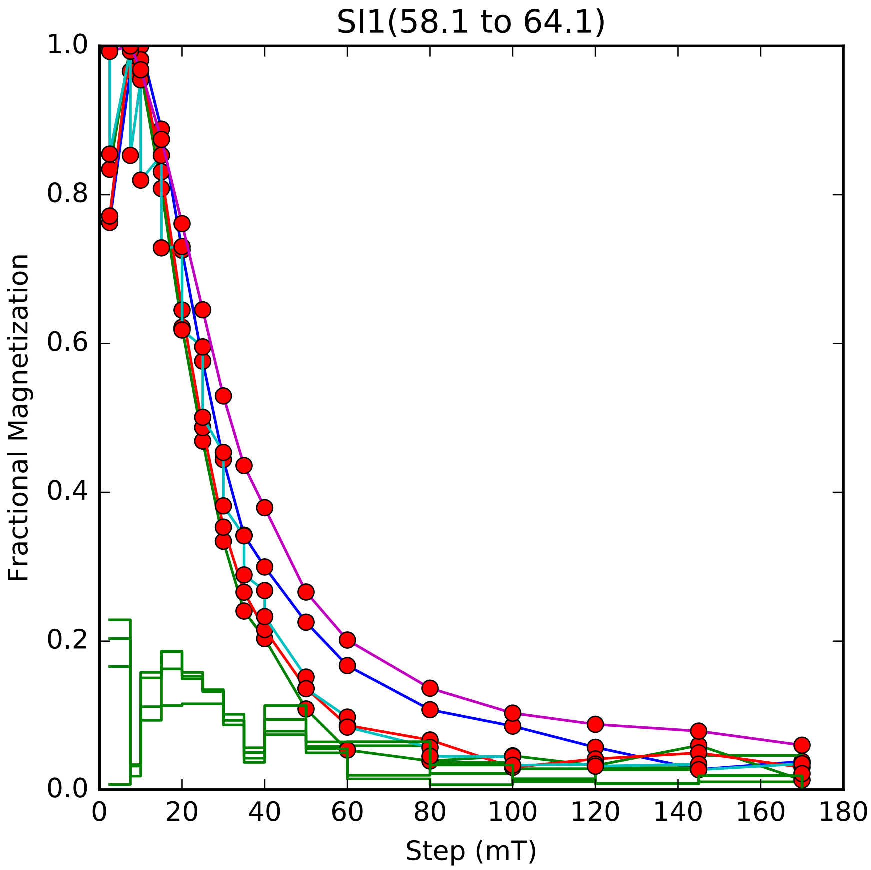
19 Demagnetization Curves

The function **ipmag.demag_magic** filters and plots demagnetization data. These data will be read and combined by expedition, location, site or sample according to the *plot_by* keyword argument. Alternatively, you can choose to plot each specimen measurement individually. By default, all plots generated by this function will be shown. If you only wish to plot a single subset of data, you can use the keyword argument *individual* to specify the name of the one site, location, sample, etc. that you would like to see.

Below, we use the *magic_measurements.txt* file of Swanson-Hysell et al., 2014 to plot demagnetization data by site. We then specify an individual site ('SI1(58.1 to 64.1)') that will plot alone. Like other functions, these plots can be optionally saved out of the notebook.

```
In [34]: ipmag.demag_magic(path_to_file='./Example_Data/Swanson-Hysell2014/',
                           plot_by='site', treat='AF', individual= 'SI1(58.1 to 64.1)')
```

```
13395  records read from  ./Example_Data/Swanson-Hysell2014/magic_measurements.txt
SI1(58.1 to 64.1) plotting by:  er_site_name
```



Go to Top

20 Interactive plotting

IPy Widgets are part of what makes the Jupyter notebook environment so powerful – these widgets allow user interaction with figures. We first demonstrate the use of the **interact** widget, imported below.

