2MASS J, WISE 12 and WISE 22

Montage Comes to Python

ADASS 2018

2MASS J, WISE 12 and WISE 22

What is Montage?

- Command-line driven toolkit
- Same code base used for running under Unix, Macs, Windows, Javascript, Python
- Designed for sustainability (since 2002)
  - Written in ANSI-C.
  - No dependence on third-party *platform* dependent packages or databases.
  - No reliance on shared memory.

2MASS J, WISE 12 and WISE 22

Wide Applicability in Astronomy and IT

- **51 astronomy peer-reviewed citations 10/17 – 10/18**
- **Supported –**
  - **Detection of NEO’s**
  - **Observation Planning for JWST and NEOCAM**
  - **Creation of Citizen Science data products**
  - **AO instrument performance studies**
  - **Creation of finding charts at ESO**
- **132 IT peer-reviewed citations 10/17 – 9/18**
  - **Exemplar application for developing cyber-infrastructure.**

2MASS J, WISE 12 and WISE 22

ADASS Interactive Presentation

Our ADASS presentation was interactive but based on the same interfaces shown in the following slides.

The actual interactive Jupyter pages can be downloaded from

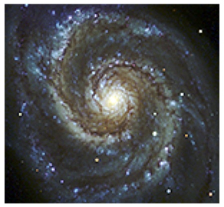
https://github.com/Caltech-IPAC/MontageNotebooks

or viewed here:

http://montage.ipac.caltech.edu/MontageNotebooks/

Both locations contain more information on installing Montage.

Using a Montage function in Python



Montage *is a general astronomical image toolkit with facilities for reprojection, background matching, coaddition and visualization. It can be used as a set of command-line tools (Linux, OS X and Windows), C library calls (Linux and OS X) and as Python binary extension modules.*

Montage source code can be downloaded from GitHub (https://github.com/Caltech-IPAC/Montage). The Python package can be installed from PyPI ("pip install MontagePy"). See http://montage.ipac.caltech.edu/ for more information.

MontagePy.main modules: mDiff

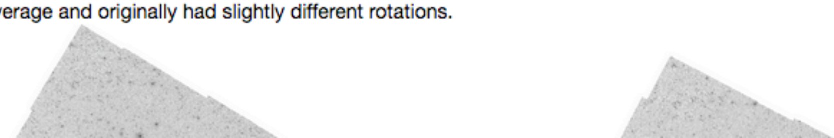
In [7]:

```
from IPython.display import Image
from MontagePy.main import mDiff, mViewer

help(mDiff)
```

```
Help on built-in function mDiff in module MontagePy.main:

mDiff(...)
    mDiff subtracts one image from another (both already in the same projection).

    Parameters
    ----------
    input_file1 : str
        First input file for differencing.
    input_file2 : str
        Second input file for differencing.  Files have to already have the same projection.
    output_file : str
        Output difference image.
    template_file : str
        FITS header file used to define the desired output.
    noAreas : bool, optional
        Do not look for or create area images as part of the differencing.
    factor : float, optional
        Optional scale factor to apply to the second image before subtracting.
```

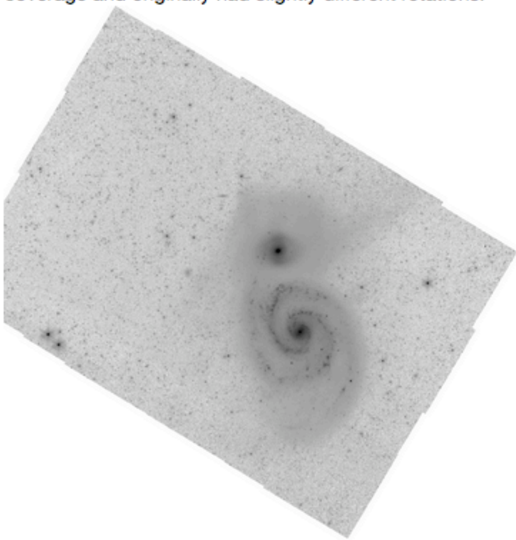
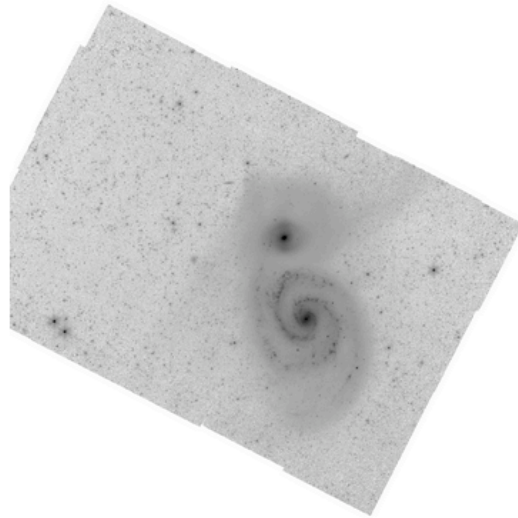
Example

Here, two Spitzer Space Telescope IRAC images have been tranformed to a common projection (using mProject). They have similar but not identical sky coverage and originally had slightly different rotations.



Example ¶

Here, two Spitzer Space Telescope IRAC images have been tranformed to a common projection (using mProject). They have similar but not identical sky coverage and originally had slightly different rotations.

mDiff gives us the difference image of the above two. In Montage,
this difference is primarily used to determine background offsets by fitting the difference with a plane. Below we stretch the difference to show the noise level and illustrate any residual effects of sky structure. Most noticeble are the bright stars in the lower left and the brightest areas in the galaxies, particularly the center of M51b. However, these are not much brighter than the noise and instrumental data collection effects, so fitting the difference with a plane (mFitplane) gives an reasonable measure of the image background difference, especially if the larger excursions from the plane are excluded in the fit.

In [2]:
```
mDiff("input/IRAC/projected/hdu0_SPITZER_I2_5504000_0000_6_E8758134_maic.fits",
      "input/IRAC/projected/hdu0_SPITZER_I2_5504256_0000_6_E8757228_maic.fits",
      "output/IRAC/irac_diff.fits",
      "input/IRAC/M51.hdr",
       noAreas=True)
```

Out[2]:
```
{'status': '0',
 'time': 0.0,
 'min_pixel': -0.33841609954833984,
 'max_pixel': 269.2658386230469,
 'min_diff': 1.4901161193847656e-08,
 'max_diff': 142.27021026611328}
```

The difference image looks like this:

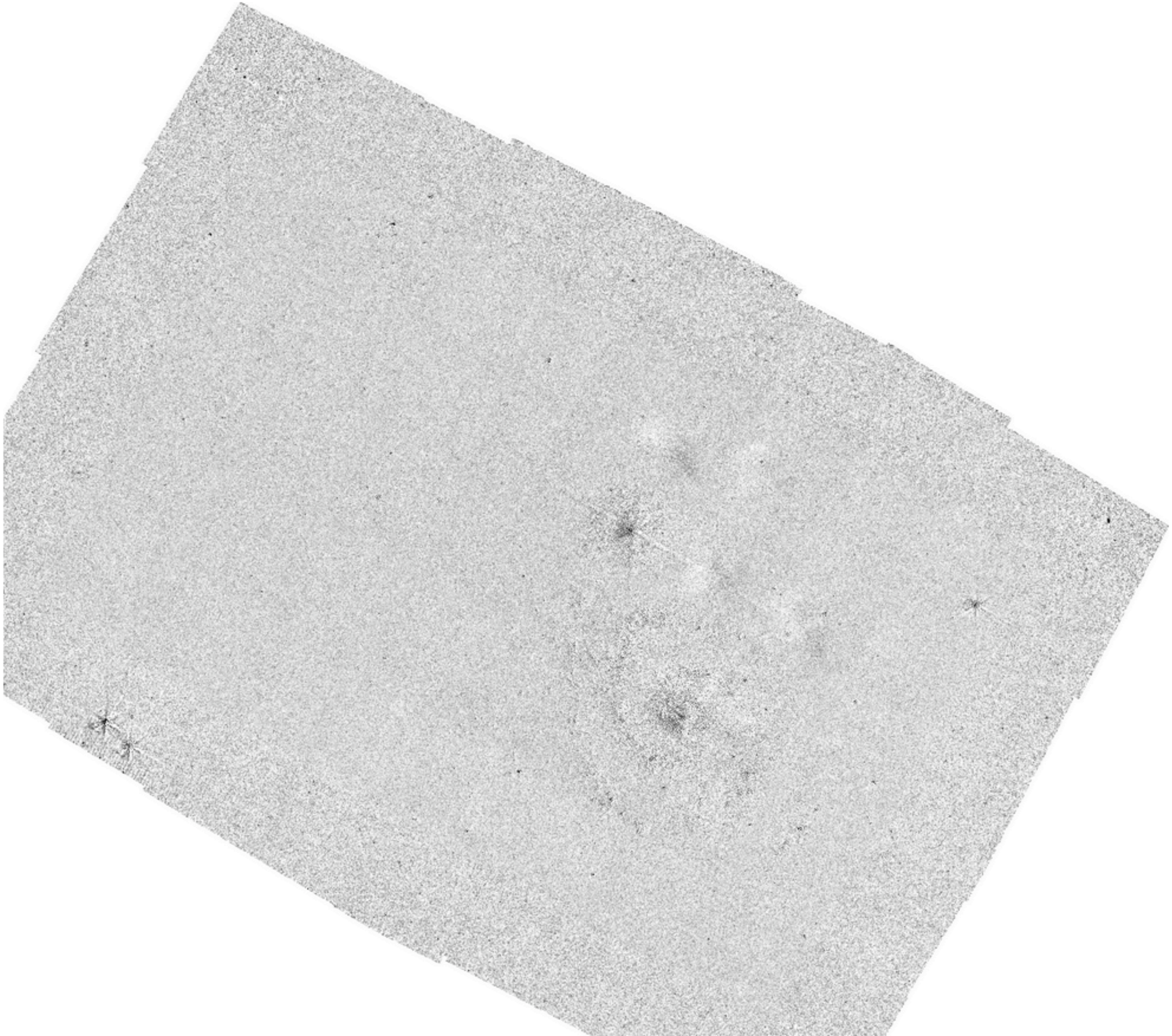
In [6]:
```
rtn = mViewer("-ct 1 -gray output/IRAC/irac_diff.fits -2s max gaussian-log -out output/temp.png",
              "", mode=2)

Image(filename='output/temp.png')
```

Out[6]:

The difference image looks like this:

In [6]:
```
rtn = mViewer("-ct 1 -gray output/IRAC/irac_diff.fits -2s max gaussian-log -out output/temp.png",
              "", mode=2)

Image(filename='output/temp.png')
```

Out[6]:



Error Handling

If mDiff encounters an error, the return structure will just have two elements: a status of 1 ("error") and a message string that tries to diagnose the reason for the error.

