

METEORFINDER DEMO

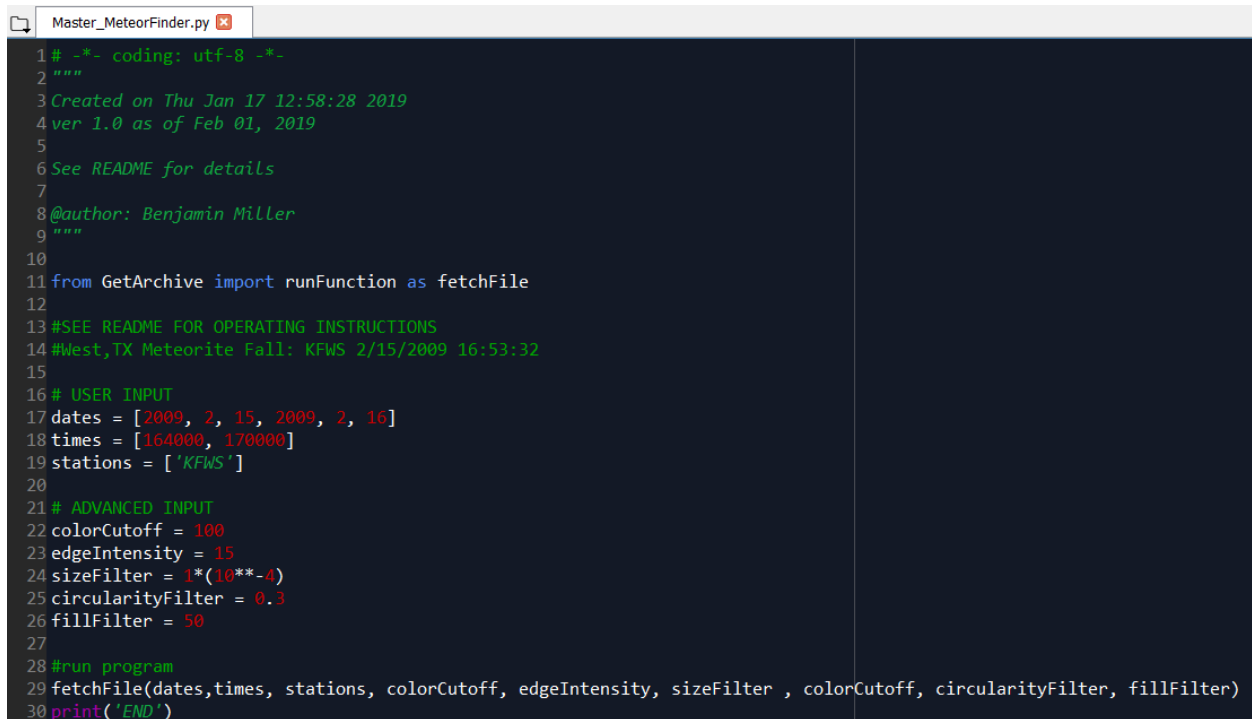
MeteorFinder ver 0.2
06/12/2019
Benjamin Miller

This file is intended to provide a visual supplement to the base README file. For a discussion of the installation requirements, processes, and variables, please see the README.

Starting the Program

The *Master_MeteorFinder.py* function serves as the primary user interface for detecting meteorite phenomena. This demonstration assumes that the user has previously installed all required packages and understands the operation of standard python IDE software such as Spyder.

After opening *Master_MeteorFinder.py*, the user will see the code presented in Figure 1. Here, the user inputs are delineated between standard and advanced variables. First, the user selects a date, time, and station range for fetching radar data. More advanced users may adjust the filters used for phenomena identification, which provides a higher level of control on narrow data ranges but is beyond the scope of this demonstration.