Note: If you do not have the ipywidgets package installed, you may choose to either install it through Anaconda or Enthought (depending on your Python distribution), manually install it (a bit more difficult), or simply skip the next few blocks of code. Below are quick installation instructions for those with either an Anaconda or Enthought Canopy distribution.

Installation on Anaconda

On the command line, enter

```
conda install ipywidgets
```

Make sure this installs within the Python 2 environment (if you have Python 3 as your default environment).

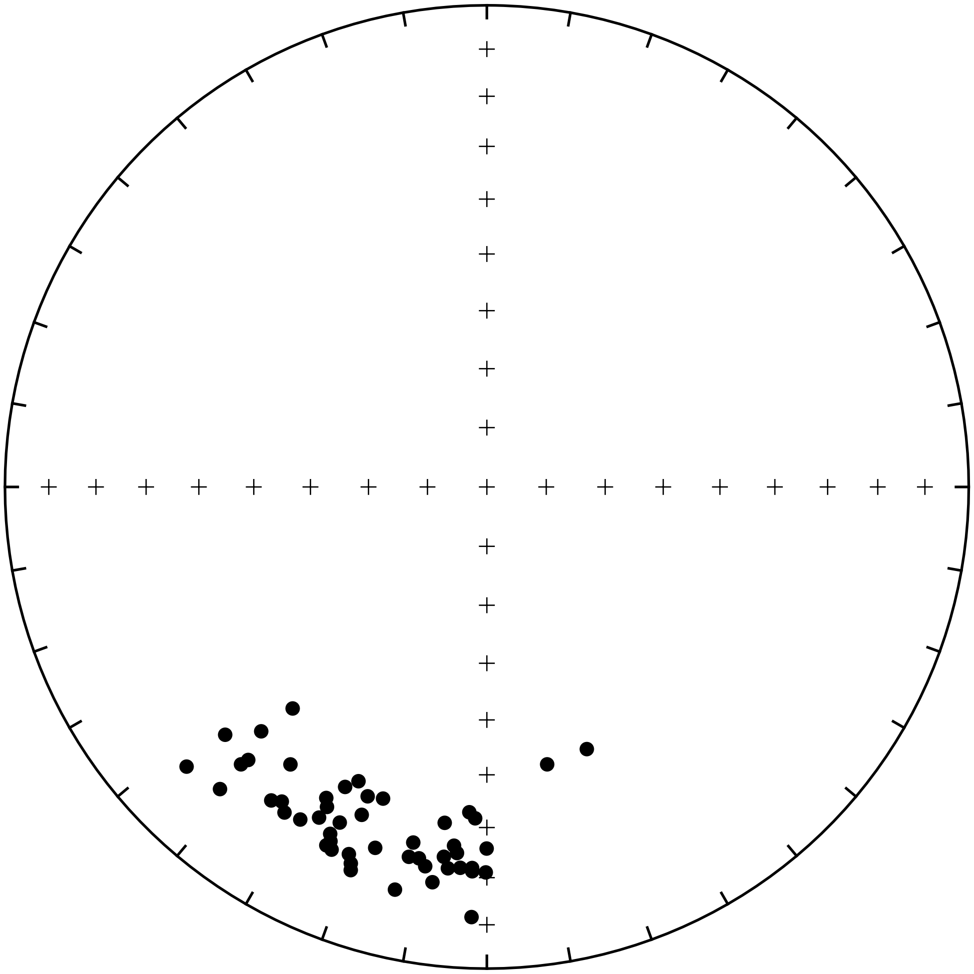
Installation on Enthought Canopy

Open the Canopy application and navigate to the Package Manager. Search for and install ipywidgets.

```
In [35]: from ipywidgets import interact
```

The **interact** widget allows adjustable values (within specified bounds) to all keyword arguments of a function. It can be used as a wrapper function, as seen below. Here we create a new function, **squish_interactive**, which streamlines the **ipmag.squish** function and automatically inputs the fisher-distributed directions created at the beginning of the notebook. This new function also allows us to reduce the keyword arguments to the *factor* variable, which is the only value we want to be actively adjustable. Finally, to make the **squish_interactive** function interactive in the notebook, we "wrap" this function with **@interact** placed directly above our new function.

