Chapter 17

IDL, FTP, and MODIS

In this section, we'll take a look at how to retrieve MODIS files from a file transfer protocal (FTP) site and write a routine to pull these images from the server based on certain criteria defined by the user.

17.1 Introduction

Popular sites for downloading NASA satellite imagery are the USGS Earth Explorer and the USGS Global Visualization Viewer (GLOVIS). These websites have interactive maps and all sorts of features and options. However, going through the motions of navigating the map, picking out options, and requesting your download for each image you need can be tedious at best if your analysis requires more than a few images.

FTP servers are quick and powerful tools for downloading data and, with a little background knowledge, we can develop tools for automating the process based on the user's needs. In this tutorial, we'll review string operations, initialize an IDLnetURL object, set the object's properties, and call its methods in order to communicate with the server. This tutorial is constructed to enable the batch downloading of MOD09A1 and MYD09A1 products. The methodology of the following sections can be applied to several other MODIS products, but because there are literally hundreds of products, the syntax and structure of these procedures may require some extra tweaking.

But first, a little background information on MODIS.

Aside:
We've used objects before, starting in Chapter 5 when we added various elements to a list using the "Add" method. We also created a cgMap object in Chapter 10 to make mapping easier.

17.2 What's in a name? MODIS naming conventions

The Moderate Resolution Imaging Spectroradiometer (MODIS) instrument is a NASA Earth Observation on two different satellites, Aqua and Terra. Imagery captured is downlinked and sent to the Land Processes Distributed Active Archive Center (LPDAAC) where raw, level 1 data is processed into higher level products.

As with all publicly available NASA data sets, imagery from the MODIS sensors follow a very specific naming convention. This helps with organization and allows the user to quickly determing the product, date, and location of the file in question. Let's look at an example of a file and break it down from left to right:

```
MOD09A1.A2008193.h00v08.005.2008019201033.hdf
```

MOD09A1 is a classic Level-3 reflectance product. It's worth noting that the second letter of the product name indicates the satellite(s) that this image was taken from. In this case, MOD indicates that this image was taken on the Terra satellite. MYD products came from Aqua, and MCD products are combined products made from imagery from both Aqua and Terra. Some products are satellite-specific, while others are not. Believe it or not, the file beginning with

```
MYD09A1.A2008193.h00v08.005
```

is an actual product from Aqua that you can download just like our original from Terra! This coincidence might prove useful to us later...

Each section of our filename is separated by periods. After our product name comes the year and date in the Julian calendar. For this file, we're looking at an image taken on (193, 2008) or July 11, 2008. [1]

Next, we have our horizontal and vertical grid indices. MODIS images are arranged in tiles roughly 10x10 degrees. It's important to note that these parameters are zero-padded to the left in the file name.

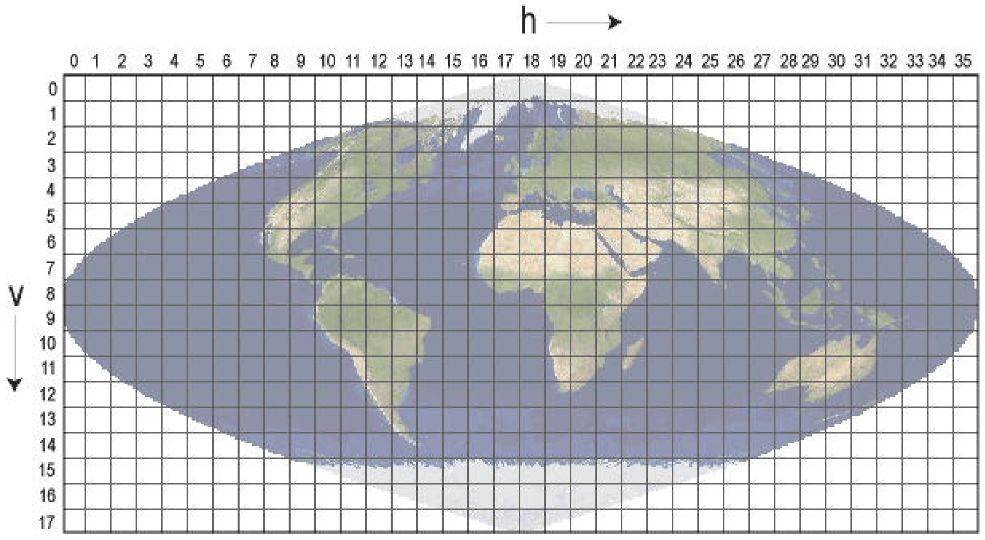

Figure 17.1: The MODIS sinusioidal grid projection

[1]Be sure to account for leap years!

Next, we have 005, which indicates that this product is located within MODIS collection #5. This collection contains all of the products we might need, so we'll keep this at 5 for the purposes of this tutorial.

The next section is the processing date, which is a bit of a wildcard. You may have noticed that, up until now, each of our parameters in this file name are easy to control and predict while we're "shopping" for MODIS images. Collecting this information will be an important step in our process of automating our downloads.

Lastly, we have the file extension .hdf. This is a common and well-documented file type for satellite imagery. We can even pop it open in our netcdf viewer. Nice!