For instance, if the user specifies an image that doesn't exist:

In [8]:
```
mDiff("input/IRAC/projected/bad.fits",
      "input/IRAC/projected/hdu0_SPITZER_I2_5504256_0000_6_E8757228_maic.fits",
      "output/IRAC/irac_diff.fits",
      "input/IRAC/M51.hdr",
       noAreas=True)
```

Out[8]:
```
{'status': '1',
 'msg': b'Image file input/IRAC/projected/bad.fits missing or invalid FITS'}
```

Classic Montage: mDiff as a Stand-Alone Program

mDiff Unix/Windows Command-line Arguments

mDiff can also be run as a command-line tool in Linux, OS X, and Windows:

Usage: `mDiff [-d level] [-n(o-areas)] [-z factor] [-s statusfile] in1.fits in2.fits out.fits hdr.template`

If you are writing in C/C++, mDiff can be accessed as a library function:

```
/*-***********************************************************************/
/*                                                                       */
/*  mDiff                                                                */
/*                                                                       */
/*  Montage is a set of general reprojection / coordinate-transform /    */
/*  mosaicking programs.  Any number of input images can be merged into  */
/*  an output FITS file.  The attributes of the input are read from the  */
/*  input files; the attributes of the output are read a combination of  */
/*  the command line and a FITS header template file.                    */
/*                                                                       */
/*  This module, mDiff, is used as part of a background                  */
/*  correction mechanism.  Pairwise, images that overlap are differenced */
/*  and the difference images fit with a surface (usually a plane).      */
/*  These planes are analyzed, and a correction determined for each      */
/*  input image (by mBgModel).                                           */
/*                                                                       */
/*   char  *input_file1    First input file for differencing             */
/*   char  *input_file2    Second input file for differencing. Files     */
/*                         have to already have the same projection      */
```

```
/*                                                                   */
/*  mDiff                                                            */
/*                                                                   */
/*  Montage is a set of general reprojection / coordinate-transform /   */
/*  mosaicking programs.  Any number of input images can be merged into */
/*  an output FITS file.  The attributes of the input are read from the */
/*  input files; the attributes of the output are read a combination of */
/*  the command line and a FITS header template file.                */
/*                                                                   */
/*  This module, mDiff, is used as part of a background              */
/*  correction mechanism.  Pairwise, images that overlap are differenced */
/*  and the difference images fit with a surface (usually a plane).  */
/*  These planes are analyzed, and a correction determined for each  */
/*  input image (by mBgModel).                                       */
/*                                                                   */
/*   char  *input_file1    First input file for differencing         */
/*   char  *input_file2    Second input file for differencing. Files */
/*                         have to already have the same projection  */
/*                                                                   */
/*   char  *output_file    Output difference image                   */
/*                                                                   */
/*   char  *template_file  FITS header file used to define the desired */
/*                         output region                             */
/*                                                                   */
/*   int    noAreas        Do not look for or create area images as part */
/*                         of the differencing                       */
/*                                                                   */
/*   double factor         Optional scale factor to apply to the second */
/*                         image before subtracting                  */
/*                                                                   */
/*   int    debug          Debugging output level                    */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

struct mDiffReturn *mDiff(char *input_file1, char *input_file2, char *ofile, char *template_file,
                          int noAreasin, double fact, int debugin)
```

Return Structure

```
struct mDiffReturn
{
   int    status;        // Return status (0: OK, 1:ERROR)
   char   msg [1024];    // Return message (for error return)
   char   json[4096];    // Return parameters as JSON string
   double time;          // Run time (sec)
   double min_pixel;     // Minimum pixel value in either input (absolute)
   double max_pixel;     // Maximum pixel value in either input (absolute)
   double min_diff;      // Minimum pixel value in difference
   double max_diff;      // Maximum pixel value in difference
};
```

```
/*                                                                       */
/*  mDiff                                                                */
/*                                                                       */
/*  Montage is a set of general reprojection / coordinate-transform /    */
/*  mosaicking programs.  Any number of input images can be merged into  */
/*  an output FITS file.  The attributes of the input are read from the  */
/*  input files; the attributes of the output are read a combination of  */
/*  the command line and a FITS header template file.                    */
/*                                                                       */
/*  This module, mDiff, is used as part of a background                  */
/*  correction mechanism.  Pairwise, images that overlap are differenced */
/*  and the difference images fit with a surface (usually a plane).      */
/*  These planes are analyzed, and a correction determined for each      */
/*  input image (by mBgModel).                                           */
/*                                                                       */
/*   char  *input_file1    First input file for differencing             */
/*   char  *input_file2    Second input file for differencing. Files     */
/*                         have to already have the same projection      */
/*                                                                       */
/*   char  *output_file    Output difference image                       */
/*                                                                       */
/*   char  *template_file  FITS header file used to define the desired   */
/*                         output region                                 */
/*                                                                       */
/*   int    noAreas        Do not look for or create area images as part */
/*                         of the differencing                           */
/*                                                                       */
/*   double factor         Optional scale factor to apply to the second  */
/*                         image before subtracting                      */
/*                                                                       */
/*   int    debug          Debugging output level                        */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

struct mDiffReturn *mDiff(char *input_file1, char *input_file2, char *ofile, char *template_file,
                          int noAreasin, double fact, int debugin)
```

Return Structure

```
struct mDiffReturn
{
   int    status;        // Return status (0: OK, 1:ERROR)
   char   msg [1024];    // Return message (for error return)
   char   json[4096];    // Return parameters as JSON string
   double time;          // Run time (sec)
   double min_pixel;     // Minimum pixel value in either input (absolute)
   double max_pixel;     // Maximum pixel value in either input (absolute)
   double min_diff;      // Minimum pixel value in difference
   double max_diff;      // Maximum pixel value in difference
};
```

A Complete Mosaic in MontagePy

jupyter Mosaic (autosaved)

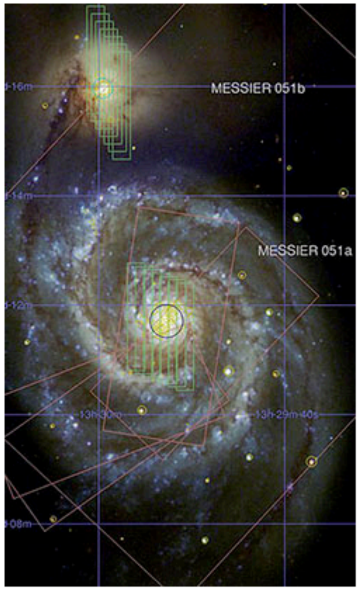
Logout

File Edit View Insert Cell Kernel Widgets Help

Not Trusted | Python 3

Run | Markdown

Building a Mosaic with Montage



Montage is a general toolkit for reprojecting and mosaicking astronomical images and generally you have to marshall the specific data you want to use carefully. But there are a few large-scale uniform surveys that cover a large enough portion of the sky to allow a simple location-based approach.

In this notebook we will choose a region of the sky and dataset to mosaic, retrieve the archive data, reproject and background-correct the images, and finally build an output mosaic. You are free to modify any of the mosaic parameters but beware that as you go larger all of the steps will take longer (possibly **much** longer). If you do this for three different wavelenths, you can put them together in a full-color composite using our Sky Visualization notebook, which produced the image on the right.

As with many notebooks, this was derived from a longer script by breaking the processing up into sequential steps. These steps (cells) have to be run one in sequence. Wait for each cell to finish (watch for the step number in the brackets on the left to stop showing an asterisk) before starting the execution of next cell or run them all as a set.

If you want to just see the code without all the explanation, check out this example.

Setup

The Montage Python package is a mixture of pure Python and Python binary extension code. It can be downloaded using `pip install MontagePy`

No other installations are necessary.

In [2]:

```
# Startup.  The Montage modules are pretty much self-contained
# but this script needs a few extra utilities.

import os
import sys
import shutil

from MontagePy.main    import *
from MontagePy.archive import *

from IPython.display import Image


# These are the parameters defining the mosaic we want to make.

location  = "M 17"
size      = 1.0
dataset   = "2MASS J"
workdir   = "M17"
```

So not much to see so far. We've defined a location on the sky (which can be either an object name (*e.g.* "`Messier 017`") or coordinates. The coordinate parser is pretty flexible; "`3h 29m 53s +47d 11m 43s`" (defaults to the Equatorial J2000 system), "`201.94301 47.45294 Equ B1950`" and "`104.85154 68.56078 Galactic`" all work. We've also defined a size. In this case we are going to use this below to construct a simple North-up gnomonic projection square box on the sky; you are free to define any header you like as Montage supports all standard astronomical projections and coordinate systems.

Working Environment

Before we get to actually building the mosaic, we need to set up our working environment. Given the volume of data possible, the Montage processing is file based and we need to set up some subdirectories to hold bits of it. This will all be under an instance-specific directory specified above ("workdir"). It is best not to use directory names with embedded spaces.

In [3]:

```
# We create and move into subdirectories in this notebook
# but we want to come back to the original startup directory
# whenever we restart the processing.

try:
    home
except:
    home = os.getcwd()

os.chdir(home)

print("Startup folder: " + home)


# Clean out any old copy of the work tree, then remake it
# and the set of the subdirectories we will need.

try:
    shutil.rmtree(workdir)
except:
    print("                Can't delete work tree; probably doesn't exist yet", flush=True)

print("Work directory: " + workdir, flush=True)

os.makedirs(workdir)

os.chdir(workdir)

os.makedirs("raw")
os.makedirs("projected")
os.makedirs("diffs")
os.makedirs("corrected")
```

```
Startup folder: /Users/jcg/MontageDocs
Work directory: M17
```

Retrieving Data from an Archive

Now the first bit of Montage processing. Montage uses standard FITS files throughout and FITS files have all the metadata describing the image (for us that mainly means pixel scale, projection type and center, rotation and so on). To drive the processing we need a "FITS header" specification from the user, which we capture in a header "template" file that looks just like a FITS header though with newlines (FITS headers have fixed 80-character card images with no line

Retrieving Data from an Archive

Now the first bit of Montage processing. Montage uses standard FITS files throughout and FITS files have all the metadata describing the image (for us that mainly means pixel scale, projection type and center, rotation and so on). To drive the processing we need a "FITS header" specification from the user, which we capture in a header "template" file that looks just like a FITS header though with newlines (FITS headers have fixed 80-character card images with no line breaks). The mHdr routine is a utility that creates a simple FITS header template with limited control over all the above (*e.g.* the projection is always gnomonic (TAN)). Other common options are to use a header extracted from some pre-existing FITS file (to create a matching mosaic) or to use mMakeHdr, which fits a bounding box around a set of images (usually the ones you are about to mosaic).

We also use the location and size to retrieve the data we want from a remote archive. Montage provides an image metadata search service (using mSearch — a very fast R-Tree / memory-mapped utility — for most datasets). This service returns URLs for all the images covering the region of interest, which are then downloaded.

There are many other ways to find images. The International Virtual Astronomy Alliance (IVOA) has developed a couple of standards for querying metadata (Simple Image Access: SIAP and Table Access Protocol: TAP) which many data providers support. Our service is focused on a few large uniform datasets (2MASS, DSS, SDSS, WISE). Other datasets require more care. For instance, simply downloading all pointed observations of a specific region for a non-survey instrument will include a wide range of integration times (and therefore noise levels) and the mosaicking should involve user-specified weighting of the images (which Montage supports but does not define).

In [4]:
```
# Create the FITS header for the mosaic.

rtn = mHdr(location, size, size, "region.hdr")

print("mHdr:             " + str(rtn), flush=True)


# Retrieve archive images covering the region then scan
# the images for their coverage metadata.

rtn = mArchiveDownload(dataset, location, size, "raw")

print("mArchiveDownload: " + str(rtn), flush=True)

rtn = mImgtbl("raw", "rimages.tbl")

print("mImgtbl (raw):    " + str(rtn), flush=True)
```

```
mHdr:             {'status': '0', 'count': 16}
mArchiveDownload: {'status': '0', 'count': 49}
mImgtbl (raw):    {'status': '0', 'count': 49, 'badfits': 0, 'badwcs': 0}
```

Reprojecting the Images

In the last step above, we generated a list of all the images (with projection metadata) that had been successfully downloaded. Using this and header template from above, we can now reproject all the images to a common frame.

Montage supplies four different reprojection modules to fit different needs. mProject is completely general and is flux conserving but this results in it being fairly slow. For a subset of projections (specifically where both the source and destination are tangent-plane projections) we can use a custom plane-to-plane algorithm developed by the Spitzer Space Telescope). While mProjectPP only supports a subset of cases, they are extremely common ones. mProjectPP is also flux conserving.

For fast reprojection, we relax the flux conservation requirement. However, even though we call attention to this explicitly in the name of the module:

Reprojecting the Images

In the last step above, we generated a list of all the images (with projection metadata) that had been successfully downloaded. Using this and header template from above, we can now reproject all the images to a common frame.

Montage supplies four different reprojection modules to fit different needs. mProject is completely general and is flux conserving but this results in it being fairly slow. For a subset of projections (specifically where both the source and destination are tangent-plane projections) we can use a custom plane-to-plane algorithm developed by the Spitzer Space Telescope). While mProjectPP only supports a subset of cases, they are extremely common ones. mProjectPP is also flux conserving.

For fast reprojection, we relax the flux conservation requirement. However, even though we call attention to this explicitly in the name of the module: mProjectQL (quick-look), the results are indistinguishable from the other algorithms for all the tests we have run.

The fourth reprojection module, mProjectCube, is specifically for three- and four-dimensional datacubes.

mProjExec is a wrapper around the three main reprojection routines that determines whether mProjectPP or mProject should be used for each image (unless overridden with mProjectQL as here). There is one final twist: FITS headers allow for distortion parameters. While these were introduced to deal with instrumental distortions, we can often use them to mimic an arbitrary projection over a small region with a distorted gnomonic projection. This allows us to use mProjectPP over a wider range of cases and still have flux conservation with increased speed.

In [5]:

```
# Reproject the original images to the  frame of the
# output FITS header we created

rtn = mProjExec("raw", "rimages.tbl", "region.hdr", projdir="projected", quickMode=True)

print("mProjExec:           " + str(rtn), flush=True)

mImgtbl("projected", "pimages.tbl")

print("mImgtbl (projected): " + str(rtn), flush=True)
```

```
mProjExec:           {'status': '0', 'count': 49, 'failed': 0, 'nooverlap': 0}
mImgtbl (projected): {'status': '0', 'count': 49, 'failed': 0, 'nooverlap': 0}
```

Coadding for a Mosaic

Now that we have a set of image all reprojected to a common frame, we can coadd them into a mosaic.

In [6]:

```
# Coadd the projected images without backgound correction.
# This step is just to illustrate the need for background correction
# and can be omitted.

rtn = mAdd("projected", "pimages.tbl", "region.hdr", "uncorrected.fits")

print("mAdd:    " + str(rtn), flush=True)
```

```
mAdd:    {'status': '0', 'time': 1.0}
```

View the Image

FITS files are binary data structures. To see the image we need to render to a JPEG or PNG form. This involves choosing a stretch, color table (if it is a single

View the Image

FITS files are binary data structures. To see the image we need to render to a JPEG or PNG form. This involves choosing a stretch, color table (if it is a single image as here) and so on. Montage provides a general visualization tool (mViewer) which can process a single image or multiple images for full color. It supports overlays of various sorts. One of its most useful features is a custom stretch algorithm which is based on gaussian-transformed histogram equalization, optionally with an extra logarithmic transform for really bright excursions. A large fraction of astronomical images share the general characteristics of having a lot of pixels with something like a gaussian distribution at a low flux level (either background noise of low-level sky structure) coupled with a long histogram tail of very bright point-like sources. If we apply our algorithm to this, stretching from the -2 or -1 "sigma" value of the low-level distribution to the image brightness maximum we usually get a good balance of seeing the low-level structure while still seeing the structure and brightness variations of the bright sources.

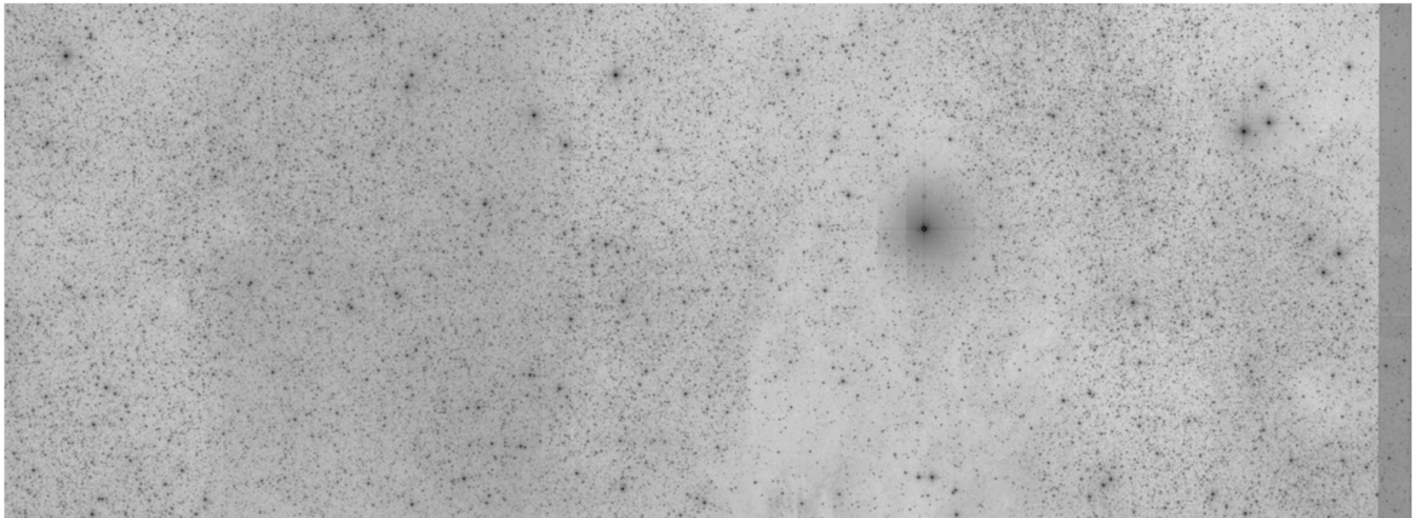
mViewer specifications can become quite lengthy so the module provides three entry mechanisms. The most terse (used here) is a "parameter string" based on the command-line arguments of the original stand-alone C program. For more complicated descriptions the user can define a JSON string or JSON file. See the Sky Visualization notebook example.

We use the built-in IPython.display utility to show the resultant image, which shrinks it to fit. There are several other tools you can use instead.

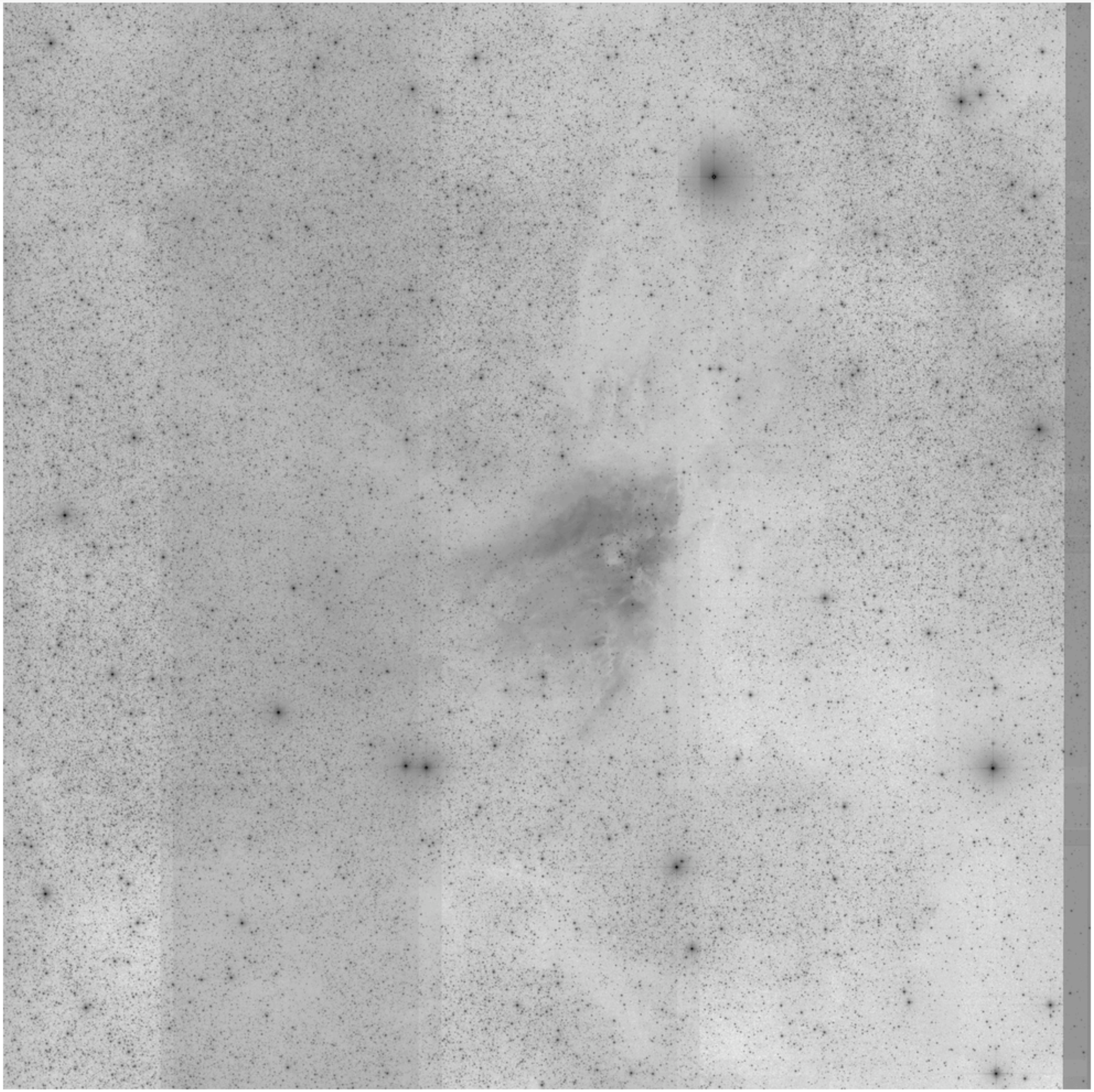
In [7]:
```
# Make a PNG rendering of the data and display it.

rtn = mViewer("-ct 1 -gray uncorrected.fits -2s max gaussian-log -out uncorrected.png", "", mode=2)

print("mViewer: " + str(rtn), flush=True)

Image(filename='uncorrected.png')
```

```
mViewer: {'status': '0', 'type': b'grayscale', 'nx': 3601, 'ny': 3600, 'grayminval': 140.15615325038922, 'grayminperc
ent': 0.0, 'grayminsigma': -2.0, 'graymaxval': 10630.5380859375, 'graymaxpercent': 100.0, 'graymaxsigma': 1709.267821
0239994, 'blueminval': 0.0, 'blueminpercent': 0.0, 'blueminsigma': 0.0, 'bluemaxval': 0.0, 'bluemaxpercent': 0.0, 'bl
uemaxsigma': 0.0, 'greenminval': 0.0, 'greenminpercent': 0.0, 'greenminsigma': 0.0, 'greenmaxval': 0.0, 'greenmaxperc
ent': 0.0, 'greenmaxsigma': 0.0, 'redminval': 0.0, 'redminpercent': 0.0, 'redminsigma': 0.0, 'redmaxval': 0.0, 'redma
xpercent': 0.0, 'redmaxsigma': 0.0, 'graydatamin': 142.23333740234375, 'graydatamax': 10630.5380859375, 'bdatamin': 0
.0, 'bdatamax': 0.0, 'gdatamin': 0.0, 'gdatamax': 0.0, 'rdatamin': 0.0, 'rdatamax': 0.0, 'flipX': 0, 'flipY': 1, 'col
ortable': 1, 'bunit': b''}
```

Out[7]:



Out[7]:



Background Matching

We can do better. In the above image (at least in the original example, it may vary if you've chosen another dataset) there are vertical stripes. Even though the images were accurately flux-calibrated the background levels in the individual image varied due to real changes in the brightness of the sky (2MASS data was taken from the ground, so the atmosphere was a variable).

This is a common problem; differential photometry is easier than absolute. So Montage provides tools for determining what is essentially a compromise background: Not flattened (since in the above mosaic there is real structure throughout the image) and not modelled (there might be a model you can develop for the Galactic structure above but it wouldn't do that good a job of cleaning up the mosaic).

Rather, we ask what is the set of minimum adjustments we can make to the individual image backgrounds to bring them all in-line with each others. Often, this is just a constant offset; at most it includes slight slopes. Anything more and we are starting to fit the sky structure rather than the background differences.

The first steps is determining the corrections is to analyze the overlap areas between adjacent images. We determine from the image metadata (which again includes sky coverage) where there are overlaps. Then for each pairwise overlap, we compute the image difference. There is an explicit assumption here that a such a pair the sources and other real-sky structure match (including flux scales) so the difference should have nothing in it but background differences. We then fit each difference with a plane (ignoring large excursions just to be safe).

Finally, given this set of difference fits, we determine iteratively a global mimimum difference which results in a set of corrections to each image.

In [8]:

```
# Determine the overlaps between images (for background modeling).

rtn = mOverlaps("pimages.tbl", "diffs.tbl")

print("mOverlaps:    " + str(rtn), flush=True)


# Generate difference images and fit them.

rtn = mDiffFitExec("projected", "diffs.tbl", "region.hdr", "diffs", "fits.tbl")

print("mDiffFitExec: " + str(rtn), flush=True)


# Model the background corrections.

rtn = mBgModel("pimages.tbl", "fits.tbl", "corrections.tbl")

print("mBgModel:     " + str(rtn), flush=True)
```

```
mOverlaps:    {'status': '0', 'count': 128}
mDiffFitExec: {'status': '0', 'count': 128, 'diff_failed': 0, 'fit_failed': 0, 'warning': 0}
mBgModel:     {'status': '0'}
```

Background Correcting and Re-Mosaicking

Now all we have to do is apply the background corrections to the individual images and regenerate the mosaic. While we don't attempt to maintain the global total flux (this would be meaningless in any case given the source of the offsets), in general our final mosaic is close to this level.

For those cases where the background should truly be flat (extragalactic fields with no foreground we want to keep) Montage also provides simple "flattening" tools.

Background Correcting and Re-Mosaicking

Now all we have to do is apply the background corrections to the individual images and regenerate the mosaic. While we don't attempt to maintain the global total flux (this would be meaningless in any case given the source of the offsets), in general our final mosaic is close to this level.

For those cases where the background should truly be flat (extragalactic fields with no foreground we want to keep) Montage also provides simple "flattening" tools.

In [9]:
```
# Background correct the projected images.

rtn = mBgExec("projected", "pimages.tbl", "corrections.tbl", "corrected")

print("mBgExec:             " + str(rtn), flush=True)

rtn = mImgtbl("corrected", "cimages.tbl")

print("mImgtbl (corrected): " + str(rtn), flush=True)


# Coadd the background-corrected, projected images.

rtn = mAdd("corrected", "cimages.tbl", "region.hdr", "mosaic.fits")

print("mAdd:                " + str(rtn), flush=True)
```

```
mBgExec:             {'status': '0', 'count': 49, 'nocorrection': 0, 'failed': 0}
mImgtbl (corrected): {'status': '0', 'count': 49, 'badfits': 0, 'badwcs': 0}
mAdd:                {'status': '0', 'time': 1.0}
```

Final Image

Now when we regenerate and display a PNG for the mosaic, it has no artifacts and all of the low-level structure is preserved.

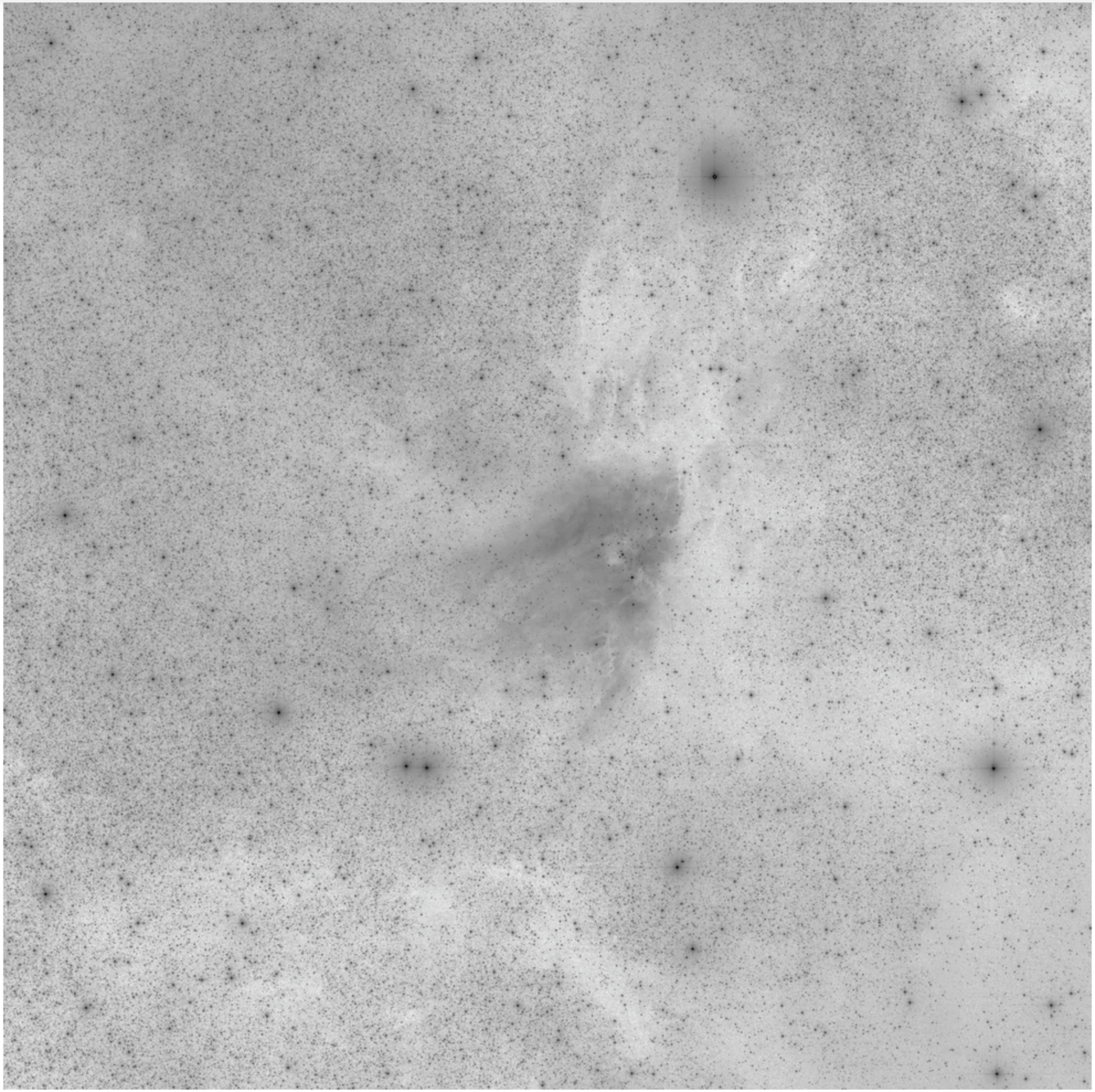
In [12]:
```
# Make a PNG rendering of the data and display it.

rtn = mViewer("-ct 1 -gray mosaic.fits -2s max gaussian-log -out mosaic.png", "", mode=2)

print("mViewer: " + str(rtn), flush=True)
```

```
mViewer: {'status': '0', 'type': b'grayscale', 'nx': 3601, 'ny': 3600, 'grayminval': 149.5791825248529, 'grayminperce
nt': 0.0, 'grayminsigma': -2.0, 'graymaxval': 10639.0999189825, 'graymaxpercent': 100.0, 'graymaxsigma': 2240.4651757
359084, 'blueminval': 0.0, 'blueminpercent': 0.0, 'blueminsigma': 0.0, 'bluemaxval': 0.0, 'bluemaxpercent': 0.0, 'blu
emaxsigma': 0.0, 'greenminval': 0.0, 'greenminpercent': 0.0, 'greenminsigma': 0.0, 'greenmaxval': 0.0, 'greenmaxperce
nt': 0.0, 'greenmaxsigma': 0.0, 'redminval': 0.0, 'redminpercent': 0.0, 'redminsigma': 0.0, 'redmaxval': 0.0, 'redmax
percent': 0.0, 'redmaxsigma': 0.0, 'graydatamin': 150.8148507701094, 'graydatamax': 10639.0999189825, 'bdatamin': 0.0
, 'bdatamax': 0.0, 'gdatamin': 0.0, 'gdatamax': 0.0, 'rdatamin': 0.0, 'rdatamax': 0.0, 'flipX': 0, 'flipY': 1, 'color
table': 1, 'bunit': b''}
```

In [11]:
```
Image(filename='mosaic.png')
```

Out[11]:

Out[11]:



The Montage Image Viewer

jupyter mViewer Last Checkpoint: 12/07/2017 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

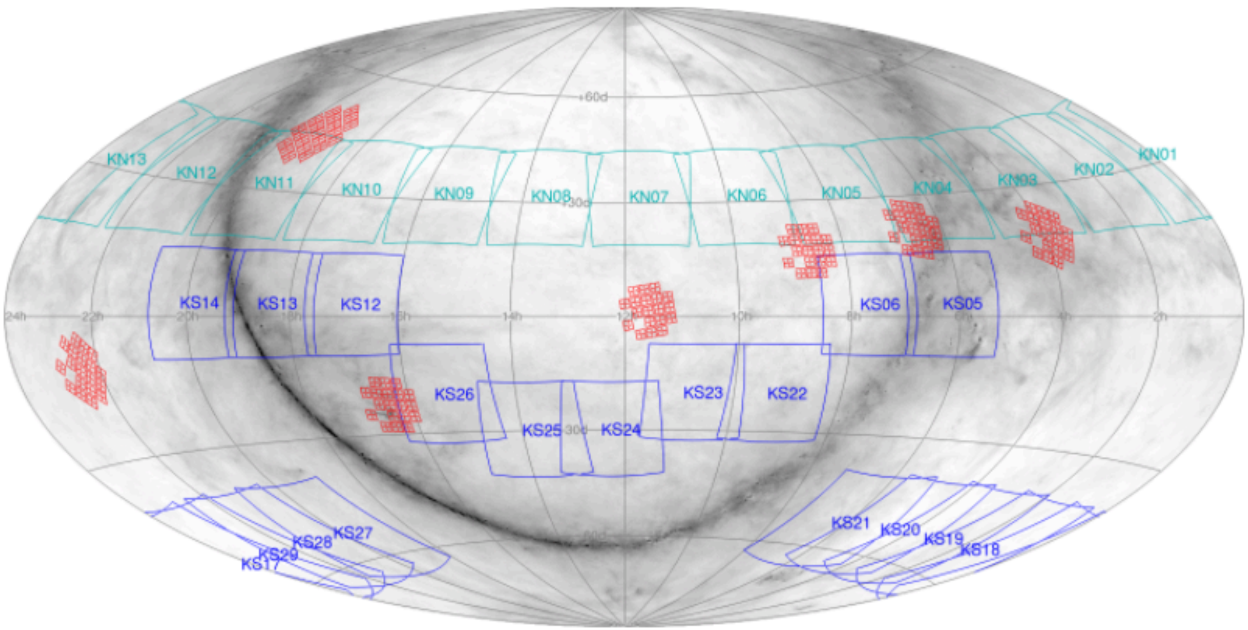
Run Markdown

mViewer Sky Visualization

The Montage toolkit is primarily involved in reprojecting, background matching, and mosaicking astronomical images but it also has fairly sophisticated tools for image display (PNG/JPEG generation). This notebook entry shows the basics of using mViewer to create a full-color image from three wavelength bands that have already been reprojected and mosaicked.

If you want to just see the code without all the explanation, check out this example.

In this example, we will focus on the image data but mViewer can also build visualizations where the image data is secondary or invisible (see the coverage map tutorial for more information and the data:



Setup

The Montage Python package is a mixture of pure Python and Python binary extension code. It can be downloaded using `pip install MontagePy`

No other installations are necessary.

In [1]:
```
from MontagePy.main import mViewer

help(mViewer)
```

```
Help on built-in function mViewer in module MontagePy.main:

mViewer(...)
    mViewer generates a JPEG image file with overlays from a FITS file (or a set of three F
ITS files in color).

    Parameters
    ----------
    cmdstr : str
        The command string (arguments or JSON).
    outFile : str
        Output PNG/JPEG.
    mode : int, optional
        Type of the command string: Type of the command string: 0 for JSON file, 1 for JSON
, 2 for 'command' string.
    outFmt : str, optional
        'png' or 'jpeg'.
    fontFile : str, optional
        Font file for labeling (overrides default)
```

The Montage toolkit contains dozens of utilities. Two that are often used along with mViewer are mShrink to rescale the input images and mSubimage to cut out sections. All of Montage is available through the MontagePy.main module.

JSON control

The number of arguments to mViewer is a little large for a standard keyword list so we will be using a JSON rendering of the parameters as the driver. This could be in the form of a file but we will use a JSON string. This will require us to set the '`mode`' parameter for mViewer to 1 when we use it below.

Here we create the JSON as a simple string (note the triple quotes which let us make this a multi-line with embedded quotes, *etc.*) Since this is an active notebook, you can modify this as you like to adjust the final image.

