McIDAS-V Tutorial
Advanced Scripting: User Defined Functions
updated November 2023 (software version 1.9)

McIDAS-V is a free, open source, visualization and data analysis software package that is the next generation in SSEC's 50-year history of sophisticated McIDAS software packages. McIDAS-V displays weather satellite (including hyperspectral) and other geophysical data in 2- and 3-dimensions. McIDAS-V can also analyze and manipulate the data with its powerful mathematical functions. McIDAS-V is built on SSEC's VisAD and Unidata's IDV libraries. The functionality of SSEC's HYDRA software package is also being integrated into McIDAS-V for viewing and analyzing hyperspectral satellite data.

McIDAS-V version 1.2 included the first release of fully supported scripting tools. Running scripts with McIDAS-V allows the user to automatically process data and generate displays for web pages and other environments. The McIDAS-V scripting API is written in java implementation of Python called Jython. The McIDAS-V scripting library system library is still under development and new tools will be added with future releases of McIDAS-V. You will be notified at the start-up of McIDAS-V when new versions are available on the McIDAS-V webpage - https://www.ssec.wisc.edu/mcidas/software/v/.

If you encounter any errors or would like to request an enhancement, please post questions to the McIDAS-V Support Forums - https://mcidas.ssec.wisc.edu/forums/. The forums also provide the opportunity to share information with other users.

This tutorial assumes that you have McIDAS-V installed on your machine, and that you know how to start McIDAS-V. If you cannot start McIDAS-V on your machine, you should follow the instructions in the document entitled *McIDAS-V Tutorial – Installation and Introduction*. More training materials are available on the McIDAS-V webpage and in the "Getting Started" chapter of the ***McIDAS-V User's Guide***, which is available from the Help menu within McIDAS-V.

Terminology

There are two windows displayed when McIDAS-V first starts, the **McIDAS-V Main Display** (hereafter **Main Display**) and the **McIDAS-V Data Explorer** (hereafter **Data Explorer**).

The **Data Explorer** contains three tabs that appear in bold italics throughout this document: *Data Sources*, *Field Selector*, and *Layer Controls*. Data is selected in the *Data Sources* tab, loaded into the *Field Selector*, displayed in the **Main Display**, and output is formatted in the *Layer Controls*.

Menu trees will be listed as a series (e.g., *Edit -> Remove -> All Layers and Data Sources*). Mouse clicks will be listed as combinations (e.g., *Shift+Left Click+Drag*).

Preparing for this Tutorial

Beginning users of McIDAS-V should review the material covered in the *An Introduction to Jython Scripting* tutorial before proceeding with this lesson for instruction on using the **Jython Shell**, basic Jython terminology, and the preparatory material for this lesson.

This lesson will use the term "VisAD data object." A VisAD data object contains the data as well as detailed information about a data's structure and type. These objects can also contain units, temporal, and geospatial information. In this tutorial, an introduction to the VisAD data object is presented as well as a general introduction to accessing the information within the data object.

Introduction to Data Manipulation in McIDAS-V

1. Load the '*<local-path>*/**Data/UserFunctions/load_grid.py**' script provided with this tutorial into the **Jython Shell** (*Tools -> Formulas -> Jython Shell*). In the *input field* of the **Jython Shell**, type:

    ```
    editFile('<local-path>/Data/UserFunctions/load_grid.py', encoding='UTF-8')
    ```
    Click **Evaluate** (or *Shift+Enter*) to load the script into the **Jython Shell**.
    Click **Evaluate** again to execute the script.

2. Use the standard Jython function **type()** to display the class of the data contained in *g17b1*:

    ```
    print type(g17b1)
    ```
    Click **Evaluate**.

    The *g17b1* object belongs to a class called "*MappedGeoGridFlatField.*" This class is returned when data is loaded via the **loadGrid()** function.

3. The Jython built-in **type** function does not describe how the data is structured. The JPythonMethod library built into McIDAS-V contains a function called "whatTypes()". This function describes the structure of any VisAD data object. **whatTypes** can be used for debugging. In the *input field* of the **Jython Shell**, type:
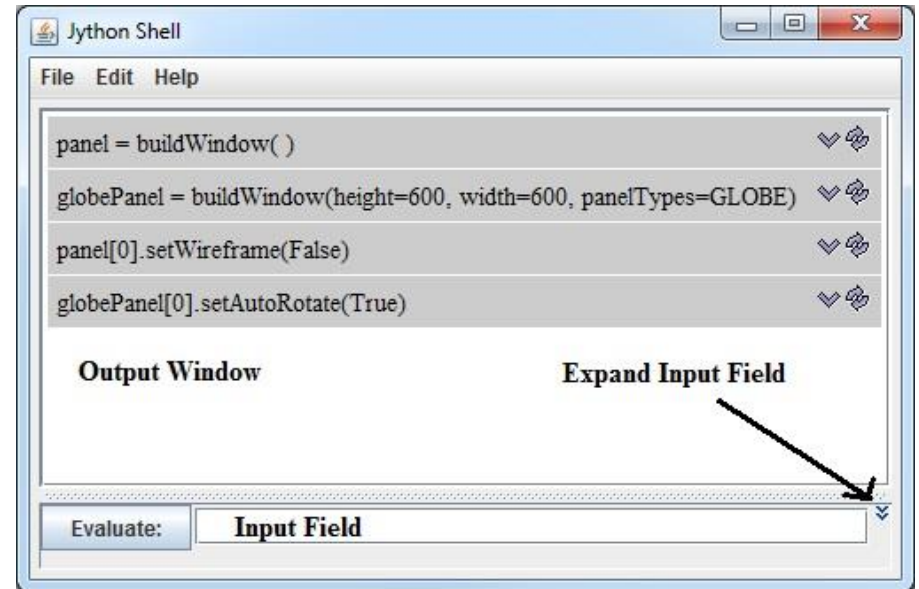
```
print whatTypes(g17b1)
```
Click **Evaluate**.

The *g17b1* object domain contains radian coordinates which are mapped to latitude/longitude values. The range of the *g17b1* data object contains the data. In this case, the data are values of radiance. Radiance has units of "W m-2 sr-1 um-1".