17.3 Any way you want it: Predicting MODIS filenames

The key to this method of downloading images is the consistency of the MODIS file naming convention. By parameterizing each element of the filename, we can build "partial" filenames that can be used to search for the image later. In essence, we can use user input to "build" this:

```
MOD09A1.A2008193.h00v08.005
```

in order to search for and download this file:

```
MOD09A1.A2008193.h00v08.005.2008019201033.hdf
```

As you can probably imagine, this will require many string operations. We'll write a procedure with a helper function or two to make this happen. Let's get started.

⟶ ⟵

```
;file documentation
PRO ess_download_modis
;routine documentation

compile_opt idl2

;error catcher:
CATCH, theError
IF theError NE 0 THEN BEGIN
  CATCH, /cancel
  HELP, /LAST_MESSAGE, OUTPUT=errMsg
  FOR i=0, N_ELEMENTS(errMsg)-1 DO print, errMsg[i]
  RETURN
ENDIF

;format input parameters as strings and zero-fill

;build the partial filename

;build url path from filename parameters

;instantiate a new IDLnetURL object and connect to the server

;return a directory listing from the current ftp path
```

```
;search for and store "long" file name of desired image

;change directories to the desired file

;"get" the file

;object cleanup

END
```

Let's see what we've got cooking here in the comments. First, we start formatting our parameters as strings, and then we'll do whatever "zero-fill" means, so we should probably set up some input parameters. Keeping things in order from the naming convention, we'll set the product[2] , year, Julian day, horizontal, and vertical grid indices:

```
PRO ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid
```

Now we have access to all of the parameters of our filename as passed to us by the user. What format will all of these inputs be in? Our product name will be forced to come in as a string, but the next four inputs will probably be long integers by default. Since we need everything to be formatted as a string, we'll make sure our product is in all caps and set up our year variable:

```
;format all input parameters as strings, zero−filling as necessary
prod = STRUPCASE(imgProd)
year = STRING(imgYear)
```

Our next parameters aren't quite so simple. Remember that the grid indices are zero-padded to the left? (h00v05) It turns out our Julian date parameter is also zero-padded to three places[3] We'll need to find a way to format our inputs so that our long integers become strings that fill precisely two or three spaces, filling in zeros as needed. Let's write a quick helper function [4] to handle this:

```
FUNCTION ess_download_modis_zerofill, strIn, fill
;function documentation

compile_opt idl2

;format strIn by converting to a string and removing all leading spaces

;determine the number of leading zeros needed

;conditionally add leading zeros to the parameter

;return the result

END

PRO ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid
```

[2] Remember that "Product" is a reserved word in IDL

[3] e.g. an image on January 9 gives us MOD09A1.A2008009.h20v02.005.2008020034715

[4] While we're writing this as a helper function for this procedure, note that there's nothing MODIS-specific that we'll be doing with this function. It may be worth it to save this function as its own file to be used in other settings as well.

Here we have a function with two inputs, the string we need to format, and the fill number. As it turns out, the STRING() typecasting function adds some annoying spaces at the beginning of your output. Let's use STRTRIM to remove them:

```
;format strIn by converting to a string and removing all leading spaces
strIn = STRTRIM( STRING(strIn),1 )
```

Next, let's compare how long this formatted string is with how many spaces the caller wants filled:

```
;determine the number of leading zeros needed
zf = fill - STRLEN(strIn)
```

Now let's think about how to conditionally add zeros to our string based on this fill. A couple of IF's should do the trick: zf should tell us how many zeros to concatenate. If zf is zero, that means our string is good to go, so we'll leave it alone and return it.

```
;conditionally add leading zeros to the parameter
IF zf EQ 2 THEN zFill = '00' + strIn $
  ELSE IF zf EQ 1 THEN zFILL = '0' + strIn $
  ELSE zFill = strIn

;return the result
RETURN, zFill

END
```

Excellent. Now back to our procedure, let's format the date and grid indices using our new helper function: Let's add a temporary print statement to make sure we got it right, too:

```
;format all input parameters as strings, zero-filling as necessary
prod = STRUPCASE(imgProd)
year = STRING(imgYear)
date = ess_download_modis_zerofill(imgDate, 3)
hInd = ess_download_modis_zerofill(imgHgrid, 2)
vInd = ess_download_modis_zerofill(imgVgrid, 2)
print, prod, year, date, hInd, vInd
```

```
IDL> ess_download_modis, 'MOD09A1', 2001, 49, 10,5
```

Formatting is a little funky, but it sure looks like we got it right. Now let's put all of these strings together. At first glance, it may seem like a simple concatenation operation between our variables and a bunch of dots. However, because of the way typecasting works, sometimes we end up with extra spaces in places we don't want. Here, if we run:

```
print, prod + '.A' + year + date + '.h' + hInd + 'v' + vInd
```

IDL would return MOD09A1.A   2001049.h10v05. Not what we wanted. Let's try a combination of STRSPLIT and STRJOIN. STRSPLIT will cut through any excess whitespace in a string and return each element as an independent string with the /EXTRACT keyword. STRJOIN will the re-concatenate these for us, much like the + operator:

```
;build the partial filename
partialFname = STRJOIN(STRSPLIT(prod + '.A' + year + date + '.h' $
  + hInd + 'v' + vInd, /EXTRACT) )
```

Run a quick print statement to check your output. With the same parameters, you should end up with something like MOD09A1.A2001049.h10v05. Now that we have a partial filename, let's get connected and search for our images!

