

METEORFINDER DEMO

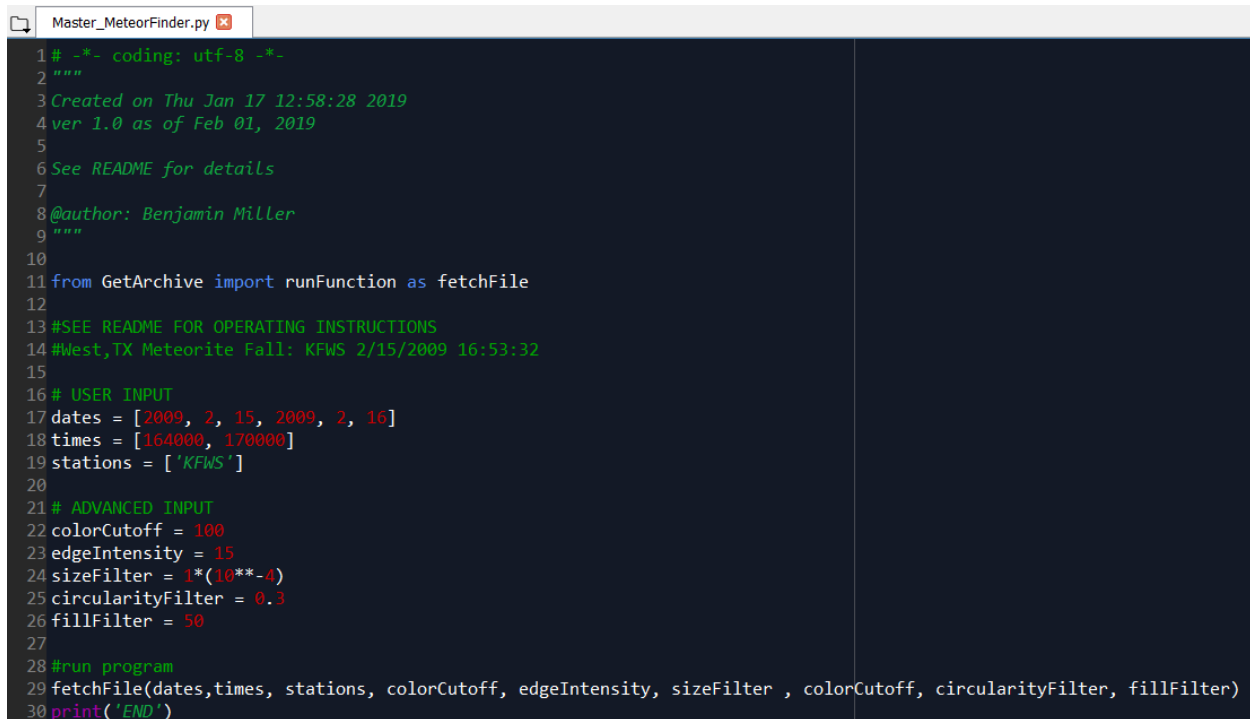
MeteorFinder ver 0.1
02/07/2019
Benjamin Miller

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.

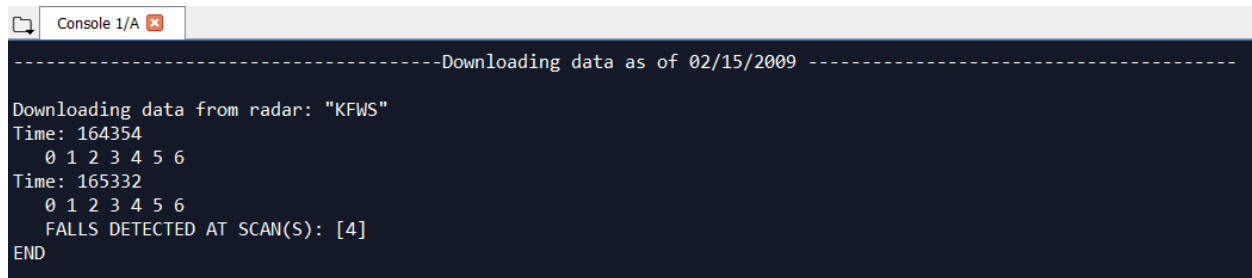


```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jan 17 12:58:28 2019
4 ver 1.0 as of Feb 01, 2019
5
6 See README for details
7
8 @author: Benjamin Miller
9 """
10
11 from GetArchive import runFunction as fetchFile
12
13 #SEE README FOR OPERATING INSTRUCTIONS
14 #West,TX Meteorite Fall: KFWs 2/15/2009 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 time file, at 16:43:53, 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, a single fall event has been detected on the 4th altitude layer of the radar scan.