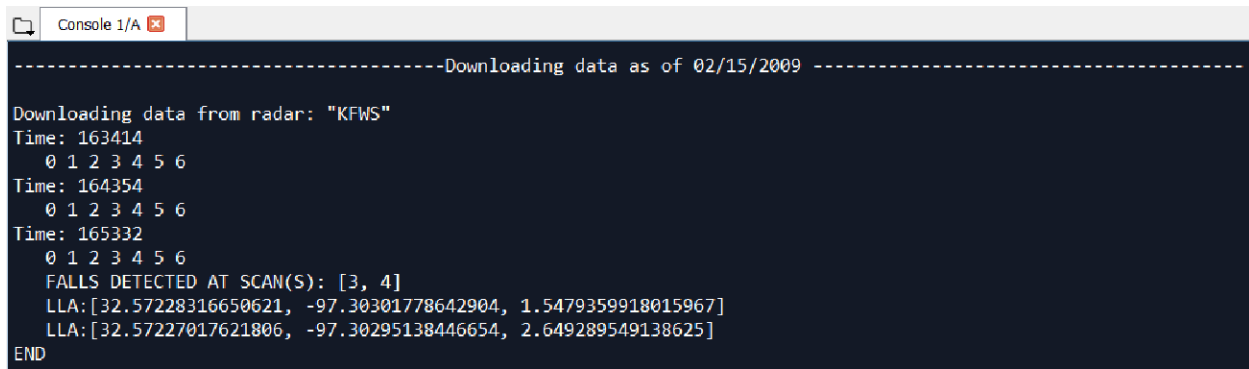


```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jan 17 12:58:28 2019
4 ver 1.0 as of Feb 01, 2019
5
6 See README for details
7
8 @author: Benjamin Miller
9 """
10
11 from GetArchive import runFunction as fetchFile
12
13 #SEE README FOR OPERATING INSTRUCTIONS
14 #West,TX Meteorite Fall: KFWs 2/15/2009 16:53:32
15
16 # USER INPUT
17 dates = [2009, 2, 15, 2009, 2, 16]
18 times = [164000, 170000]
19 stations = ['KFWs']
20
21 # ADVANCED INPUT
22 colorCutoff = 100
23 edgeIntensity = 15
24 sizeFilter = 1*(10**-4)
25 circularityFilter = 0.3
26 fillFilter = 50
27
28 #run program
29 fetchFile(dates,times, stations, colorCutoff, edgeIntensity, sizeFilter , colorCutoff, circularityFilter, fillFilter)
30 print('END')
```

Figure 1: *Master_MeteorFinder.py* interface

On running the master function, the console will display output to the console as shown in Figure 2. The algorithm will process the requested data in chronological dates, processing each radar site alphabetically for the entire time range.

In the figure, which has been selected for the example radar fall event in West, TX, two time files have been processed. For the first two time files, no fall events were detected. This will be the predominant output of the system, showing only the progressive altitude layers as they are unwrapped. For the time file at 16:53:32, two fall phenomena have been detected at the 3rd and 4th altitude cut. The latitude, longitude (decimal degrees), and altitude (kilometers) for both phenomena are provided.



```
-----Downloading data as of 02/15/2009 -----
Downloading data from radar: "KFWS"
Time: 163414
 0 1 2 3 4 5 6
Time: 164354
 0 1 2 3 4 5 6
Time: 165332
 0 1 2 3 4 5 6
FALLS DETECTED AT SCAN(S): [3, 4]
LLA: [32.57228316650621, -97.30301778642904, 1.5479359918015967]
LLA: [32.57227017621806, -97.30295138446654, 2.649289549138625]
END
```

Figure 2: *MeteorFinder* console output for KFWS 02/15/2009 16:30:00-17:00:00

Analyzing Output

Whenever a fall has been detected by the program, it saves the unwrapped images of the identified phenomena, as well as the .gz file for the full radar scan. In Figure 3, a new site-specific folder has been created in which these saved files are stored. Note that if the program is canceled at any time, the program will not be able to delete temporary data, and a residual folder will remain.