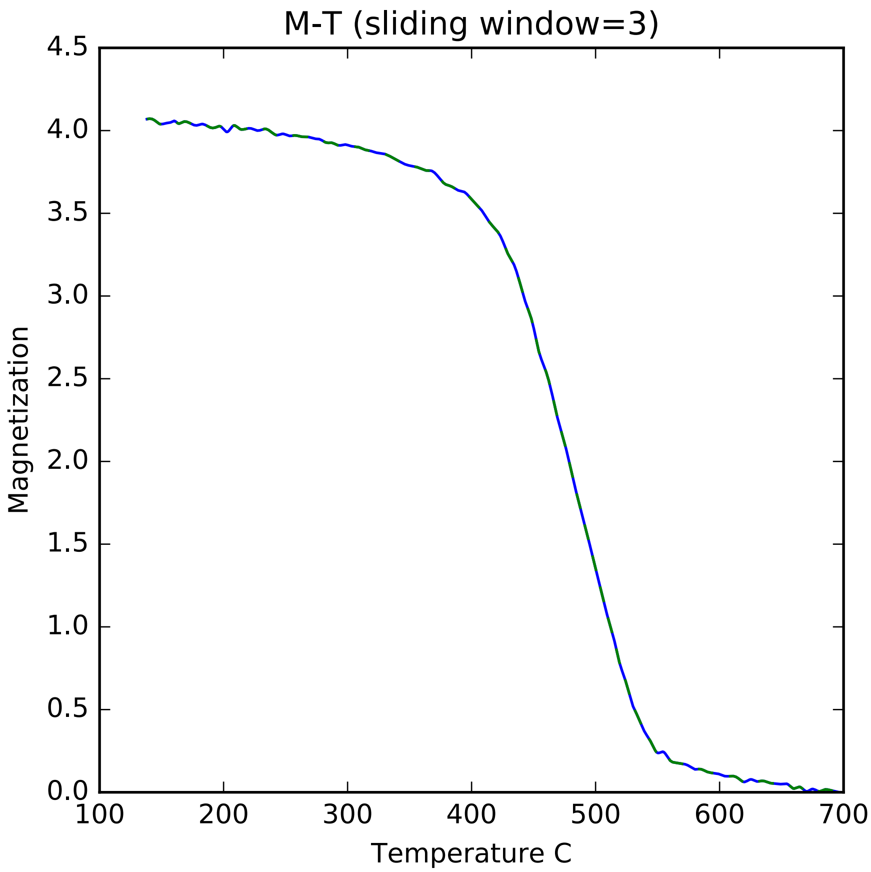
```
In [36]: @interact
         def squish_interactive(flattening_factor=(0.,1.,.1)):
             squished_incs = []
             for inclination in inclinations:
                 squished_incs.append(ipmag.squish(inclination, flattening_factor))

             # plot the squished directional data
             plt.figure(num=1,figsize=(6,6))
             ipmag.plot_net(1)
             ipmag.plot_di(declinations,squished_incs)
```