4. View the data mapping. Type evaluate: `print getType(g17b1)`

The result, ((x[unit:rad], y[unit:rad]) -> Rad[unit:W_m-2_sr-1_um-1]), displays a map of the radiance data (stored in the range) to the radians stored in x and y.

5. Determine the size of the data and range of the coordinates. In the **Jython Shell**, type:

```
g17ds = getDomainSet(g17b1); print g17ds
```
Click **Evaluate**.

The semicolon (;) links the commands together. In this way, multiple commands can be entered on one line, and executed in sequence. This output can be interpreted with the help of other commands run earlier. For example, step 3 evaluated the **whatTypes()** function which showed that the domain index for the x coordinate is 0 and the y coordinate is 1. This can also be inferred from the **getType()** command evaluated in step 4. Therefore, in the output of **getDomainSet()**, Dimension 0 is x which has a range of 0.06552000343799591 to 0.09338000416755676, and Dimension 1 is y which has a range of 0.11032000184059143 to 0.08246000111103058. The **getDomainSet()** output also shows that the length, or number of data points in this data are 40,000. If a user should use **len(g17b1)**, the length of the *MappedGeoGridFlatField* metadata is returned.

6. Once the data object domain set of a data object is retrieved (remember, this is the set of latitude and longitude coordinates), the latitude and longitude values for each grid point can be accessed. Using the variable *h8ds* created in the previous step, type:

```
g17latlons = getLatLons(g17ds)
g17lats = g17latlons[0]
g17lons = g17latlons[1]
```

Note: The JPythonMethod **getLatLons** resets the order of the longitude and latitude coordinates. This means that the latitude values are always returned in the zero index, and the longitudes are always returned as the first index. To get return the shape of the **g17latlons** array, type evaluate:

```
import Numeric
print Numeric.shape(g17latlons)
```

7. Print the first four latitude points. Note: In the **Jython Shell**, it is a good practice to limit the size of data printed. Type, **print g17lats[0:4]**

8. Additional methods for working with this data can be found in JPythonMethods (http://www.ssec.wisc.edu/visad-docs/javadoc/visad/python/JPythonMethods.html). For example, a method to return the min/max of the data is **getMinMax()**. In the **Jython Shell**, type evaluate:

```
print getMinMax(g17b1)
```

This method returns that the range of radiance values in **g17b1** goes from a minimum value of ~62.58 to a maximum value of ~364.69.

9. An additional JPythonMethod, **getValues()**, can be used to return the data points as float values. The values returned can be saved to a variable named *myData*. In the **Jython Shell**, type and evaluate:

```
myData = getValues(g17b1)
```

a. Determine the type of *myData*. In the **Jython Shell**, type:

```
print type(myData)
```

The type returned is an array.

b. What is the length of the *myData* array? Does it match the length found in the domain set? In the **Jython Shell**, type and evaluate:

```
print len(myData)
```

The length is 1. This does not match the length found in the domain set. See step 9c.

c. What is the length of *myData[0]*? In the **Jython Shell**, type and evaluate:

```
print len(myData[0])
```

The length is 40,000, which should match the length reported in **getDomainSet(g17b1)** in step 5. The actual shape of this array is a 1x40,000. Therefore, the first **len()** in step 9b returns the length of the first dimension of the array *myData*. In this case, since there is only one timestamp in the file, that length is 1. This is similar to a Fortran array of REAL myData(1,441). An alternative way of returning the shape of the array would be to run: `print Numeric.shape(myData)`

Exercise 1: Practice using the functions and methods introduced earlier in this tutorial. Use '*<local-path>*/**Data/UserFunctions/load_rgb.py**' for this exercise.

a.   Load '*<local-path>*/**Data/UserFunctions/load_rgb.py**' script into the **Jython Shell** and evaluate the script. This script loads three bands of GOES-17 data (three independent *MappedGeoGridFlatField* objects). These three bands of data are combined with the RGB function **mycombineRGB**. This will produce a data object named *rgbData*.

b.   Using *rgbData*, repeat the steps above to determine the class of *rgbData* and *rgbData* structure. Note the differences between the class and structure of *g17b1* and *rgbData*.

c.   What differences are there in the mapping of *g17b1* and *rgbData*?

d.   How do these differences translate into the length of the unindexed array returned from **getValues(rgbData)**?   Why is the length different than the dimensions of the flat field containing one band of data?

Note, the solution to Exercise 1 can be found below.

Exercise 1 Solution: Practice using the functions and methods introduced earlier in this tutorial. Use '<*local-path*>/**Data/UserFunctions/ load_rgb.py**' for this exercise.

a. Load '<*local-path*>/**Data/UserFunctions/load_rgb.py**' script into the **Jython Shell** and evaluate the script. This script loads three bands of GOES-17 data (three independent *MappedGeoGridFlatField* objects). These three bands of data are combined with the RGB function **mycombineRGB**. This will produce a data object named *rgbData*.

b. Using *rgbData*, repeat the steps above to determine the class of *rgbData* and *rgbData* structure. Note the differences between the class and structure of *g17b1* and *rgbData*.

```
print type(rgbData)
```
This is a visad.FlatField

```
print whatTypes(rgbData)
```
The significant difference in this step is the result in the range. As opposed to the single banded data, this RGB has three components in the range. These components are the red, green and blue values of the RGB data object.

```
print getType(rgbData)
```
The data mapping reinforces the information from the **whatType** result.

c. What differences are there in the mapping of *g17b1* and *rgbData*?

```
print getType(g17b1)
```
The data mapping reinforces the information from the **whatType** result. The data object of *g17b1* has one 1 component in the range, this is an array of radiance values:
((x[unit:rad], y[unit:rad]) -> Rad[unit:W_m-2_sr-1_um-1])

while the data object of rgbData has 3 components in the range, the red, green and blue image values:
((x[unit:rad], y[unit:rad]) -> (redimage1, greenimage1, blueimage1))

d. How do these differences translate into the length of the unindexed array returned from **getValues(rgbData)**? Why is the length different than the dimensions of the flat field containing one band of data?

```
print len(myData)
```

The length is 1.
```
myRGBdata = getValues(rgbData)
print len(myRGBdata)
```
The length is 3.