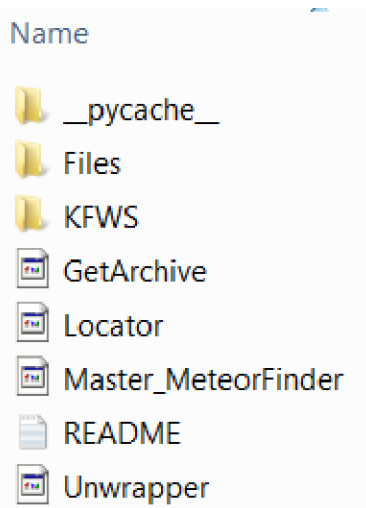


Figure 3: Folder created for a fall identified by KFWS

Within the KFWS folder, the user should see two files available. These files, beginning with a tag representing the site, data, local time, and scan format, display the identified phenomena with regards to radial velocity (Figure 4a) and spectrum width (Figure 4b). The blue contours represent where the fall has been identified, and the other color coding represents magnitudes of the respective data. These files may be useful in confirming the validity of the meteor fall, and the advanced user may troubleshoot variable settings.

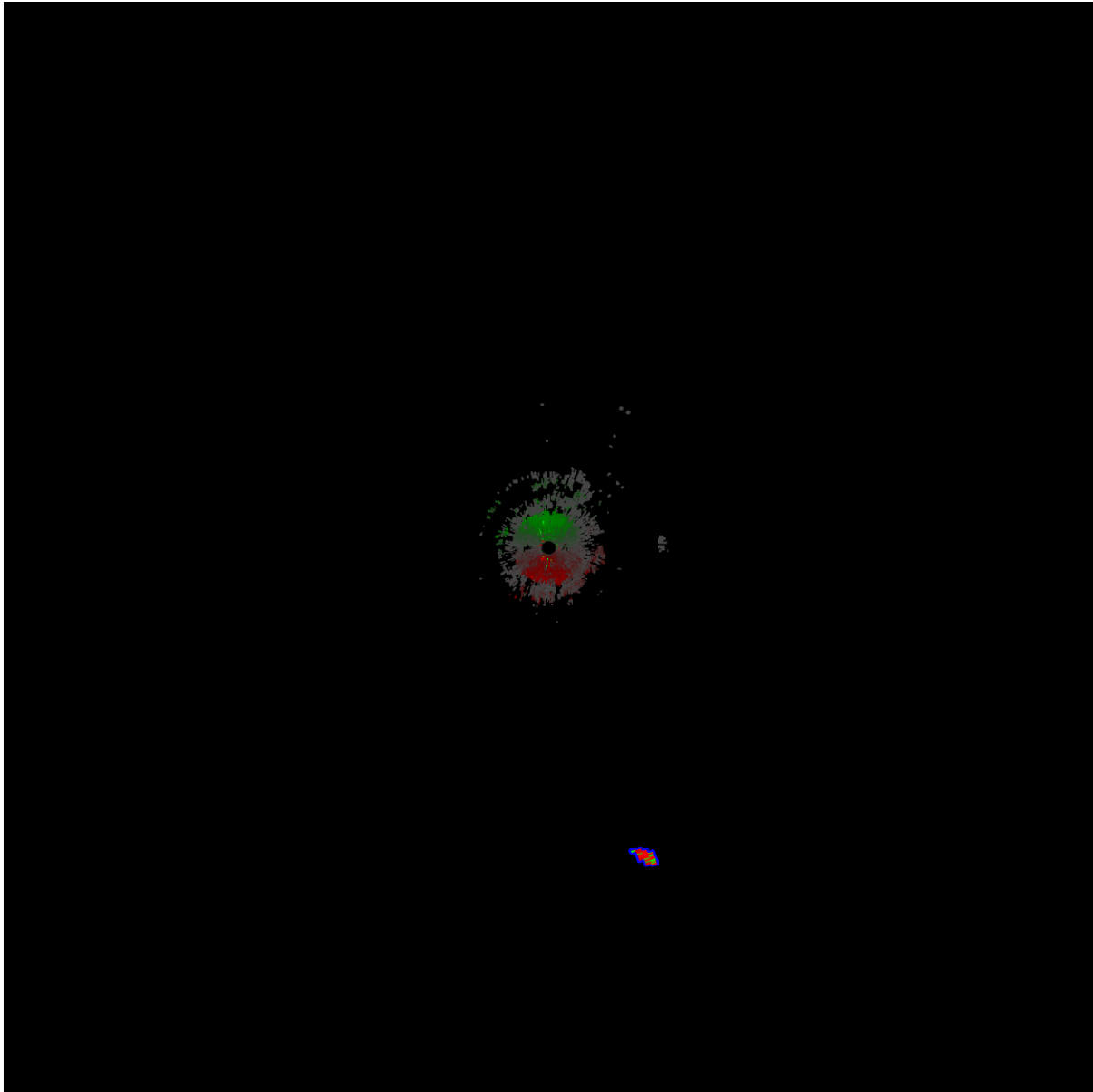


Figure 4a: *KFWS20090215_165332_V03_VEL4.png*

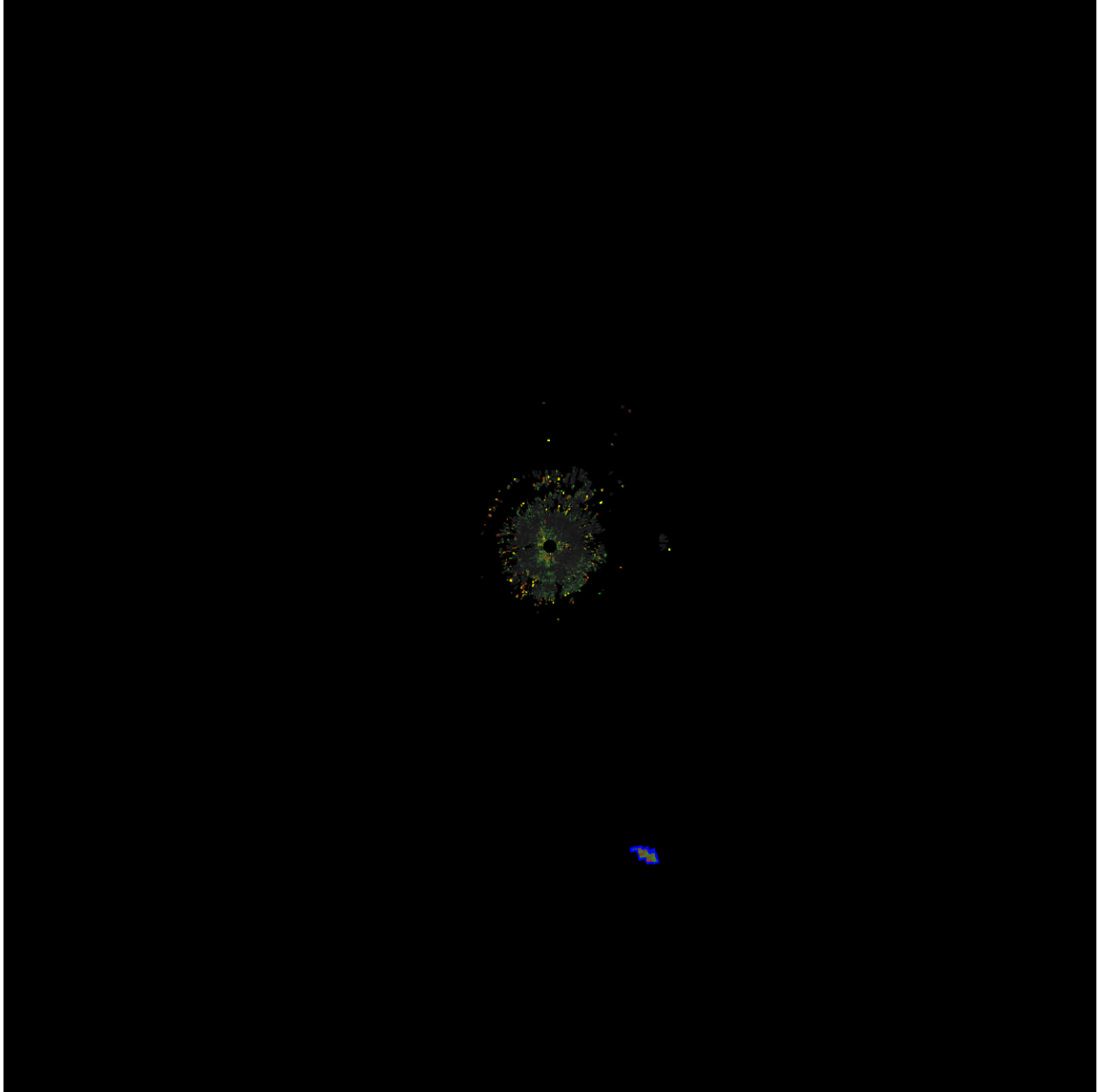


Figure 4b: *KFWS20090215_165332_V03_SPW4.png*

Advanced Variables

Advanced input gives the user a level of control over the *Locator.py* subfunction, and a brief summary of these variables is provided in the README. For the advanced user, this demonstration provides an overview on how these variables affect the code. A basic understanding of the OpenCV package will help in the user's understanding.

The *colorCutoff* provides a way to ignore low-magnitude events from consideration. Data is stored as red-green color channels by the Unwrapper, ranging from 0 to 255 for the maxima, and so the filtering follows the two-step process shown in Figure 5.

Changing the variable will control the magnitude of data relative to the maxima that is permitted to pass for both radial velocity and spectral width analysis. This is done for each level, excluding the lowermost quarter scans, which are more susceptible to weather noise.

```
for x in pyRadarData['velocity']:
    if velCounter < scanHt/4:
        velCounter = velCounter + 1
        continue
    img = np.copy(x)
    #filter image color content
    gfilt = cv2.inRange(img, np.array([0,colorIntensity,0]), np.array([255,255,255]))
    rfilt = cv2.inRange(img, np.array([0,0,colorIntensity]), np.array([255,255,255]))
    imgMask = (gfilt+rfilt)>0
    del gfilt, rfilt
    #filter false-positives close to station
    masksize = 200
    circ = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (masksize,masksize))
    overlay = np.zeros(imgMask.shape)
    lo = 900 - np.int_(masksize/2)
    hi = 900 + np.int_(masksize/2)
    overlay[lo:hi,lo:hi] = circ
    overlay = overlay.astype(np.bool_, copy=False)
    imgMask[overlay] = 0
    del circ, overlay
    #Apply masks to image
    imgOrig = np.copy(img)
    img[:,0,:] = 0
    imgMask = np.expand_dims(imgMask,axis=1)
    imgMask = np.tile(imgMask, (1,1,1))