```
-----Downloading data as of 02/15/2009 -----
Downloading data from radar: "KFWS"
Time: 164354
 0 1 2 3 4 5 6
Time: 165332
 0 1 2 3 4 5 6
FALLS DETECTED AT SCAN(S): [4]
END
```

Figure 2: *MeteorFinder* console output for KFWS 02/15/2009 16:40:00-17:00:00

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.

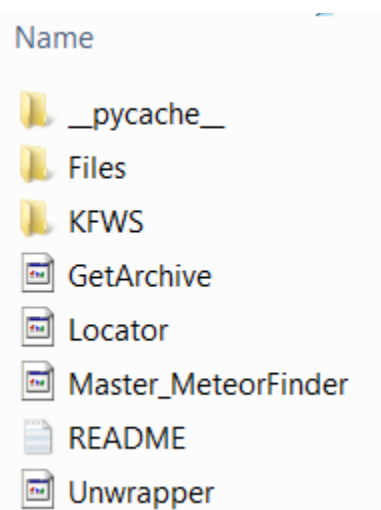


Figure 3: Folder created for a fall identified by KFWS

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.

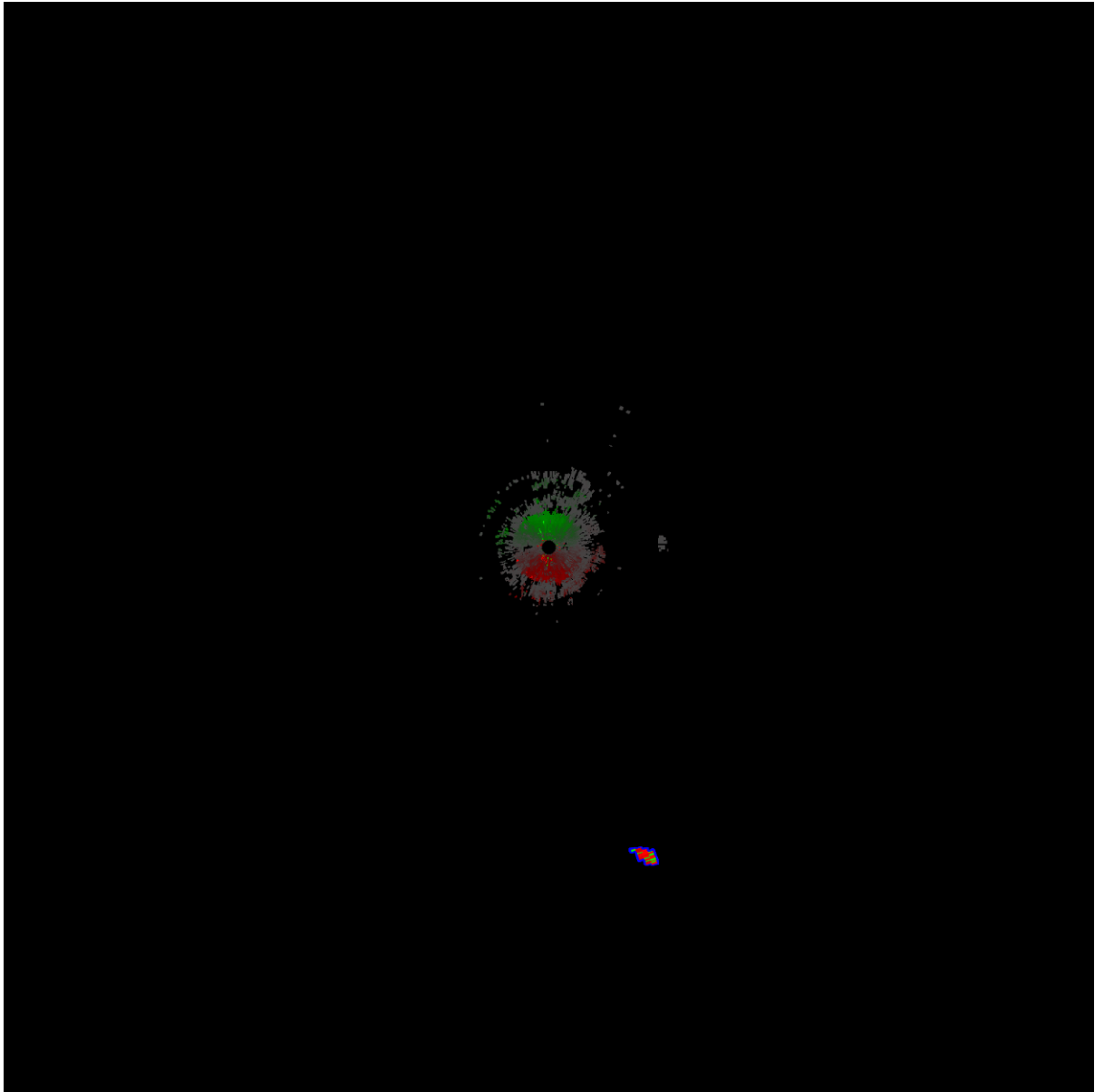


Figure 4a: *KFWS20090215_165332_V03_VEL4.png*

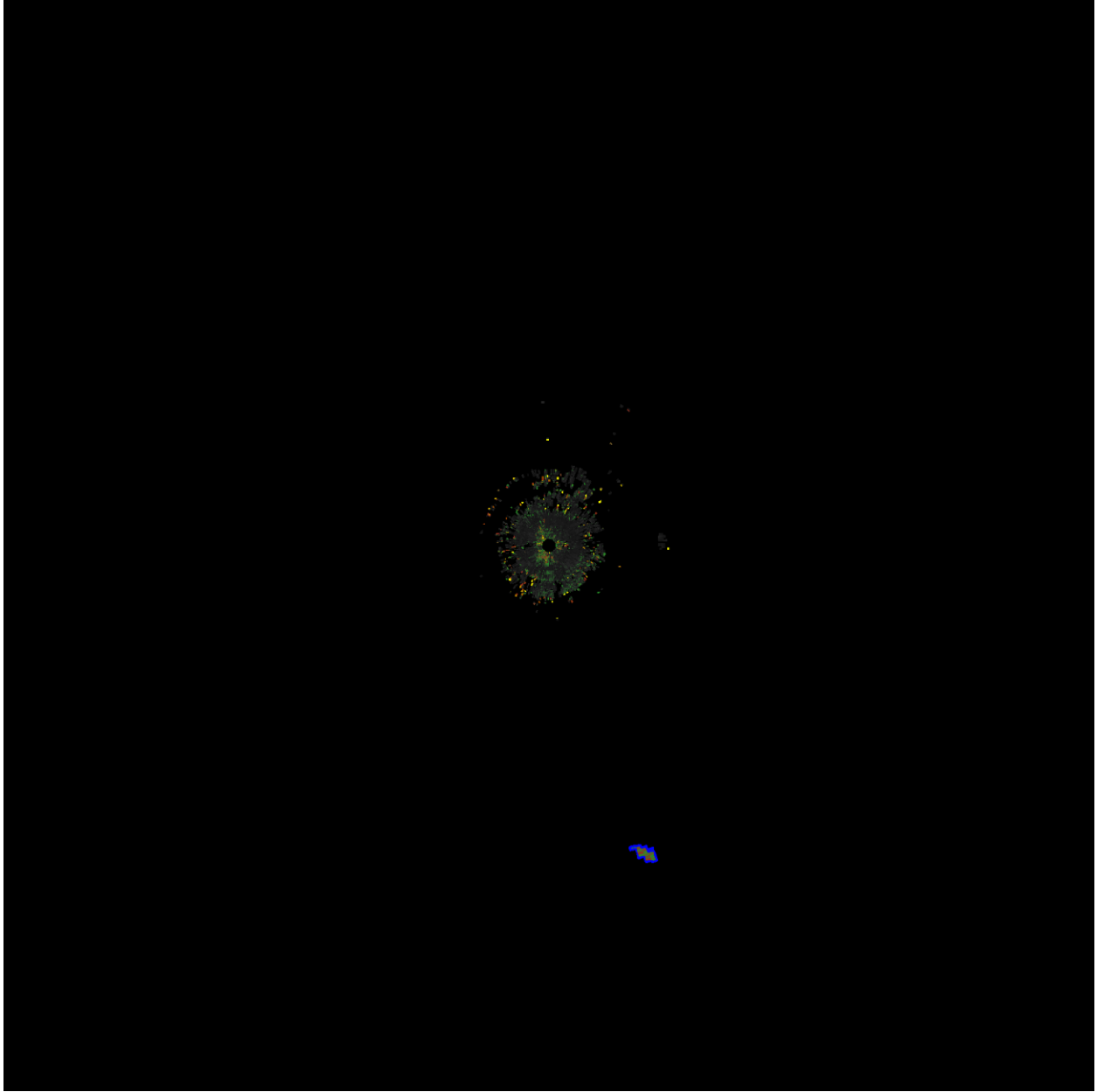


Figure 4b: *KFWS20090215_165332_V03_SPW4.png*

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.

```
#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
```

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. For trimming out weak wind speeds, the filter is applied in Figure 6b to cut out shallow-gradient effects, and is followed by a calculation which removes any “trivial” edges (i.e. the standard boundaries of single-velocity winds).

```
#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[:, :, 1], -1, xfilter)
rx = cv2.filter2D(img[:, :, 2], -1, xfilter)
gy = cv2.filter2D(img[:, :, 1], -1, yfilter)
ry = cv2.filter2D(img[:, :, 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* 100)
```

Figure 6a: Edge detection between color channels

```
#Filter trivial gradients
kernel = np.ones((2,2),np.uint8)
img = img.astype(np.uint8, copy=False)
img = cv2.threshold(img,edgeIntensity,255,cv2.THRESH_BINARY)[1]
img = cv2.erode(img, kernel)
img = cv2.dilate(img, kernel)
```

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

Strewn Fields

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