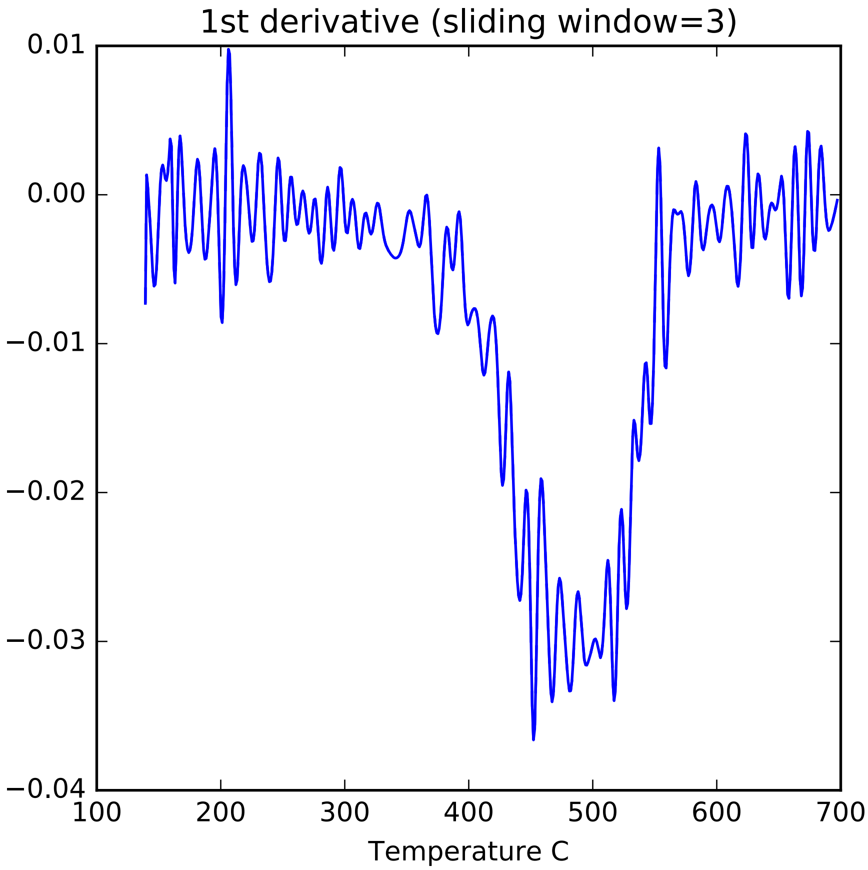


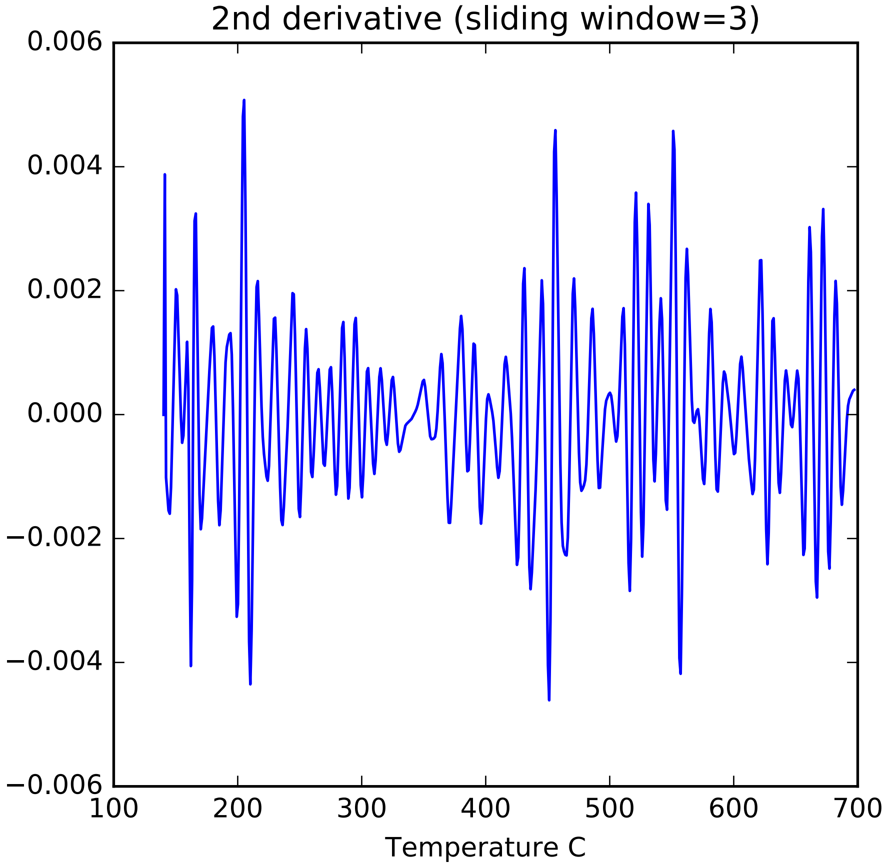
interact can also be used as a regular function call – the name of the interactive function is passed as the first argument, followed by the adjustable keyword arguments. Below, we demonstrate passing the *curie* function's parameters to **interact**.