    img[imgMask] = imgOrig[imgMask]
    img[:,0,:] = 0
    img = img.astype(np.float32, copy=False)
    img = img / 255
    del imgOrig, imgMask
```

Figure 5: *colorCutoff* implementation

The *edgeIntensity* is concerned only with velocity processing, and relates to the color gradient calculation shown in Figure 6a. This derivative algorithm allows for exploitation of the wind shear produced by falling bodies, but is also sensitive to weather. A solution to weather is in collapsing all altitude data into a single scan, as in Figure 6b, as weather across multiple scans will be removed as a single object and dispersed fall phenomena will be enhanced.

```
#Calculate color (ie velocity) gradients
yfilter = np.array([[1.,2.,1.],[0.,0.,0.],[-1.,-2.,-1.]])
xfilter = np.transpose(yfilter)
gx = cv2.filter2D(img[:,0,1],-1,xfilter)
rx = cv2.filter2D(img[:,0,2],-1,xfilter)
gy = cv2.filter2D(img[:,1,1],-1,yfilter)
ry = cv2.filter2D(img[:,1,2],-1,yfilter)
Jx = gx**2 + rx**2
Jy = gy**2 + ry**2
Jxy = gx*gy + rx*ry
D = np.sqrt(np.abs(Jx**2 - 2*Jx*Jy + Jy**2 + 4*Jxy**2)+1)
e1 = (Jx + Jy + D) / 2
img = np.sqrt(e1*20)
del gx,rx,gy,ry,Jx,Jy,Jxy,D,e1
img = img.astype(np.uint8, copy=False)
img = cv2.threshold(img,edgeIntensity,255,cv2.THRESH_BINARY)[1]
edgeArray.append(img)
velCounter = velCounter + 1
```