Creating Functions for the Jython Library

In this section, a function called **probe** will be created, which can be expanded to load multiple channels from the GOES-17 satellite. Later, these data will be used in a cloud algorithm. The goal of this section is to provide the user with the knowledge base that will enable them to create their own functions to meet their research needs.

10. Using the **Jython Library**, create a new function called **probe()**.

   a. In the **Main Display,** select *Tools -> Formulas -> Jython Library* to open the **Jython Library.**

   b. In the left panel of the **Jython Library**, select *Local Jython -> User's Library.*

   c. The right panel of the **Jython Library** is the location where new functions are defined.

   d. **Create a new function** in the *Training Library*. **Expand *Local Jython* and select *Training Library* (this library was created in the *An Introduction to Jython Scripting* tutorial).** *Left-Click* in the right panel of the Jython Library. Type: `def probe():`

   This defines a new function called **probe()**. This function will require three input parameters: the data object (2D lat/lon flat field of data), the latitude location, and the longitude location. Modify the definition line to include the input parameters. The final structure of the definition line should be: `def probe(field, lat, lon):`

   This function returns the data value at the input latitude and longitude location.

   Next, add the return statement: `return valueAtLocation`
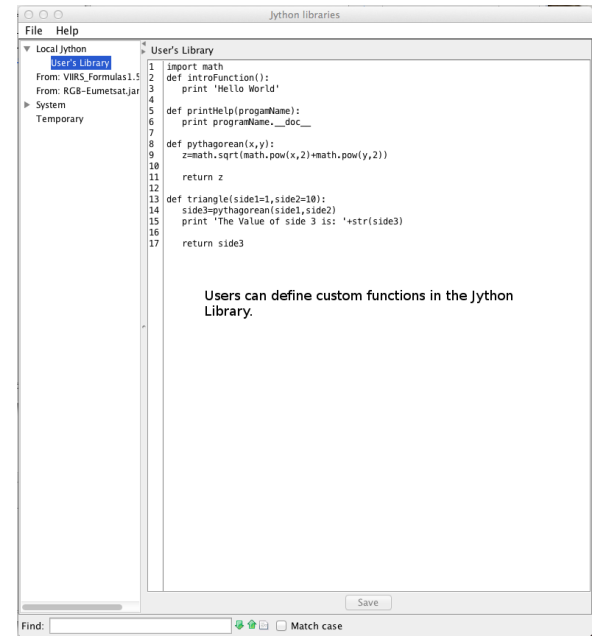
In Jython, a code block is defined by consistent indention. In this example, code block indentation will be four spaces. The two lines of the code block should now be:

```
def probe(field, lat, lon):
    return valueAtLocation
```

e. Two McIDAS-V libraries are needed for this function. Add import statements to the code block between the definition line and the return line:

```
def probe(field, lat, lon):
    from ucar.unidata.data.grid import GridUtil
    from visad.georef import EarthLocationTuple

    return valueAtLocation
```

f. Add code to retrieve data value at input lat/lon point:

```
altitude = 0.0
    loc = EarthLocationTuple(lat, lon, altitude)
    valueAtLocation = GridUtil.sampleToReal(field, loc, None)
```

The completed **probe**() function is:

```
def probe(field, lat, lon):
    from ucar.unidata.data.grid import GridUtil
    from visad.georef import EarthLocationTuple

    altitude = 0.0
    loc = EarthLocationTuple(lat, lon, altitude)
    valueAtLocation = GridUtil.sampleToReal(field, loc, None)

    return valueAtLocation
```

g. Select **Save** and close the **Jython Library**.

Preparing to Test the probe() Function by Blending Interactive and Scripting Methods

11. The Jython method, os.path.join is used in the following sequence, if the Jython "os" module has not already been imported, import it now. In *input field* of the **Jython Shell**, type: **import os**

12. If the tutorial *An Introduction to Jython Scripting* was completed before beginning this tutorial, proceed to step 14.

13. If the step to create a local ADDE entry for the Joplin IR data was not completed in the *An Introduction to Jython Scripting* tutorial, create a local ADDE entry for data for the Joplin Tornado GOES-13 IR data. In the **Jython Shell**, type:

```
homePath = expandpath('~')
dataPath = os.path.join(homePath,"Data")

areaPath = os.path.join(dataPath,"Scripting","tornado-areas")
irPath = os.path.join(areaPath,"IR")

irDataSet = makeLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR', mask=irPath, format='McIDAS
Area', save=True)
```

14. Load this data into the *Field Selector* via the interactive *Data Chooser*. Display the data using the McIDAS-V scripting API.

a. Bring the **Main Display** forward. In the **Jython Shell**, type:
```
activeDisplay().toFront()
```

b. Click the [button icon] button on the **Main Toolbar** to bring the **Data Explorer** to the front.

c. In the **Data Explorer**, open the *Data Sources* tab.

d. Using the *Satellite -> Imagery* chooser, load the *<LOCAL-DATA>* created in previous step (**Dataset**: *TORNADO*, **Image Type**: *GOES-13 IR*) via the interactive Data Chooser. In the *Relative* tab, enter *5* in the **Number of times** box and click **Add Source**. Additional

instructions for loading satellite imagery can be found under *Choosing Satellite Imagery* in the User's Guide or the *Satellite Imagery Tutorial*.

    e. When the Joplin data is in the **Field Selector**, click the **Jython Shell** (or select *Tools -> Formulas -> Jython Shell)* to bring the **Jython Shell** forward.

    f. The Joplin data will be selected from the **Jython Shell**, and used to test the **probe**() function. In the *input field* of the **Jython Shell** type:
        joplin = selectData('IR')

    g. Click **Evaluate**. Note: The **Field** Selector title is "IR." The title matches the string entered in the **selectData()** call**.**

    h. In the secondary **Field Selector** window, select the *GOES-13 IR -> 10.7 um IR Surface/Cloud-top Temp -> Temperature* data source.

    i. Once the bottom *Times* tab in the window becomes visible, click **OK**.

    j. Display the Joplin image, type:

```
panel = buildWindow()
panel[0].createLayer('Grid Table', joplin[0])
```

15. Notice the section of data displayed in the grid table with no latitude or longitude and a value of 0 for *Band4_TEMP_0*. These are points in the area that are not navigated on the earth.

    a. At the top of the *Layer Controls* turn on the option for **Show native coordinates**. Now, instead of seeing invalid latitude and longitude values, image element and image lines are listed. This demonstrates that while there are data points without a latitude and longitude, they are points included in the AREA itself.

    b. Once done interrogating the chart, turn **Show native coordinates** off.