17.4 Communicating with the outside: IDLnetURL

Our primary tool for downloading these images will be the IDLnetURL package. Documentation for these objects can be found here `http://www.exelisvis.com/docs/IDLnetURL.html` In order to connect and download the files we need, we'll instantiate an IDLnetURL object, set its parameters, and then call the various methods included in the object to interact with the server.

The FTP server we'll be using is `ftp://ladsftp.nascom.nasa.gov` Feel free to navigate there in a browser and explore. The data we need are located in the directory \allData\5. It is important to get familiar with the directory structure. The data are sorted into product name, year, and Julian date directories. After that, we'll have to sort through a page of file names sorted by their grid indices to pick out the images we want. For testing, pick out a MOD09A1 file and use its parameters. This way, you can guarantee that the image exists when you start talking to the server.

Now, let's set up some of our connection parameters. For the host, we'll leave out the beginning "ftp://" because we have a property that we can set to tell the object that we're using an FTP instead of a URL. To set the path, we'll dance the same jig we did building the partial filename, this time leaving out the grid indices.

```
;build url path from filename parameters
urlHost = 'ladsftp.nascom.nasa.gov'
urlPath = STRJOIN(STRSPLIT('allData/5/' + prod + '/' $
  + year + '/' + date + '/', /EXTRACT) )
```

Now, we have everything we need to instantiate our IDLnetURL object. The OBJ_NEW() function creates a new object and allows you to set properties, much like keywords for a procedure. There are built-in methods to set each property one-by-one, but this tends to be tedious for more that a few properties. Let's set them all at once and then discuss:

```
;instantiate a new IDLnetURL object and connect to the server
oUrl = OBJ_NEW('IDLnetUrl', $
                URL_SCHEME='ftp', $
                FTP_CONNECTION_MODE=0, $
                URL_HOST=urlHost, $
                URL_Port=21, $
                URL_USERNAME='anonymous', $
                URL_PASSWORD='', $
                URL_PATH=urlPath)
```

There's a lot going on here, so let's break it down. We've created a new variable, oUrl, to be our object. The first property of our function sets the object type: here, we need an 'IDLnetURL' object. Next, we set the URL_SCHEME property to 'ftp.' CONNECTION_MODE gets set to 0 which, per the documentation, establishes a passive connection. Certain servers require a passive connection and ladsftp is one of them. We've already prepared our URL_HOST property, so we just plug it in here.

The default Transmission Connection Port (TCP) for sending and receiving files from an FTP server is 21, so we'll set it here just in case. All ftp servers require a login, but ladsftp, like many other servers, is an "Anonymous" server. This means that a secure username and password are not required for accessing files for downloading. Instead, your browser automatically logs you in as username *anonymous* with a null character password whenever you browse there. For IDLnetURL, the process is not automated, so we'll supply the appropriate "credentials" ourselves.

Lastly, we'll plug in the URL path that we've already created. You may also want to add the property VERBOSE=1 in order to see any responses from the server as you poke around on the server.

This all may sound well and good, but does it actually work? In order to test, let's create a little main level program with some hard-coded values. This should do the trick:

```
END

url = 'allData/5/MOD09A1/2001/049/MOD09A1.A2001049.h10v05.005.2006362053450.hdf'
host = 'ladsftp.nascom.nasa.gov'

oUrl = OBJ_NEW('IDLnetUrl', $
                URL_SCHEME='ftp', $
                FTP_CONNECTION_MODE=0, $
                URL_HOST=host, $
                URL_Port=21, $
                URL_USERNAME='anonymous', $
                URL_PASSWORD='', $
                URL_PATH=url)

void = oURL->Get(FILENAME='test.hdf')

OBJ_DESTROY, oURL

END ; main level program
```

Now, a couple of things to note here. Most striking should be this new OBJ_DESTROY command. This step is crucial because we need to close all of our connections and get rid of any new objects we create. It may not seem necessary for a simple routine like this, but in larger projects it becomes much easier to create memory leaks that will crash your program. Good programming practice dictates that we destroy objects as soon as they've fulfilled their purpose, so we'll do the same here.

Once we instantiated our object, we created a new variable called void and assigned it the value of one of the methods in our IDLnetURL object. The Get() method downloads the file specified in the URL_PATH property and saves it to IDL's current working directory. We supplied a separate filename for the image we want to save to our disk. In our procedure, we'll make sure this name matches the full MODIS filename for organizational purposes. Get() also accepts an optional keyword, URL, to which you can set to be the entire path (think URL_HOST + URL_PATH + filename) and download that way in case you didn't want to set your URL_PATH variable to include the image you want to download. [5]

But wait a minute, what about our void variable? What value does it actually receive? Since we're at the main level, you can run a help, void from the command line:

[5] For our purposes, we'll just set URL_PATH to include our image. For a simple Connect>Download>Disconnect task like this, it makes sense just to set the path for each image, rather than juggling and concatenating even more strings.

```
IDL> help, void
VOID            STRING    =    '\\uahdata\rhome\ATS509\IDLPro\test.hdf'
```

Oh, it stores the location of our new image file. Sweet! This could be useful for any number of reasons - error checking, generating reports, etc. Feel free to follow that path and make sure you have your file.

Speaking of, the filename we hard-coded for this test includes the collection number, processing date, and the .hdf file extension - all parameters that we don't have (or even know how to get). At this point, our procedure is not yet able to recreate these steps to download the same file. To do this, we will need to follow a few more steps and interact with the server a little.

17.5 Browsing the server

Now that we've established a connection, we'll need to take a look around and get our bearings. Remember what we set our URL_PATH property to:

```
urlPath = STRJOIN(STRSPLIT('allData/5/' + prod + '/' $
  + year + '/' + date + '/', /EXTRACT) ) ;allData/5/MOD09A1/2001/049/
```

To double-check that your path is correct for the current example, try adding these lines after you instantiate oUrl:

```
oUrl->GetProperty, URL_PATH=checkPath
help, checkPath
```

You can use this GetProperty method for many other object properties in IDLnetURL. As we move forward, we'll be changing certain properties around (such as the path), so having a tool to check in on our properties is convenient.

Now that we know where we are, let's focus on the problem at hand. Remember that we now have a partial filename based on the inputs supplied by the caller. What we need to do is somehow search the server for an image that matches our partial filename, collect some information about this image, and download it. We've navigated to the correct product, year, and date directories on the server, and are now sitting in a directory that is (theoretically) full of .hdf files.

Let's confirm this theory by calling another method of our IDLnetURL object. GetFtpDirList works similarly to an ls command in a UNIX environment. Here, it will return a long listing of the files in the current directory (URL_PATH) Since we expect this directory to be full of files, let's take a peek at the first few elements of this list:

⟶ ⟵

```
;instantiate a new IDLnetURL object and connect to the server
oUrl = OBJ_NEW('IDLnetUrl', $
                URL_SCHEME='ftp', $
                FTP_CONNECTION_MODE=0, $
                URL_HOST=urlHost, $
                URL_Port=21, $
                URL_USERNAME='anonymous', $
                URL_PASSWORD='', $
                URL_PATH=urlPath)

;return a directory listing from the current ftp path
result = oUrl->GetFtpDirList()
```

```
print, result[0:9]
```

Here is an example of what that output might look like:

```
-r-xr-xr-x  1  ftp  ftp  54165591 Feb 10 2011 MOD09A1.A2001049.h00v08.005.2006364114637.hdf
-r-xr-xr-x  1  ftp  ftp  53501118 Feb 10 2011 MOD09A1.A2001049.h00v09.005.2006364115902.hdf
-r-xr-xr-x  1  ftp  ftp  24101191 Feb 10 2011 MOD09A1.A2001049.h00v10.005.2006364112904.hdf
-r-xr-xr-x  1  ftp  ftp  61472350 Feb 10 2011 MOD09A1.A2001049.h01v07.005.2006364152441.hdf
-r-xr-xr-x  1  ftp  ftp  56651348 Feb 10 2011 MOD09A1.A2001049.h01v08.005.2006364115207.hdf
-r-xr-xr-x  1  ftp  ftp  55182909 Feb 10 2011 MOD09A1.A2001049.h01v09.005.2006364120657.hdf
-r-xr-xr-x  1  ftp  ftp  59442305 Feb 10 2011 MOD09A1.A2001049.h01v10.005.2006364134737.hdf
-r-xr-xr-x  1  ftp  ftp  21239538 Feb 10 2011 MOD09A1.A2001049.h01v11.005.2006364131836.hdf
-r-xr-xr-x  1  ftp  ftp  51184765 Feb 10 2011 MOD09A1.A2001049.h02v06.005.2006364100613.hdf
-r-xr-xr-x  1  ftp  ftp  57341015 Feb 10 2011 MOD09A1.A2001049.h02v08.005.2006364115613.hdf
```

Nice, so there are files in this directory! Now we need to find a way to programmatically "grab" that filename so we can append it to our path and download the image with our Get() method. If your String-ey Senses are tingling, you might use our STRSPLIT function to cut this list up and pull out the last element of each line. I like your enthusiasm, but you'd be working too hard. The documentation points out that our GetFtpDirList method can accept an optional keyword /SHORT. Let's check it out:

```
result = oUrl->GetFtpDirList(/SHORT)
print, result[0:9]
```

```
MOD09A1.A2001049.h00v08.005.2006364114637.hdf
MOD09A1.A2001049.h00v09.005.2006364115902.hdf
MOD09A1.A2001049.h00v10.005.2006364112904.hdf
MOD09A1.A2001049.h01v07.005.2006364152441.hdf
MOD09A1.A2001049.h01v08.005.2006364115207.hdf
MOD09A1.A2001049.h01v09.005.2006364120657.hdf
MOD09A1.A2001049.h01v10.005.2006364134737.hdf
MOD09A1.A2001049.h01v11.005.2006364131836.hdf
MOD09A1.A2001049.h02v06.005.2006364100613.hdf
MOD09A1.A2001049.h02v08.005.2006364115613.hdf
```

Much better, that's exactly what we needed. Now we have an array of strings that stores all of the "full" MODIS filenames for the given date. Searching through the list to find a substring may prove tricky, so let's pack up and head down to the main level again to work our our algorithm here: [6]

```
url = 'allData/5/MOD09A1/2001/049/'
host = 'ladsftp.nascom.nasa.gov'

oUrl = OBJ_NEW('IDLnetUrl', $
                  URL_SCHEME='ftp', $
                  FTP_CONNECTION_MODE=0, $
                  URL_HOST=host, $
                  URL_Port=21, $
                  URL_USERNAME='anonymous', $
```

[6]Note the shortened URL. We're trying to build this full path on our own, so we're going to start backing off with the hard-coding.

```
                  URL_PASSWORD='', $
                  URL_PATH=url)

result = oURL->GetFtpDirList(/SHORT)
print, result[0:9]

;void = oURL->Get(FILENAME='test.hdf')

OBJ_DESTROY, oURL

END ; main level program
```

Okay, now our main level program is caught up to speed. We've commented out the Get() method because it won't work for us at the moment. It looks like we'll need to compare our partial filename with a substring of the elements of our results variable. We'll need to use this comparison to find *where* our match occurs in this list, and then use that information to attain the rest of our filename.

There are countless functions and routines for string operations in IDL. A little searching reveals that STRMID [7] might be the perfect candidate for the job. The documentation indicates that STRMID takes a string, a position, and a length as its inputs. Let's run a test case on our file list to see how it handles an array:

```
result = oURL->GetFtpDirList(/SHORT)
;print, result[0:9]

ind = STRMID(result, 0, 7)
print, ind[0:9]
```

Gives us:

```
MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1 MOD09A1
```

This is perfect! For each element of our array, STRMID takes the 0th element and returns a substring that is 7 characters long. Let's start counting and expand this call to return an array of all "partial" filenames:

```
result = oURL->GetFtpDirList(/SHORT)
;print, result[0:9]

ind = STRMID(result, 0, 23)
print, ind[0:9]
```

Returns:

```
MOD09A1.A2001049.h00v08    MOD09A1.A2001049.h00v09    MOD09A1.A2001049.h00v10    ...
```

Looking good, it's incredibly easy to search this entire list programmatically now. Let's wrap a WHERE statement around this function and determine where (or if) a match occurs:[8]

```
url = 'allData/5/MOD09A1/2008/193'
host = 'ladsftp.nascom.nasa.gov'
partial = 'MOD09A1.A2001049.h10v05'
```

[7] http://www.exelisvis.com/docs/STRMID.html

[8] Notice that a new variable, "partial," was hard-coded to simulate previous steps.

```
oUrl = OBJ_NEW('IDLnetUrl', $
                  URL_SCHEME='ftp', $
                  FTP_CONNECTION_MODE=0, $
                  URL_HOST=host, $
                  URL_Port=21, $
                  URL_USERNAME='anonymous', $
                  URL_PASSWORD='', $
                  URL_PATH=url)

result = oURL->GetFtpDirList(/SHORT)
;print, result[0:9]

ind = WHERE(partial EQ STRMID(result, 0, 23), count)
print, count, ind

;void = oURL->Get(FILENAME='test.hdf')

OBJ_DESTROY, oURL

END ; main level program
```

Which returns to us (1, 49) - one match located at position 49! I think we can handle things from here. Let's scroll back up to our procedure and add this code.

```
;return a directory listing from the current ftp path
result = oUrl->GetFtpDirList(/SHORT)

;search for and store "long" file name of desired image
ind = WHERE(partialFname EQ STRMID(result, 0, 23), count)
fName = result[ind]
```

This works, but let's practice good programming and avoid hard-coding any values. This way, if we come across a product that has more or less than 23 characters in it's partial filename, we won't run into trouble.

```
;search for and store "long" file name of desired image
ind = WHERE(partialFname EQ STRMID(result, 0, STRLEN(partialFName)), count)
fName = result[ind]
```

Great, now we should have everything we need. Let's change directories to the location of the image we want to download:

```
;change directories to the desired file
oUrl->SetProperty, URL_PATH = urlPath + fName
```

Uh-oh, what's going on here? IDL is telling us that our expression for setting this property must be a scalar. Wasn't fName a scalar string variable? A quick 'help' down in our main level reveals that fName is in fact a string array with one element. Frustrating, but we can make it work:

```
;change directories to the desired file
oUrl->SetProperty, URL_PATH = urlPath + fName[0]
```

Good to go. Now, *finally* after all of that hassle, we're looking directly at the image we want to download. Let's get it:

```
;"get" the file
void = oURL->Get(FILENAME=fName)
```

Wait...what? Now you're getting angry - you put in all of that effort, IDL off-ed herself, and you still don't have your image. Let's take another look at our code. We just called the "Get" method at our current URL_PATH. Our FILENAME keyword was set to fName. Wait a minute, fName is an array with one element, not a scalar! As it turns out, this is yet another scenario where singleton arrays are not the same as scalars. Here, we caused a segmentation fault by trying to create a file with a vector for a filename. You freaked out, IDL freaked out, your operating system freaked out, it was a bad time. Let's apply the same fix here:

```
;"get" the file
void = oURL->Get(FILENAME=fName[0])
```

That should do the trick. Run your code again, then check your current working directory in IDL for your new file. It should be noted that you can also supply an absolute path for the FILENAME keyword as well if you like a little more control over how you organize things.

One more thing: don't forget to destroy your object. It wasn't a big deal for one image, but we're about to start running multiple instances of this object and don't want to cause any memory leaks.

```
;object cleanup
OBJ_DESTROY, oUrl
```

17.6 Downloading multiple files

At this point, we've written a routine that can handle downloading a single MODIS file based on five parameters supplied by the caller. You're probably thinking that, for all the work we've put into this, our methodology isn't very practical. Think about it, you could seriously just click the link on the FTP server in your browser. Or better yet, why not just log in to GLOVIS like a normal person? And what happens if the user inputs parameters for a file that doesn't exist? [9]

Truth be told, we haven't done a lot of testing or error checking because up to this point, our goal was to make sure we could even download a *single* file. Remember that the goal of this project was to automate downloading *multiple* images.

For our next task, we're going to write a script that uses our old code to iteratively make attempts to download images. Imagine having a tool that can download all of the MODIS tiles over the Gulf of Mexico for a certain product over the past two years! In order to help us stay organized, let's have IDL generate a report listing all of the files we've downloaded in a single batch process. To do this, we might want to convert our ess_download_modis procedure into a function. That way, we can return our filename variable to the script that calls our function so we can use it later.

$\longrightarrow$ $\longleftarrow$

```
FUNCTION ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid
;function documentation

compile_opt idl2

;error catcher:
```

[9] Try running your routine with ['MOD09A1', 2001, 48, 10, 5] and see what happens. Yuck!

```
CATCH, theError
IF theError NE 0 THEN BEGIN
  CATCH, /cancel
  HELP, /LAST_MESSAGE, OUTPUT=errMsg
  FOR i=0, N_ELEMENTS(errMsg)-1 DO print, errMsg[i]
  RETURN,!VALUES.F_NAN
ENDIF
```

Also, don't forget to return a value. fName could be very useful for generating a report later on, so we'll return it.

```
;Return the filename to the caller
RETURN, fName[0]
```

Let's try calling our function from the main level for some testing.

```
RETURN, fName[0]

END ; ess_download_modis

temp = ess_download_modis('MOD09A1', 2001, 2, 10, 5)
help, temp

END ; main level program
```

As we hinted earlier, the error codes for attempting to downloading nonexistant files can get pretty nasty. As you scroll through the directories on the ftp site, you'll notice that a MOD09A1 product is produced every 8 days. If we wanted to download a span of days, we could try to calculate which days will sport a new image. However, this method would only work for the 8-day MODIS products and could prove to be very inaccurate if there was any sort of offset.

Instead, why don't we improve our error management in ess_download_modis and see if we can convince IDL to handle these server rejections more gracefully. Looking at the error that IDLnetURL put out for that nonexistant file, it seems that very little of that information is actually important to us. Think about it, if we want to run a batch process to download all MODIS images within a month for a particular grid index, we will probably use a loop to iteratively call our function to pull those images down. As our counter increases, we're bound to make a function call for nonexistant images, and that's okay. The network doesn't mind if we make failed requests, so why not let our errors bounce off quietly rather than crash our program?

Our current error-checker is, erm, verbose in its description of the problem. This is useful for finding syntax errors and the like, but there's nothing keeping us from adding a little *ad hoc* error-checker for networking bounces. Add a few breakpoints throughout your ess_download_modis function to find where we crash. It seems that the error is coming from trying to navigate to the directory of a date that doesn't exist. That means our culprit the code block where we instantiate oUrl as an IDLnetURL object.

Let's add our new error-catcher right after those lines. Something like this should do the trick:

```
;instantiate a new IDLnetURL object and connect to the server
oUrl = OBJ_NEW('IDLnetUrl', $
  URL_SCHEME='ftp', $
  FTP_CONNECTION_MODE=0, $
  URL_HOST=urlHost, $
  URL_Port=21, $
  URL_USERNAME='anonymous', $
  URL_PASSWORD='', $
  URL_PATH=urlPath)

;network error-catcher
CATCH, connectionError
IF connectionError NE 0 THEN BEGIN
  CATCH, /cancel
  print, 'Image for date ', date, ',', year, ' does not exist'
  RETURN, !VALUES.F_NAN
ENDIF

;return a directory listing from the current ftp path
result = oUrl->GetFtpDirList(/SHORT)
```

Fun fact: our main error-catcher will still work and handle errors elsewhere throughout the program! Test again with an invalid date to see if it works.

Now let's continue along with our main level program. We want to write a loop that will iterate through a specified range of dates and make function calls along the way. It would also be nice to keep track of all of the files we manage to acquire as well. Since we're returning our filename each time we call the function, we'll have to find a place to store them during each iteration. We could add them to an array, but if we want our code to be flexible, we don't want to define a set size for an array of filenames.

Let's define another type of object, the list. Lists behave similarly to arrays, but their memory is dynamically allocated, meaning we will start with an empty list and make room as needed.

First, let's create some variables to set up our inputs and instantiate a list. Remember that now, we'll want to define a start date and an end date to control our loop with:

```
;temp = ess_download_modis('MOD09A1', 2001, 2, 10, 5)
;help, temp

;define inputs
prod = 'MOD09A1'
year = 2001
jStartDate = 1
jEndDate = 26
hGridInd = 10
vGridInd = 5

;instantiate filename list
hdf = LIST()

END ; main level program
```

Now, we'll set up our loop. We'll want to loop from our start date until our end date. During each iteration of the loop, we'll want to call our download function, storing the output in a temporary variable.

```
hdfList = LIST()

FOR i = jStartDate, jEndDate DO BEGIN
     temp = ess_download_modis(prod, year, LONG(i), hGridInd, vGridInd)
    hdfList->ADD, temp
```

Let IDLnetURL do it's thing and then help and print 'hdfList' once IDL gives you back the command line. Well this is interesting. Our function is spitting out NaN's any time the image isn't found.[10] It would be really nice to remove these NaN's from our hdfList and instead keep a separate count of how many failures have occurred.