Figure 6a: Edge detection between color channels

```
#Collapse data
edgeShape = np.shape(edgeArray[0]);
edgeSum = np.zeros((edgeShape[0],edgeShape[1]))
for img in edgeArray:
    edgeSum = edgeSum+img
```

Figure 6b: *edgeIntensity* implementation

In effect, the above code controls the production of binary objects which can now be detected and formatted for processing as an array through the following filters, as shown

in Figure 7. These detected “contours” track the pixel content of each individual body, and both velocity and spectrum analysis begin with removing small noise bodies. The *sizeFilter* input is actually a percentage of the full image area used to define the minimum size below which bodies are ignored.

The *circularityFilter* further filters contours by calculating an eccentricity-like term which removes spread-out features. Such features can form by small, noisy weather effects occurring close enough together to avoid the *sizeFilter*, and so this term serves to add robustness. The implementation must be relaxed when looking for distributed fall events, but this adjustment can increase false positives.

Finally, the *fillFilter* is uniquely related to the velocity edge-detection methods. Significant wind shear will cause edges to form solid objects, ideally as fully filled contours. To define this process in OpenCV, the code creates a crop of the image to the bounding box around the contour, and then calculates the density of passable edges. As this bounding box is not the contour itself, the variable can never have a value of exactly 1.0, and will be substantially less for non-circular events.

```
img = img.astype(np.uint8, copy=False)
im2, contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
contourfix = []
for cnt in contours:
    #Area-based filtering
    area = cv2.contourArea(cnt)
    #Shape-based filtering
    (x,y), radius = cv2.minEnclosingCircle(cnt)
    aspect = area/(np.pi*radius**2)
    #Density-based filtering
    x,y,w,h = cv2.boundingRect(cnt)
    imageROI = img[y:y+h,x:x+w]
    total = cv2.countNonZero(imageROI) - cv2.arcLength(cnt,True)
    if area>0:
        filled = (total/area)*100
    else:
        filled=0
    #Apply filters to object set
    if (area>size_filter[0]) & (aspect>circularityFilter) & (filled>fillFilter):
        contourfix.append(cnt)
```

Figure 7: *sizeFilter*, *circularityFilter*, and *fillFilter* implementation

If a single reflectivity object is detected, the second processing step commences. The spectrum width processing ignores edges and performs object detection after color filtering and level collapsing. A 10% density-based filter is included, seen in Figure 8, to add robustness against weather effects, which will tend to surround each other in high-density formations.

```
im2, contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
contourfix = []
for cnt in contours:
    #Area-based filtering
    area = cv2.contourArea(cnt)
    #Shape-based filtering
    (x,y), radius = cv2.minEnclosingCircle(cnt)
    aspect = area/(np.pi*radius**2)
    #Anti-weather density-based filtering
    if (area>size_filter[0]) & (aspect>circularityFilter):
        x,y,w,h = cv2.boundingRect(cnt)
        imageROI = img[y-(1*h):y+(1*h),x-(1*w):x+(1*w)]
        total = cv2.countNonZero(imageROI) - cv2.arcLength(cnt,True)
        area = 1*h*1*w
        if (total/area)<.1:
            contourfix.append(cnt)
```