16. For analysis and sometimes display purposes, these points can be set to "Missing". In the **Jython Shell** type:

```
joplinMiss = setMissingNoNavigation(joplin[0])
panel[0].createLayer('Grid Table', joplinMiss)
```
Notice the values are now or (dependent on your platform), corresponding to a missing value.

Exercise 2: Test the **probe()** function:

a. Select five navigated points from the **Grid Table** display. Write the five test points in the table below:

Latitude	Longitude	Data Value

b. Test the probe function on the five points by using each lat/lon value pair in the function call. In the **Jython Shell**, type:

```
print probe(joplinMiss, <latitude>, <longitude>)
```

c. Determine the class of *joplin* and *joplin[0]* using **print type()**. In the **Jython Shell**, type:

```
print type(joplin)
print type(joplin[0])
```

Data Object	Belongs to class
joplin	
joplin[0]	

d. Using the **getType** method **(object.getType())**, inspect the data mapping of *joplin* and *joplin[0]*. Write the mapping in the space below. In the **Jython Shell**, type:

```
print getType(joplin)
print getType(joplin[0])
```

e. Use the **getImageTimes()** method on *joplin*. In the **Jython Shell**, type:

```
print joplin.getImageTimes()
```

Note, the solution to Exercise 2 can be found on page 12

Extra: Using the **describe()** function scheduled for release with McIDAS-V 1.6, type:

```
print describe(joplin[0])
```

Click **Evaluate**.

This function is meant to show quick statistics for a flatField or multiple flat fields. If you have questions, comments, requests for additional functionality or bug reports, please post them to the Scripting Forum.

It is possible to calculate statistics on any array, provided it is converted to a VisAD field:
```
myField = field([1, 2, 3, 4, 5, 9, 9, 9, 9])
print describe(myField)
```

Click **Evaluate**.

Exercise 2 Solution: Test the **probe()** function:

a. Select five navigated points from the **Grid Table** display. Write the five test points in the table below:

Latitude	Longitude	Data Value
24.725	-94.83	296.1

This chart contains one sample latitude/longitude/data point to demonstrate an example solution.

b. Test the probe function on the five points by using each lat/lon value pair in the function call. In the **Jython Shell**, type:

```
print probe(joplinMiss, 24.725, -94.83)
```

This returns a value of 296.1, matching the grid table output.

c. Determine the class of *joplin* and *joplin[0]* using **print type()**. In the **Jython Shell**, type:

```
print type(joplin)
print type(joplin[0])
```

Data Object	Belongs to class
joplin	visad.meteorology.ImageSequenceImpl
joplin[0]	visad.metrorology.NavigatedImage

d. Using the **getType** method **(object.getType())**, inspect the data mapping of *joplin* and *joplin[0]*. Write the mapping in the space below.
In the **Jython Shell**, type: `print getType(joplin)`

(Time -> ((ImageElement, ImageLine) -> Band4_TEMP))
The *joplin* data object is an image sequence, so this command shows that the time domain (which could be multiple times) is linked to the data coordinates.

Type: `print getType(joplin[0])`
 ((ImageElement, ImageLine) -> Band4_TEMP)
The *joplin[0]* data object is the navigated image, this reveals that the image has element and line coordinates mapped to the data.

e. Use the **getImageTimes()** method on *joplin*. In the **Jython Shell**, type and evaluate:

`print joplin.getImageTimes()`
This step emphasizes the point that the image sequence data object *joplin* is an image sequence and the times can be extracted as a list.

Scripting Exercise #1: Simple Radar Rain Rate Formula

Despite the complexity of the **VisAD Data Model**, simple mathematical transforms on data objects are easy. Standard mathematical operators work on entire **VisAD objects** just like they do on individual numbers. This exercise will demonstrate use of arithmetic operators on **VisAD data objects** to calculate rain rate from radar reflectivity.

Many simple empirical relationships between rain rate and radar reflectivity have been derived over the years. The following table lists a few; these all take the same mathematical form.

| Table 1. Z-R RECOMMENDATIONS		
RELATIONSHIP	**Optimum for:**	**Also recommended for:**
Marshall-Palmer ($z=200R^{1.6}$)	General stratiform precipitation	
East-Cool Stratiform ($z=130R^{2.0}$)	Winter stratiform precipitation - east of continental divide	Orographic rain - East
West-Cool Stratiform ($z=75R^{2.0}$)	Winter stratiform precipitation - west of continental divide	Orographic rain - West
WSR-88D Convective ($z=300R^{1.4}$)	Summer deep convection	Other non-tropical convection
Rosenfeld Tropical ($z=250R^{1.2}$)	Tropical convective systems	

Source: http://www.ou.edu/radar/z_r_relationships.pdf

17. Note: The rain rate formulas above are in terms of radar reflectivity Z. However, Level 3 radar products give radar reflectivity in dBZ ("decibels of Z"), so the following conversion is needed: $dBZ = 10\log_{10}(Z)$. In addition, units of mm/hr are converted to in/hr for easy comparison to other NEXRAD products later on.

    a. Type the following Jython function into your **Jython Library** (or paste from file **marshall_palmer.py**).