In [4]:
```
imgjson = """
{
   "image_file":"viewer.png",
   "image_type":"png",
   "true_color":1.50,
   "font_scale":1.1,

   "blue_file":
   {
      "fits_file":"fits/SDSS_u.fits",
      "stretch_min":"-0.1s",
      "stretch_max":"max",
      "stretch_mode":"gaussian-log"
   },
```

```
In [4]: imgjson = """
{
   "image_file":"viewer.png",
   "image_type":"png",
   "true_color":1.50,
   "font_scale":1.1,

   "blue_file":
   {
      "fits_file":"fits/SDSS_u.fits",
      "stretch_min":"-0.1s",
      "stretch_max":"max",
      "stretch_mode":"gaussian-log"
   },

   "green_file":
   {
      "fits_file":"fits/SDSS_g.fits",
      "stretch_min":"-0.1s",
      "stretch_max":"max",
      "stretch_mode":"gaussian-log"
   },

   "red_file":
   {
      "fits_file":"fits/SDSS_r.fits",
      "stretch_min":"-0.1s",
      "stretch_max":"max",
      "stretch_mode":"gaussian-log"
   },

   "overlays":
   [
      {
         "type":"grid",
         "coord_sys":"Equ J2000",
         "color":"8080ff"
      },
      {
         "type":"imginfo",
         "data_file":"tbl/irspeakup.tbl",
         "coord_sys":"Equ J2000",
         "color":"ff9090"
      },
      {
         "type":"catalog",
         "data_file":"tbl/fp_2mass.tbl",
         "data_column":"j_m",
         "data_ref":16,
         "data_type":"mag",
         "symbol":"circle",
         "sym_size":1.0,
         "coord_sys":"Equ J2000",
         "color":"ffff00"
      }
   ]
}
"""
```

Running mViewer

Along with this notebook we have a set of data files in data subdirectories (as referenced in the above JSON). So all we need to do is run mViewer, handing it the JSON file reference and specifying where we want the output PNG file. All the Montage tools output JSON return structures; mViewer's gives various statisticsfrom the processing (for instance, 'gdatamax' is the maximum data value for the 'green' image ('SDSS_g.fits').

```
In [6]: mViewer(imgjson, "test.png", mode=1)
```


```
Out[6]: {'bdatamax': 18833.51097929482,
 'bdatamin': 1650.1333317872777,
 'bluemaxpercent': 100.0,
 'bluemaxsigma': 7293.270993655378,
 'bluemaxval': 18833.51097929482,
 'blueminpercent': 45.51304816901208,
 'blueminsigma': -0.10000000000002897,
 'blueminval': 1660.948886289596,
 'bunit': b'',
 'colortable': 0,
 'flipX': 0,
 'flipY': 1,
 'gdatamax': 26276.085778495264,
 'gdatamin': 608.5073366952034,
 'graydatamax': 0.0,
 'graydatamin': 0.0,
 'graymaxpercent': 0.0,
 'graymaxsigma': 0.0,
 'graymaxval': 0.0,
 'grayminpercent': 0.0,
 'grayminsigma': 0.0,
 'grayminval': 0.0,
 'greenmaxpercent': 100.0,
 'greenmaxsigma': 14790.776567073211,
 'greenmaxval': 26276.085778495264,
 'greenminpercent': 43.049856485589686,
 'greenminsigma': -0.1000000000000262,
 'greenminval': 612.1976698225872,
 'nx': 1200,
 'ny': 1200,
 'rdatamax': 20567.63987893109,
 'rdatamin': 890.1481250441316,
 'redmaxpercent': 100.0,
 'redmaxsigma': 6285.599891788824,
 'redmaxval': 20567.63987893109,
 'redminpercent': 42.62543763181061,
 'redminsigma': -0.10000000000000908,
 'redminval': 896.9336930051037,
 'status': '0',
 'type': b'color'}
```

Displaying the PNG

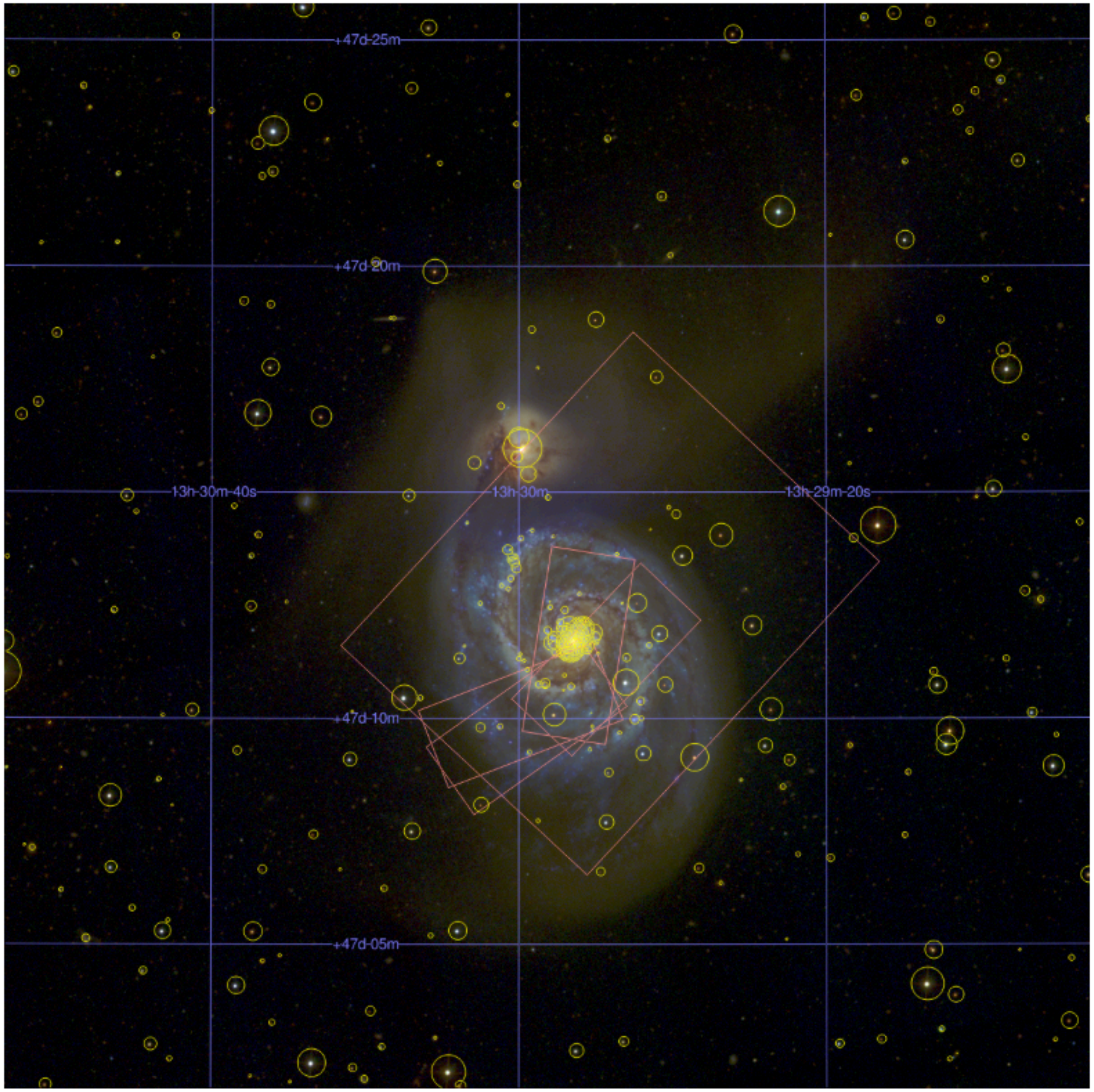
Below, we use a Jupyter internal function to display the image. The current Jupyter implementation has an overly-strict limit on I/O and our image exceeds this. If you see the same problem, you can overcome it by restarting the system with

```
jupyter notebook --NotebookApp.iopub_data_rate_limit=10000000000
```

In [8]:

```
from IPython.display import Image

Image(filename='test.png', unconfined=False)
```

Out[8]:



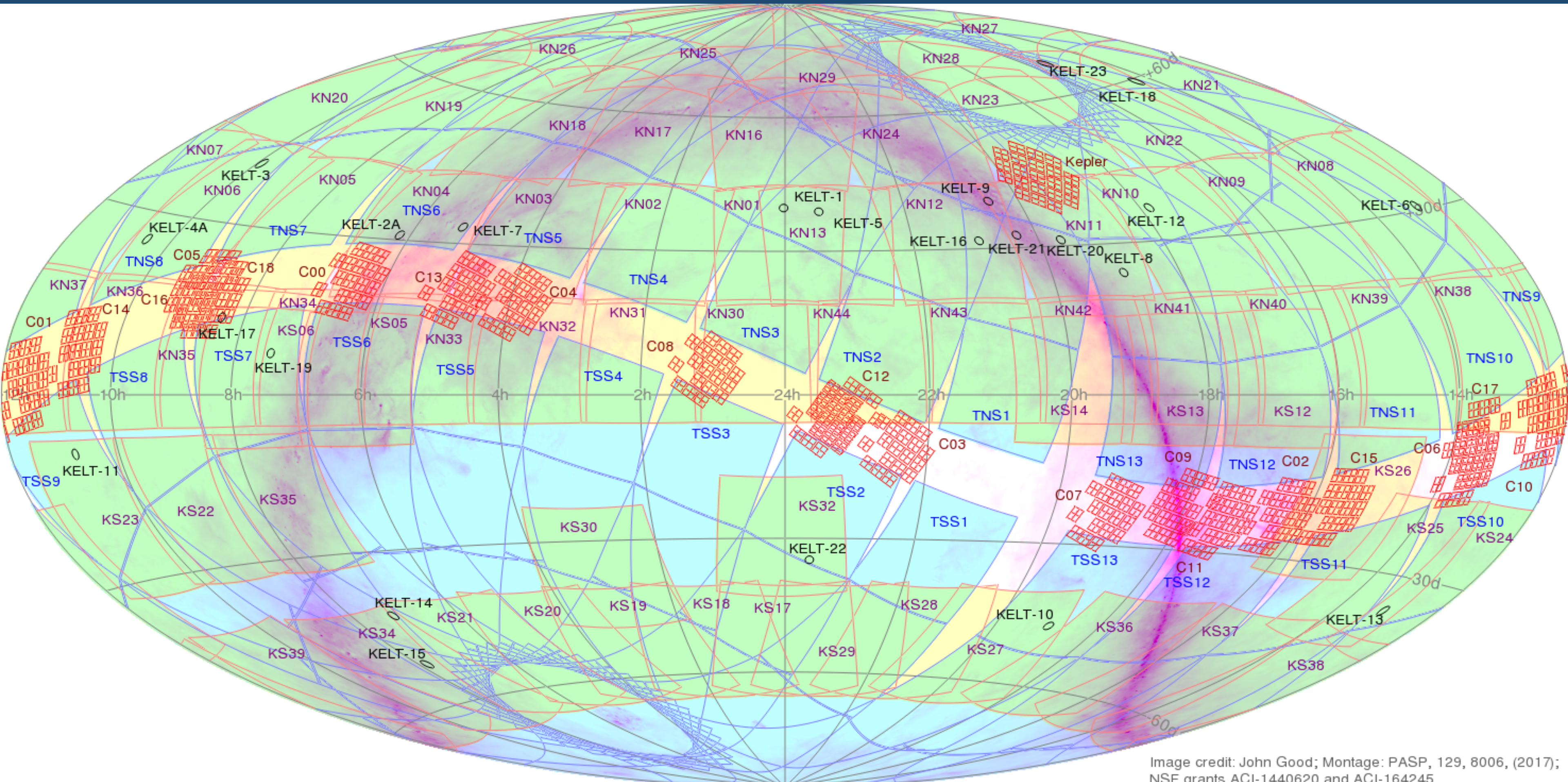


Image credit: John Good; Montage: PASP, 129, 8006, (2017);
NSF grants ACI-1440620 and ACI-164245

An Example Interactive Viewer

mView: Interactive mViewer Wrapper

The Montage mViewer utility can be used to created fairly complex sky maps (grayscale or three-color images, overlays of coordinate grids, catalogs, image metadata, *etc.*). Often this is just done directly and the resultant PNG viewed directly, placed on a web page, or incorporated into documentation.

Sometimes, however, we would like to interact more directly with the data, updating parameters and viewing the result through a GUI. For this, we have wrapped mViewer (and some other Montage modules) with an interactive interface written using the PyQt widget library. This interface, mView, can be run directly or through a Jupyter notebook, as shown here.

For more detail on using mViewer directly, check out this example.

Initialization

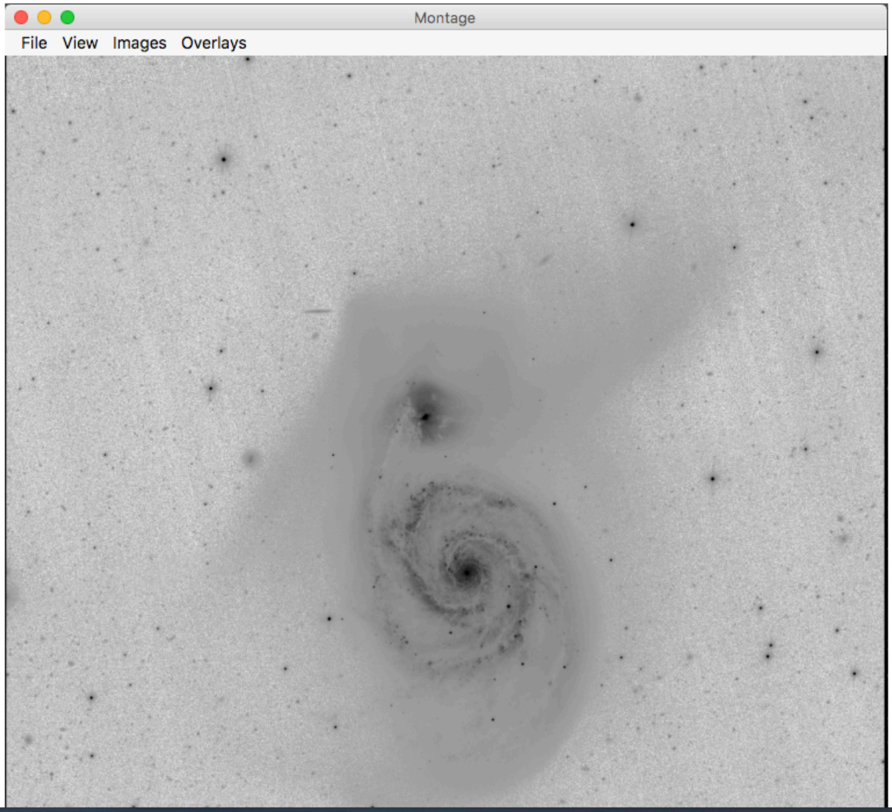
mView can be initialized in a number of ways but all of these result in generating the JSON structure described in the mViewer documentation. Here we will use the simplest form: a single FITS image name.

In [ ]:

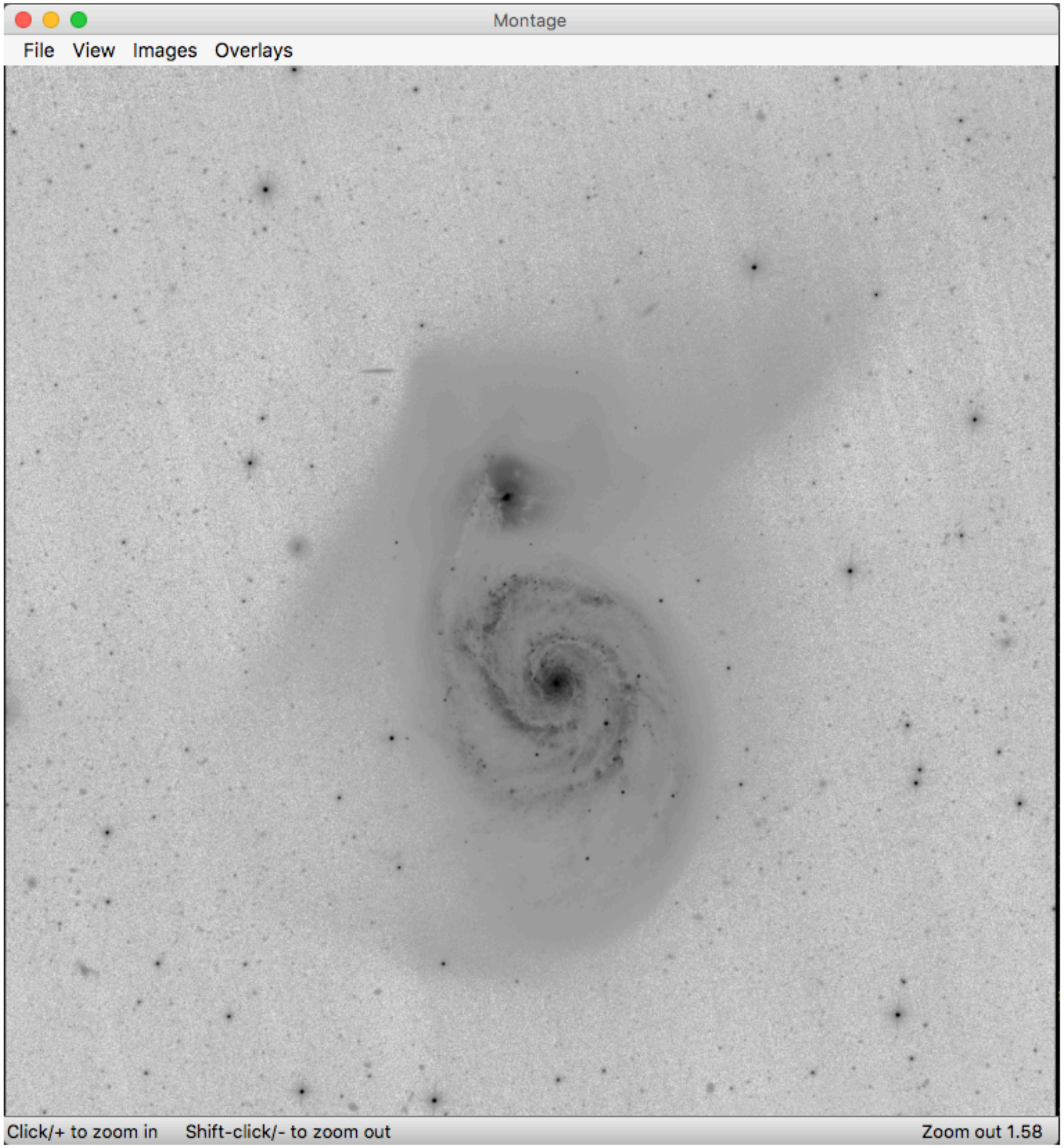
```
import mView

mView.main(["","fits/SDSS_g.fits"])
```

```
Close mView window to continue.
```

Running the above will cause a second (non-browser) window to pop up, showing the data properly scaled to grayscale and zoomed to fit the display window. Since this display doesn't render in the notebook (and therefore won't be visible when not running) we show a screen dump of it below:

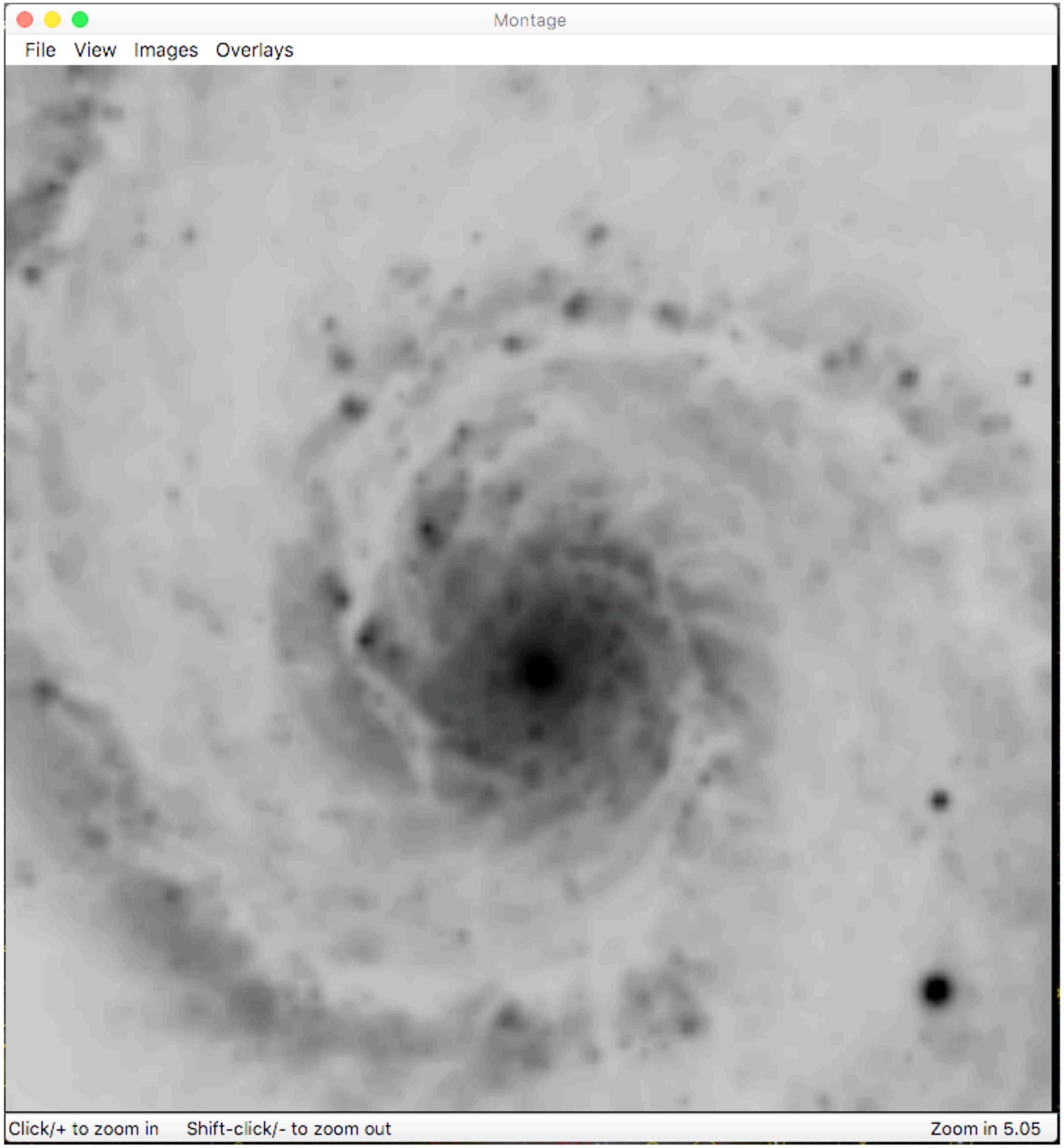


Running the above will cause a second (non-browser) window to pop up, showing the data properly scaled to grayscale and zoomed to fit the display window. Since this display doesn't render in the notebook (and therefore won't be visible when not running) we show a screen dump of it below:



Zooming and Panning

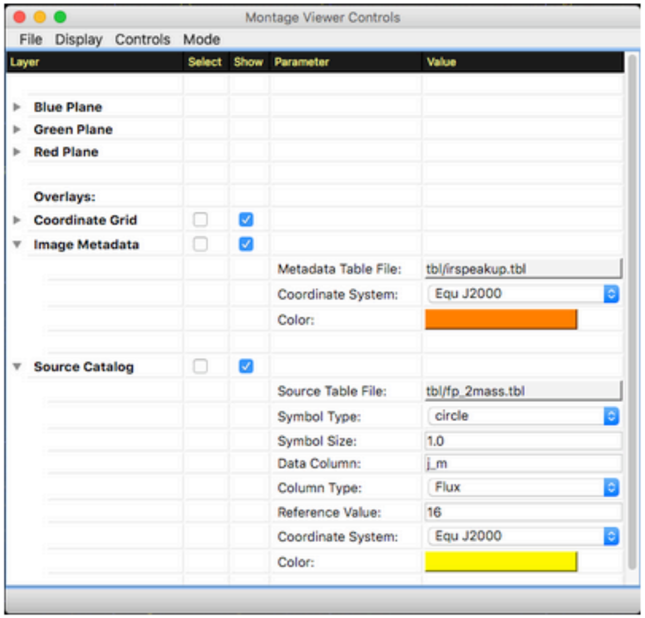
The GUI provides all the standard capabilities for manipulating the display (zooming in and out, fitting the image to the window and resizing the window, zooming to 'actual pixel' size, dragging the window to recenter).



Display Controls

[This section is still a work in progress.] A fully-defined display can require dozens of parameters. To start with we have the image or images, with color stretching information. The there is a list of overlays and the parameters for each one. A source catalog overlay is based on a table of data and we may have to specify the table coordinate system, which column to use for scaling the points, whether that column is in flux units or magnitudes, a reference symbol size and the corresponding data value, the symbol shape and its color.

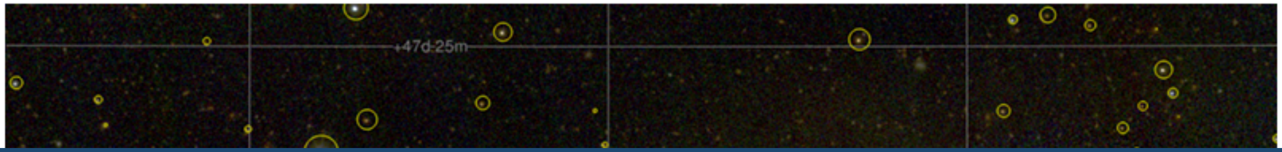
This can be set up by modifying the input JSON with an editor but for the GUI we provide an editor. The current state of that widget is shown here:

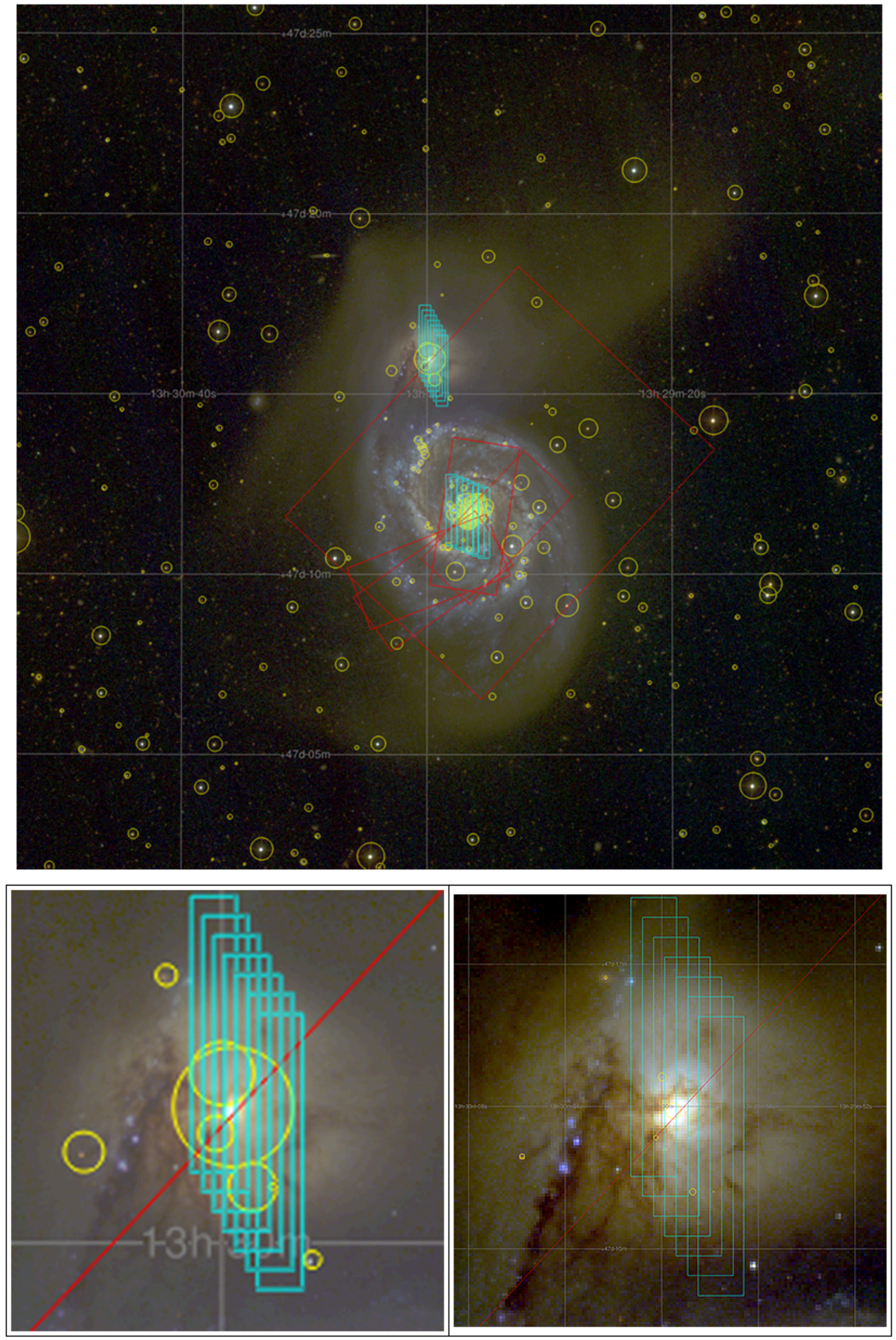


We have just started working on this so it should evolve rapidly. For instance, only one of the color fields has been linked to a color picker and the coordinate system field may be a pulldown list.

Cropping and Redrawing

As you zoom in on a given location, all the overlay graphics (which the this point has been zooming with the data) can become blurry and pixelated. Also, the appropriate color stretch for the image as a whole may not be optimum. Therefore, the GUI allows the user to crop the original data to the current display region, resize the pixels, stretch to the new data range, and redraw the overlays.




+47d 25m
+47d 20m
13h 30m 40s
13h
13h 29m 20s
+47d 10m
+47d 05m
13h