To do this, we'll check the value of temp before we add it to hdfList to ensure that only legitimate filenames make it on the list. How do we check whether a variable is NaN or not? IDL Coyote offers some sage advice here: `http://www.idlcoyote.com/math_tips/nans.html` It seems that NaN's are literally not equal to anything - even themselves. So all we have to do is check to see if temp equals temp. We'll throw that counter in here, too:

```
;set counter for failed downloads
failCount = 0

FOR i = jStartDate, jEndDate DO BEGIN
    temp = ess1_download_modis(prod, year, LONG(i), hGridInd, vGridInd)
    IF TEMP EQ TEMP THEN hdfList->ADD, temp ELSE failCount += 1
ENDFOR
```

17.7 Downloading both Aqua and Terra Images

We mentioned briefly in our introduction to the MODIS naming convention that many images from the Aqua and Terra satellites have the same parameters. Wouldn't it be cool if we could set a keyword to download both Aqua and Terra images for the same day (if both are avaiable)?

As it turns out, this is actually a pretty simple task. However, it requires an advanced programming technique called recursion to work. Recursion involves a piece of code, usually a function, making a call to itself. This creates sort of a forked loop in your code that requires control statements to prevent things from going haywire. IDL Coyote has a short and to the point recursion example here: `http://www.idlcoyote.com/tips/recursive_function.html`

$\longrightarrow$ ——— $\longleftarrow$

```
FUNCTION ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid
;function documentation

;enable recursion for this function
FORWARD_FUNCTION ess_download_modis

compile_opt idl2
```

$\longrightarrow$ ——— $\longleftarrow$

The "FORWARD_FUNCTION" modifier notifies IDL that this function may have to call itself. Now, let's think about how we're going to tell our code to switch satellites. This will be some form of string operation to switch the "O" in

[10] We told it to do this. Double check our error-catcher.

our product name to a "Y", or verse-visa. Let's run a check to compare the first three letters to "MOD" and then reassign those as necessary. **DO NOT** run the following code. It isn't safe yet for reasons we will soon discuss.

```
alt = (STRMID(imgProd,0,3) EQ 'MOD' ) ? 'MYD' : 'MOD'
alt = alt + (STRMID(imgProd, 3, STRLEN( imgProd ) ) )
downloadAlt = ess_download_modis(alt, imgYear, imgDate, imgHgrid, imgVgrid)
```

Remember our ternary operator? [11] So in this code block, we define a string variable, alt, to be either "MYD" or "MOD" depending on the first three letters of imgProd. [12] Then we reassign alt to include our new satellite designation with the following product numbers of our original image. Finally, we recursively call our ess_download_modis function using our new "alt" product string as the input.

What would happen if you ran this code now? Assuming we place this block at the top of our function, we would infinitely alternate between "MOD" and "MYD" without ever reaching the rest of our code. Eventually, we'd run out of memory and our computer would crash. So how do we control this? Let's at a few conditions for our function to check before it makes any dangerous infinite recursive calls. We'll start by adding a function keyword to control whether we want the alternate satellite image or not:

```
FUNCTION ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid, $
                              dualSat=dsat
;function documentation
```

Next, let's wrap all of this around an IF clause to control whether or not our code will even consider using recursion.

```
IF dsat THEN BEGIN
  alt = (STRMID(imgProd,0,3) EQ 'MOD' ) ? 'MYD' : 'MOD'
  alt = alt + (STRMID(imgProd, 3, STRLEN( imgProd ) ) )
  downloadAlt = ess_download_modis(alt, imgYear, $,
                     imgDate, imgHgrid, imgVgrid)
ENDIF

;format input parameters as strings and zero-fill
prod = STRUPCASE(imgProd)
```

Now, logically, this code will only ever be executed if our dualSat keyword is set to a nonzero value. Just to be safe, let's explicitly set our keyword to 0 in the recursive function call.

```
downloadAlt = ess_download_modis(alt, imgYear, $,
                     imgDate, imgHgrid, imgVgrid, dualSat=0)
```

And done! Now in our main level program, if we want images from both satellites, simply set the dualSat keyword to 1. The cool thing is that the loop in our main level program will always have the keyword set, while none of the recursive calls will. This means our loop will pull all of the images from both satellites for each date specified *without* crashing our program.

[11] http://physics.nyu.edu/grierlab/idl_html_help/expressions4.html#wp1009778

[12] What would happen if the user input an "MCD" MODIS product? The server simply woulnd't find any images from Aqua and carry on with its assigned task.