    ```
    def marshall_palmer(dbz):
        a = 200.0
        b = 1.6
        z = 10.0**(dbz/10.0)
        r_mmhr = (z/a)**(1/b)    # rainrate [mm/hr]
    ```

```
r_inhr = r_mmhr/25.4     # rainrate [in/hr]
return newUnit(r_inhr, 'in/hr', 'in/hr')
```

b. Create a formula using this Jython function.
   1. Select **Tools -> Formulas -> Create Formula** from the **Main Display**.
   2. For **Description** and **Name**, type: *marshall_palmer*
   3. For **Formula**, type: *marshall_palmer(dbz)*
   4. Click **Add Formula.**

c. Create a data source for the five most recent times of Base Reflectivity data for a station with active ongoing precipitation (check an online radar mosaic to find a good station!). Display the data.
   1. Using the *Radar -> Level II -> Remote* chooser, connect to the *https://thredds.ucar.edu/thredds/radarServer/catalog.xml* catalog.
   2. For **Collection**, select *Nexrad Level II Radar from IDD*.
   3. In the **Station** panel, select a station with active precipitation.
   4. In the *Relative* tab of the **Times** panel, enter *5* in the **Number of times** text entry box.
   5. Click **Add Source**.
   6. In the *Field Selector*, select the *Reflectivity -> Reflectivity Elevation Angle .352* (or whatever the lowest elevation angle is for the data) field and the *Radar Sweep View in 2D* display type. Click **Create Display**.

d. In the **Field Selector**, click the *Formulas* data source, and select your *marshall_palmer* formula. Select the *Radar Displays -> Radar Sweep View in 2D* display type and click **Create Display**. In the **Field Selector** window, select the *Level II Radar Data -> BaseReflectivity -> BaseReflectivity Elevation Angle .352* field for the *dbz* field and click **OK**.

e. You should now have your Marshall-Palmer rain rate displayed. Choose an enhancement table and range to make precipitation features stand out. Probe the reflectivity and rain rate values. Do the Marshall-Palmer rain rates seem realistic?

f. Using the *Radar -> Level II -> Remote* chooser, change the **Collection** to *NEXRAD Level III Radar from IDD*, set the **Product** to *Digital Instantaneous Precipitation Rate (DPR)*. Load the 5 most recent times and display the *DigitalInstanteneousPrecipitationRate* field. Does this level 3 product compare well to Marshall-Palmer?
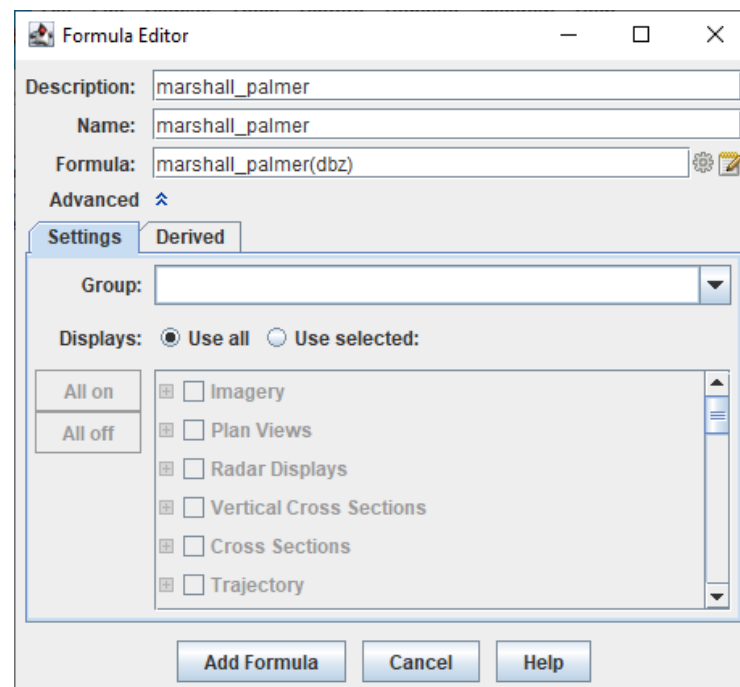
g. Optional exercise: starting with the **marshall_palmer** Jython function above, implement one function for each rain rate formula in the table above. Use what you have learned about Python syntax to avoid repeat code. When finished, determine if any of these formulas match the *DPR* product better than Marshall-Palmer. An example solution is available at *<local path>*/**Data/UserFunctions/rainrate_solution.py**.

Scripting Exercise #2: Build a GOES-17 Cloud Mask (Comprehensive Exercise)

Using **loadGrid**, the **Jython Library**, and knowledge of the **VisAD data object,** build a simple cloud mask algorithm for GOES-17's ABI sensor. This exercise will extend your knowledge of the **VisAD data model** by using methods to extract raw data arrays from **VisAD data objects and** operate on the data arrays in a way that is familiar to users already comfortable with programming languages like MATLAB® or Fortran.

First, explore a GOES-17 scene interactively and determine cloud/no-cloud thresholds for various ABI channels. Next, add the outline of a Jython cloud mask function provided to your **Jython Library**. Implement the algorithm of the cloud mask Jython function with the thresholds determined interactively. Finally, run the algorithm as many times as needed while exploring the results. Iterate between interactively assessing thresholds and algorithm development until you are satisfied with the cloud mask.

18. Load GOES-17 bands interactively and determine cloud thresholds. The GOES-17 data distributed with this tutorial is split into a single band per file. For example, band 1 contains the string "M3C01" at the end of the filename. Create a data source per band you are interested in.

  a. Add a data source for band 1.

    1. In the **Data Explorer**, open the *Data Sources* tab and navigate to the *Gridded Data -> Local* chooser.

    2. Under **Data Type**, select "*Grid files (netCDF/GRIB/OPeNDAP/GEMPAK)*".

    3. Navigate to '*<local path>*/**Data/UserFunctions/G17**' and *Double-Click* the file *OR_ABI-L1b-RadM2-M3C01_G17_s20190142000569_e20190142001026_c20190142001060.nc*.

  b. Repeat step 18a for other bands of interest. Add a data source for at least three more bands, including band 2 (high resolution visible channel), band 5 (snow/ice band), band 9 (a channel with high water vapor absorption), and band 14 (an infrared window channel).

  c. Display the data for band 1.

    1. Create a new one panel map tab in the **Main Display** window.
    2. In the **Data Sources** panel of the *Field Selector*, select the top *OR_ABI-L1b** data source. The order of the data sources in this

panel is determined by the order they were added. Since band 1 was added first in step 18a, this is at the top of the list.