Figure 7: 10% density filter for spectrum width

To increase detections, after a single spectrum width body is detected, a 2-step “loosening” process reduces restrictions to increase the rate of detection, with false-positives reduced by the preceding steps. In Figure 8, a dilation filter is applied which aids in object recognition, and if no bodies are detected, e.g. due to dispersion, the filter is extended and the *circularityFilter* requirement is reduced.

```
while loopCheck==1:
    spwCounter=np.ceil(scanHt/4)
    for x in spwArray:
        #adaptability to loosen
        img = np.copy(x)
        img = img.astype(np.uint8)
        img = cv2.threshold(img,colorCutoff,255,cv2.THRESH_BINARY)[1]
        img = img.astype(np.uint8, copy=False)
        #Dilation permits greater capture range
        dilationScale = 2*errorCheck
        img = cv2.dilate(img, np.ones((dilationScale,dilationScale),np.uint8))
        #cv2.imwrite('spwCheck'+str(spwCounter)+'.png', img)
        #Repeats detection of spectrum width bodies
        img2,contours,hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        for cnt in contours:
            area = cv2.contourArea(cnt)
            (x,y), radius = cv2.minEnclosingCircle(cnt)
            aspect = area/(np.pi*radius**2)
            if (area>=size_filter[0]) & (aspect>circularityFilter/errorCheck): #note /errorCheck
                x,y,w,h = cv2.boundingRect(cnt)
                imageROI = img[y-(1*h):y+(1*h),x-(1*w):x+(1*w)]
                total = cv2.countNonZero(imageROI) - cv2.arcLength(cnt,True)
                area = 2*h*3*w
                spwrough = spwrough+1
                if (total/area)<.1:
                    spwIdentity.append(spwCounter)
                    spwLoosen.append(cnt)
                    loopCheck = 0
                #end loop after detections
            spwCounter = spwCounter + 1
        errorCheck = errorCheck+1
    if errorCheck>=10:
        #print('loosening loop error')
        break
```

Figure 8: Dilation filter and circularity reduction

To allow for further analysis, the algorithm provides data on the latitude, longitude, and altitude (LLA) of each detection (in decimal degrees and kilometers). In Figures 9a and 9b, the processes in Locator.py and Unwrapper.py respectively are shown. The Cartesian positions are based on a 100km scan radius provided by NOAA, and the projection to LLA follows a pyART-native azimuthal equidistant projection. Note that only the spectrum width data is presently used for extending detections, assuming the match from velocity data holds. In future releases, a matching algorithm will be introduced to add additional certainty.

```
for lev in idMatch:
    lev = lev.astype(np.uint8)
    #Instantiate plot data
    nametag = pth.splitext(name)[0]
    velCopy = pyRadarData['velocity'][lev].astype(np.uint8, copy=True)
    spwCopy = pyRadarData['spectrum_width'][lev].astype(np.uint8, copy=True)
    #Mark detections on data images
    velIm = cv2.drawContours(velCopy, spwLoosen[spwIdentity.index(lev)], -1, [255, 0, 0], thickness = 2) #Adjusted for spwIdentity track
    spwIm = cv2.drawContours(spwCopy, spwLoosen[spwIdentity.index(lev)], -1, [255, 0, 0], thickness = 2)
    #Record data images
    cv2.imwrite(nametag+'_VEL'+str(lev)+'.png', velIm)
    cv2.imwrite(nametag+'_SPW'+str(lev)+'.png', spwIm)
    #Calculate centroid positions
    M = cv2.moments(spwLoosen[spwIdentity.index(lev)])
    dim = np.shape(spwIm)[0]/2
    x = (int(M["m10"] / M["m00"])-dim)*100/dim
    y = (dim-int(M["m01"] / M["m00"]))*100/dim
    XY.append([x,y]) #km from instrument
```

Figure 9a: Cartesian position determination in Locator.py

```

if fallCount>0:
    fallId = list(map(int,fallId))
    print('\n FALLS DETECTED AT SCAN(S): '+str(fallId))
    lonlat0 = [radar.longitude['data'],radar.latitude['data']]
    lla = []
    for ind, r in enumerate(xy, start=0):
        lon,lat = pyart.core.cartesian_to_geographic_aeqd(r[0],r[1],lonlat0[0],lonlat0[1])
        dist = np.sqrt(np.abs(r[0])**2+np.abs(r[1])**2)
        el = radar.get_elevation(fallId[ind])
        alt = dist*np.tan(el[0]*np.pi/180)
        lla.append([lat.item(),lon.item(),alt])
    print(' LLA: '+str([lat.item(),lon.item(),alt]))

```

Figure 9b: LLA position determination in Unwrapper.py

Strewn Fields

This product is currently under development. Stay tuned for future updates.