17.8 Code Example

```
; docformat = 'rst'

;+
; This file consists of two functions and a main level program
; designed to download MODIS images from `<ftp://ladsftp.nascom.nasa.gov/allData$
  />`
; using the IDLnetURL object class.
;
; :Author:
;    Tim Klug
;
; :History:
;    tklug, 3 Dec 2015: initial template
;    tklug, 9 Dec 2015: completed batch processing, error catching, satellite $
  designation
;
; :Copyright:
;    (c) 2015, Tim Klug <tim.klug@uah.edu>
;
;    All rights reserved.
;-

FUNCTION ess_download_modis_zerofill, strIn, fill
;+
; Helper function for main ess_download_modis routine.
; This routine formats string parameters of MODIS filenames
; to zero-fill numbers to a designated length.
;
; :Params:
;   strIn : in, required, type=string
;     the input spring to be zero-filled
;   fill : in, required, type=integer scalar
;     the number of characters the output must fill
; :Keywords:
;   none
; :Uses:
;   none
;-

compile_opt idl2

;format strIn by converting to a string and removing all leading spaces
strIn = STRTRIM( STRING(strIn),1 )

;determine the number of leading zeros needed
zf = fill - STRLEN(strIn)

;conditionally add leading zeros to the parameter
IF zf EQ 2 THEN zFill = '00' + strIn $
  ELSE IF zf EQ 1 THEN zFILL = '0' + strIn $
  ELSE zFill = strIn
```

```
;return the result
RETURN, zFill

END ; ess_download_modis_zerofill


FUNCTION ess_download_modis, imgProd, imgYear, imgDate, imgHgrid, imgVgrid, $
                  dualSat=dsat

;+
; This function formats input parameters into a partial MODIS
;   filename, instantiates a IDLnetURL object, searches directories
;   for an image matching the partial filename, and downloads the
;   image. Recursive calls are enabled to allow for downloading both
;   Aqua and Terra MODIS images.
;
; :Params:
;   imgProd : in, required, type=string
;     the MODIS product name of the image to be downloaded
;   imgYear : in, required, type=integer scalar
;     the year of the MODIS image to be downloaded
;   imgDate : in, required, type=integer scalar
;     the Julian calendar date of the MODIS image to be downloaded
;   imgHgrid : in, required, type=integer scalar
;     the horizontal grid cell of the MODIS image to be downloaded
;   imgVgrid : in, required, type=integer scalar
;     the vertical grid cell of the MODIS image to be downloaded
;
; :Keywords:
;   dualSat : in, optional, type=integer scalar
;     enables recursive call to ess_download_modis to download
;     additional image from satellite counterpart
; :Uses:
;   ess_download_modis_zerofill
;-

;enable recursion for this function
FORWARD_FUNCTION ess_download_modis

compile_opt idl2

;error catcher:
CATCH, theError
IF theError NE 0 THEN BEGIN
  CATCH, /cancel
  HELP, /LAST_MESSAGE, OUTPUT=errMsg
  print, ''
  print, errMsg
  print, ''
  FOR i=0, N_ELEMENTS(errMsg)-1 DO print, errMsg[i]
  RETURN, !VALUES.F_NAN
ENDIF
```

```
IF dsat THEN BEGIN
  alt = (STRMID(imgProd,0,3) EQ 'MOD' ) ? 'MYD' : 'MOD'
  alt = alt + (STRMID(imgProd, 3, STRLEN( imgProd ) ) )
  downloadAlt = ess_download_modis(alt, imgYear, $,
                     imgDate, imgHgrid, imgVgrid, dualSat=0)
ENDIF

;format input parameters as strings and zeroâĹŠfill
prod = STRUPCASE(imgProd)
year = STRING(imgYear)
date = ess_download_modis_zerofill(imgDate, 3)
hInd = ess_download_modis_zerofill(imgHgrid, 2)
vInd = ess_download_modis_zerofill(imgVgrid, 2)

;build the partial filename
partialFname = STRJOIN(STRSPLIT(prod + '.A' + year + date + '.h' $
  + hInd + 'v' + vInd, /EXTRACT) )

;build url path from filename parameters
urlHost = 'ladsftp.nascom.nasa.gov'
urlPath = STRJOIN(STRSPLIT('allData/5/' + prod + '/' $
  + year + '/' + date + '/', /EXTRACT) )

;instantiate a new IDLnetURL object and connect to the server
oUrl = OBJ_NEW('IDLnetUrl', $
  URL_SCHEME='ftp', $
  FTP_CONNECTION_MODE=0, $
  URL_HOST=urlHost, $
  URL_Port=21, $
  URL_USERNAME='anonymous', $
  URL_PASSWORD='', $
  URL_PATH=urlPath)

CATCH, connectionError
IF connectionError NE 0 THEN BEGIN
  CATCH, /cancel
  ;print, 'Specified file does not exist. Continuing...'
  print, 'Image for date ', date, ',', year, ' does not exist'
  RETURN, !VALUES.F_NAN
ENDIF

;return a directory listing from the current ftp path
result = oUrl->GetFtpDirList(/SHORT)

;search for and store "long" file name of desired image
ind = WHERE(partialFname EQ STRMID(result, 0, STRLEN(partialFName) ), count)
fName = result[ind]

;change directories to the desired file
oUrl->SetProperty, URL_PATH = urlPath + fName[0]

;"get" the file
void = oURL->Get(FILENAME=fName[0])
```

```
;object cleanup
OBJ_DESTROY, oUrl

RETURN, fName[0]

END ; ess_download_modis

;main level program

;define inputs
prod = 'MOD09A1'
year = 2003
jStartDate = 1
jEndDate = 7
hGridInd = 10
vGridInd = 5
ds = 1


;instantiate filename list
hdfList = LIST()

;set counter for failed downloads
failCount = 0

FOR i = jStartDate, jEndDate DO BEGIN
    temp = ess_download_modis(prod, year, LONG(i), hGridInd, vGridInd, dualSat=$
      ds)
    IF TEMP EQ TEMP THEN hdfList->ADD, temp ELSE failCount += 1
ENDFOR

END ; main level program
```

$\longrightarrow$ $\longleftarrow$