3. Select the *2D grid -> ABI L1b Radiances* field.
4. Open the *Data Sampling* tab. Uncheck Use Default, and set X and Y to "*All Points*".
5. Choose the *Imagery -> Image Display* display type and click Create Display.

d. Create a *Data Probe/Time Series* display for all data sources created in steps 18a and 18b.

1. In the **Data Sources** panel of the *Field Selector*, select the top *OR_ABI-L1b** (band 1) data source. Choose the *Data Probe/Time Series* display type.
2. Click **Create Display**.
3. To avoid confusion as more layers are added to the display, change the parameter name below the chart from *Rad* to *Band 1*. To do this, *Right-Click* on *Rad* below the chart and choose *Parameter Rad -> Chart Properties*. In the **Properties** window, set the **Legend Label** field to be *Band 1* and click **OK**. An alternative way of getting to the **Properties** window for a layer in the chart is to *Double-Click* on the parameter.

e. Add the other fields added in step 18b to the *Data Probe/Time Series* display. In the *Layer Controls* of the *Data Probe/Time Series* display, *Left-Click* on the *Band 1* data readout line to select the line.

1. Once the line is highlighted, *Right-Click* to select *Add Parameter...*.
2. Use the secondary *Field Selector* select the other fields, clicking **Add Selected>>** after each addition.
3. When all the desired fields have been added to the panel below **Add Selected>>** click **OK**.
4. Repeat the last step in 18d to set the parameter name of each field to be the band number.

f. Explore the scene. Using the Data Probe (hold down the middle mouse button over the display) and the **Data Probe/Time Series** display, write down a cutoff value that is representative of the transition from not cloudy to cloudy. This is just to get started – clearly, you won't find a single value for a single channel that works perfectly for the entire scene.

Hint: Create an Image Display of band 5 Radiances to determine radiance cutoff values for snow vs. clouds and land. Snow will have lower radiance values than clouds and snow-free land, so it will appear darker in the Image Display.

19. Copy the script '<*local-path*>/**Data/UserFunctions/abi_cloudmask.py**' to the **Jython Library**. This is the outline of a cloud mask script that can be improved in later steps. Try running the cloud mask provided without modification:

a. Open abi_cloudmask.py in a text editor.

b. In the **Main Display**, click *Tools -> Formulas -> Jython Library*.

c. Create a new **Jython Library** by selecting *File->New Jython Library* from the menu. Enter *ABI-cloudmask* and click **OK**.

d. Open the *Local Jython -> ABI-cloudmask* library. Copy and paste the text in abi_cloudmask.py from the text editor into the library and click **Save**.

e. Open the Jython Shell (click *Tools -> Formulas -> Jython Shell*) and type:
    ```
    run_abi_cloudmask()
    ```
    Click **Evaluate**.
    You should get a new display showing a cloud mask, but at this point it is pretty bad! It misses most of the clouds in the scene.

20. Modify the cloud mask function to implement the thresholds you determined in step 18:

a. In the **Jython Library**, examine the **abi_cloudmask**() code pasted in the previous step. The code locates the ABI data on the disk, loads it into McIDAS-V with **loadGrid**, and then organizes it by band number in a Python dictionary for easy access. For the purposes of this exercise, the focus will be on the function **do_cloudmask**, which implements the cloud mask algorithm. The outline of this function is already implemented; it is up to the user to fill in the cloud/no-cloud thresholds determined in step 18.

b. Now, implement your cloud mask. The portion of the script you will work on is labeled "*You can fill in additional clear/cloudy tests here*". Directly above this comment, there are two tests already implemented, one for band 14 and another for band 1. These tests use the function **mask** from *JPythonMethods* to create a simple cloud/no-cloud mask based on brightness temperature or reflectance threshold. Modify these two tests with the thresholds determined for band 1 and band 14 in previous steps. For example, assuming you wrote down a threshold of 82 for band 14, change the line:

    ```
    threshold_mask = mask(abi_datas[14], 'lt', 70, False)
    ```
    to
    ```
    threshold_mask = mask(abi_datas[14], 'lt', 82, False)
    ```

c. Now, copy the block labeled "Test number 2" (line 123), and paste it directly below. Change the comment to "Test number 3", change the dictionary key to the next channel you would like to work on, and change the threshold as desired. For example, band 5 could be added to the threshold mask to mask out areas of snow cover.

    ```
    threshold_mask = mask(abi_datas[5], 'gt', 10, False) * threshold_mask
    ```

Note that each new band added to this threshold mask will need to be added to the bandList list specified in the load_data function.

d. Redo the previous step to implement all the thresholds you determined.

e. Once all of the cloud mask tests have been added to the library, click **Save** at the bottom of the **Jython Library**.

f. In the **Jython Shell**, rerun **run_abi_cloudmask()** and examine the results. Has your cloud mask improved?

g. These radiance threshold tests are simple to define and run quickly because they can be implemented with built-in functions like **mask** that operate on an entire VisAD data object in a single function call. You may ask: what if I have an algorithm that cannot be expressed using functions from *JPythonMethods*? For example, cloudy areas are often more spatially heterogeneous than cloud-free areas and decide you want to develop a test that considers the standard deviation of neighboring pixels as a simple measure of spatial heterogeneity.

    To illustrate this, the file abi_cloudmask.py has already implemented a function called **do_std_dev_test**, but it is initially unused by **do_cloudmask**. As the first step, modify **do_cloudmask** to use **do_std_dev_test**. This part is simple because **do_std_dev_test** returns a **VisAD Data object** (similar to what is returned by **mask**) that can be multiplied by the current cloud mask result to obtain the combined cloud mask:

    ```
    return threshold_mask
    ```

    becomes:

    ```
    std_dev_mask = do_std_dev_test(abi_datas)
    return threshold_mask*std_dev_mask
    ```

h. In the **Jython Shell**, rerun **run_abi_cloudmask()** and examine the results. Inspect both large, spatially uniform clouds, and smaller spatially complex clouds. How did the standard deviation test affect the results? What are the test's strengths and weaknesses? To see the results of **do_cloudmask** clearly, modify **do_cloudmask** so that it *only* uses the **do_std_dev_test** mask.

i. The algorithm used by **do_std_dev_test** is simple: calculate the standard deviation of a 3x3 array of pixels surrounding a given pixel for some channel, and label that pixel cloud or clear based on some threshold for that channel. However, this cannot be expressed with *JPythonMethods*! So, the implementation must loop through each pixel, extract the neighboring pixels, and calculate the standard deviation of that set of pixels. This has a performance cost (Jython is an interpreted language, not a compiled one, so long *for* loops will be slow) but it is the only way to implement this algorithm in pure Jython. Here is a more detailed description of the implementation of **do_std_dev_test**:

1. Create an array to write to contain the cloud mask result. The easiest way to do that is to **clone()** one of the input data objects:

```
output = ahi_datas[14].clone()
```

2. The full **VisAD data object** is not needed, we just want the data array it is holding on to. This is done with **getFloats(False)[0]**. In this case, **False** means the values of the array are not copied, however, the range of the array is maintained (If **getFloats(True)** is used, the cloud mask modifications will not affect the output!). The index, **[0]**, in the **getFloats(False[0])** statement indicates that the first "**range component**" of the object is selected. This pattern will work for most **VisAD data objects**. A common exception is a **VisAD data object** that represents an RGB image; in that case, there are three "**range components**", so **[0]** refers to the red image, **[1]** refers to the green image, and **[2]** refers to the blue image. Back to the present case, the full line of code to get the data array is:

```
outFloats = output.getFloats(False)[0]
```

The array is named **outFloats** to signify it is an array of floating point values. This combination of **clone** and **getFloats** is a common pattern in McIDAS-V Jython for obtaining an output array to write a result to.

3. The next section is similar to the previous step, but a new Python dictionary called **ahi** is defined and used. Each key in **ahi** is mapping to a data array for that band. This a convenient reference to the data array for each band later on:

```
ahi = dict()

for data_key in ahi_datas.keys():

    ahi[data_key] = (ahi_datas[data_key].getFloats(False))[0]
```

4. Next, iterate through each pixel. We use the Python function **enumerate** to get a reference to both the index **i** and the value of the image at **i** (hereafter, the value of the image at the index **i** will be called **pixel**):

```
for i, pixel in enumerate(outFloats):
```

5. Next, the standard deviation tests are done. For each pixel, extract the surrounding 3x3 pixels using a helper function called **get_3x3**. VisAD internally represents all data as a 1-dimensional array, so the dimensionality must be handled correctly. The key to the implementation is determining how many elements are in each scan "line" using the function **getDomainSizes**. Use that scan line element size information to skip ahead to the same horizontal location in the next "line". Inspect the implementation of **get_3x3** and convince yourself that it is correct.

6. Finally, implement one more helper function called **std_dev** that calculates the standard deviation of a list of numbers according to the usual statistical formula. Send the result of **get_3x3** into **std_dev** to obtain a simple measure of spatial heterogeneity, and experimentally determine useful thresholds to label the current pixel cloudy or clear.

21. Modify your thresholds for brightness temperatures, reflectances, and standard deviations as needed until you are happy with the cloud mask. Feel free to implement other tests you may think of, such as channel differences.

Files Used In This Tutorial

load_grid.py

```
# import the jython library used
import os

homeDirectory=expandpath('~')
dataDirectory=os.path.join(homeDirectory,'Data','UserFunctions','G17')

# Add the filename to the data
band1File=os.path.join(dataDirectory,'OR_ABI-L1b-RadM2-
M3C01_G17_s20190142000569_e20190142001026_c20190142001060.nc')

# Initialize the loadGrid parameters.
parms = dict(
    field='Rad',
    xStride = 5,
    yStride = 5,
    )

# load the data from the file
g17b1=loadGrid(filename=band1File,**parms)
```

load_rgb.py

```
import os

#create data path and file names with expandpath and jython os.path.join
homeDirectory=expandpath('~')
dataDirectory=os.path.join(homeDirectory,'Data','UserFunctions','G17')

band1File=os.path.join(dataDirectory,'OR_ABI-L1b-RadM2-
M3C01_G17_s20190142000569_e20190142001026_c20190142001060.nc')
band2File=os.path.join(dataDirectory,'OR_ABI-L1b-RadM2-
M3C02_G17_s20190142000569_e20190142001026_c20190142001052.nc')
band3File=os.path.join(dataDirectory,'OR_ABI-L1b-RadM2-
M3C03_G17_s20190142000569_e20190142001026_c20190142001061.nc')

# Initialize parameters for loadGrid.
parms = dict(
    field='Rad',
    xStride = 1,
    yStride = 1,
    xRange = (100,110),
    yRange = (100,110)
    )

# load data from file with parameters specified in dictionary
b1=loadGrid(filename=band1File,**parms)
b2=loadGrid(filename=band2File,**parms)
b3=loadGrid(filename=band3File,**parms)

#create true color rgb from (0.86,0.64,0.47)
#note that this does not attempt to simulate a true green band
rgbData=mycombineRGB(b3,b2,b1)
```

abi_cloudmask.py

```
"""
McIDAS-V script showing techniques for for creating a simple cloudmask.
"""
import os
import math

def std_dev(data):
    """Calculate standard deviation of array of numbers.

    Input:
      data: array of numbers

    Output:
      standard deviation of data
    """
    mean = sum(data) / len(data)
    a = 0.0
    for pix in data:
        a += (pix - mean)**2.0
    sigma = math.sqrt(a / len(data))
    return sigma


def get_3x3(i, data, data_obj):
    """Find 3x3 (in 2-dimensional space) set of pixels surrounding index i.

    Input:
      i: index representing center of 3x3 set.
      data: 1-dimensional array of numbers to take 3x3 set from.
      data_obj: VisAD object corresponding to "data". We use it to determine
                dimensionality of "data", since data itself is just a 1D array.

    Output:
      List of 9 numbers representing the 3x3 set of pixels surrounding i.
    """
    nx = getDomainSizes(data_obj)[0]
```

```python
    try:
        result = [
                    data[ i - nx - 1 ],   # upper left
                    data[ i - nx     ],   # upper
                    data[ i - nx + 1 ],   # upper right
                    data[ i + nx - 1 ],   # lower left
                    data[ i + nx     ],   # lower
                    data[ i + nx + 1 ],   # lower right
                    data[ i - 1      ],   # left
                    data[ i + 1      ],   # right
                    data[ i          ],   # self
                ]
    except IndexError:
        return None  # can't calculate 3x3 @ boundaries
    return result


def do_abi_std_dev_test(abi_datas):
    """The guts of the standard deviation (spatial heterogeneity) test.

    We split it into a separate function because the technique is quite
    different than the simple threshold tests. We loop through each pixel
    instead of using array operations like "mask".

    Input:
      abi_datas: dictionary with keys for each required ABI input band,
                 each pointing to a VisAD Data object corresponding to that
                 band.

    Output:
      VisAD Data object containing your cloudmask. 0==cloudy; 1==clear
    """
    CLOUDY = 0
    CLEAR = 1

    # Make a data object to write the result to:
    output = abi_datas[14].clone()
```

```
# Right now we have VisAD Data objects. We need to get the actual data
# arrays before we can iterate over each pixel...
# Get the actual data array corresponding to output object:
outFloats = output.getFloats(False)[0]
# Note that changes to outFloats will change the output.
# Get the actual data arrays for each band we have data for:
abi = dict()
for data_key in abi_datas.keys():
    abi[data_key] = (abi_datas[data_key].getFloats(False))[0]


# Loop through each pixel, doing any cloudy/clear checks you desire:
for i, pixel in enumerate(outFloats):
    # Band 1 standard deviation test
    pix3x3 = get_3x3(i, abi[1], abi_datas[1])
    if pix3x3 is not None:
        sigma = std_dev(pix3x3)
        if sigma > 120:
            outFloats[i] = CLOUDY
            continue

    # Band 14 standard deviation test
    pix3x3 = get_3x3(i, abi[14], abi_datas[14])
    if pix3x3 is not None:
        sigma = std_dev(pix3x3)
        if sigma > 5:
            outFloats[i] = CLOUDY
            continue

    # passed!  pixel is clear!
    outFloats[i] = CLEAR

return output


def do_abi_cloudmask(abi_datas):
    """This is the guts of our cloudmask algorithm.

    Input:
```

```
    abi_datas: dictionary with keys for each required ABI input band,
              each pointing to a VisAD Data object corresponding to that
              band.

    Output:
      VisAD Data object containing your cloudmask. 0==cloudy; 1==clear
    """
    # Test number 1: Basic 11.2um radiance threshold check.
    threshold_mask = mask(abi_datas[14], 'lt', 70, False)

    # Test number 2: Basic 0.47um radiance threshold check.
    # From this point forward, we need to multiply by previous cloudmask.
    threshold_mask = mask(abi_datas[1], 'gt', 120, False)*threshold_mask

    ######################################################################
    # You can fill in additional clear/cloudy tests here,
    # using the pattern shown in tests 1 and 2 above.
    ######################################################################

    return threshold_mask


def load_abi_data():
    """Load all available input data from GOES-17 files in 'dataDirectory'.

    Returns:
      Dictionary holding all the data objects read from GOES-17 input files.
      Each (integer) key corresponds to a band number, and each value is a VisAD
      Data object.
    """
    from glob import glob
    homeDirectory = expandpath('~')
    dataDirectory = os.path.join(homeDirectory,'Data','UserFunctions','G17')
    fList = os.listdir(dataDirectory)

    # Limit to a handful of bands for current testing: 0.47um, 6.2um, 11.2um
    # See band list: https://www.goes-r.gov/education/ABI-bands-quick-info.html
    bandList = [1, 5, 8, 14]
```

```
    # Initialize the empty dictionary:
    abi_datas = dict()

    # Load the CMI data from each file:
    for x in bandList:
        data = os.path.join(dataDirectory,fList[x-1])
        dataParms = dict(
            filename = data,
            field = 'Rad',
            )
        print 'Loading data from band %s' % x
        abi_datas[int(x)] = loadGrid(**dataParms)

    # Remap everything to 2km bands for 2km final output product.
    for band in bandList:
        if int(band) > 5 and int(band) != 4:
            # Skip bands that are already 2km.
            continue
        abi_datas[int(band)] = resampleGrid(abi_datas[int(band)], abi_datas[14])

    return abi_datas


def run_abi_cloudmask():
    """ The main driver function.

    This is where we load data, call the algorithm, and create a display.
    """
    # Load the ABI data using our custom data load function:
    abi_datas = load_abi_data()

    # Run our custom cloudmask algorithm:
    cloudmask = do_abi_cloudmask(abi_datas)

    # Get a reference to the panel we will create displays in:
    panel = activeDisplay()
```

```
    # Display our cloudmask and set the layer label:
    cloudmask_layer = panel.createLayer('Image Display', cloudmask)
    cloudmask_layer.setEnhancement('default')
    cloudmask_layer.setLayerLabel(label='Custom Cloudmask. Red=Cloudy; Blue=Clear',
            size=18, color='yellow')
```

marshall_palmer.py

```
def marshall_palmer(dbz):
    a = 200.0
    b = 1.6
    z = 10.0**(dbz/10.0)
    r_mmhr = (z/a)**(1/b)    # rainrate [mm/hr]
    r_inhr = r_mmhr/25.4     # rainrate [in/hr]
    return newUnit(r_inhr, 'in/hr', 'in/hr')
```

rainrate_solution.py

```
def rainrate(dbz, a, b):
    z = 10.0**(dbz/10.0)
    r_mmhr = (z/a)**(1/b)    # rainrate [mm/hr]
    r_inhr = r_mmhr/25.4     # rainrate [in/hr]
    return newUnit(r_inhr, 'in/hr', 'in/hr')


def marshall_palmer(dbz):
    return rainrate(dbz, 200.0, 1.6)


def wsr_88d_convective(dbz):
    return rainrate(dbz, 300.0, 1.4)


def west_cool_stratiform(dbz):
    return rainrate(dbz, 75.0, 2.0)
```