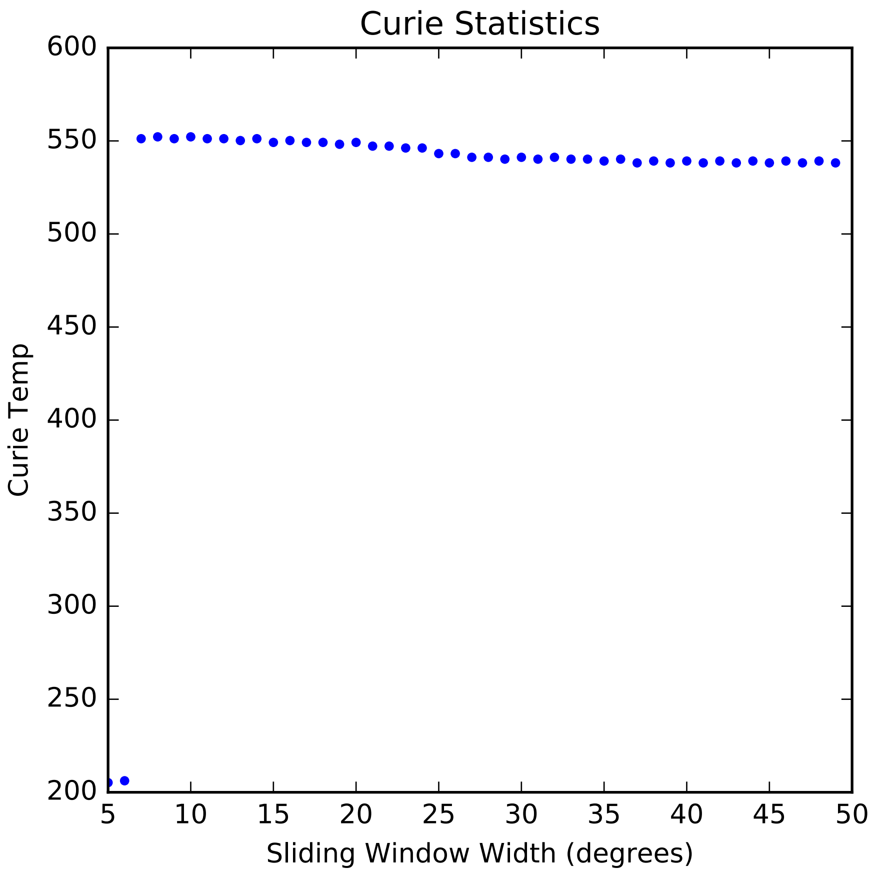
```
In [37]: interact(ipmag.curie, path_to_file='./Additional_Data/curie/',file_name='curie_example.dat',wi
```

```
second deriative maximum is at T=205
```


M-T (sliding window=3)
Magnetization
Temperature C
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
100
200
300
400
500
600
700


1st derivative (sliding window=3)
0.01
0.00
−0.01
−0.02
−0.03
−0.04
100
200
300
400
500
600
700
Temperature C


2nd derivative (sliding window=3)
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
100
200
300
400
500
600
700
Temperature C


Curie Statistics
Curie Temp
600
550
500
450
400
350
300
250
200
5
10
15
20
25
30
35
40
45
50
Sliding Window Width (degrees)

Go to Top

In [ ]: