GIPPY Documentation

Release 1.0.0

Matthew Hanson

Jun 01, 2017

Contents

Gippy is a Python library for image processing of geospatial raster data. Gippy includes a C++ library, libgip, that encapsulates the functionality of GDAL and CImg, with Python bindings generated with SWIG. Gippy handles issues common to geospatial data, such as handling of nodata values and chunking up of very large images by saving chains of functions and processing image in pieces when finally read. It is reasonably lightweight, requiring only GDAL system libraries and a reasonably modern C++ compiler.

Gippy uses the GDAL C++ API, but provides a simplified object-oriented and Pythonic interface to using data. Gippy provides another level of abstraction where an image is a collection of raster bands and contains additional metadata important for processing. A GeoImage can contain bands from different files, and images can be opened from multiple files, combined, raster bands removed, reordered, then saved as a new file. In Gippy, the focus is on manipulation and processing of a collection of raster bands for a given region of interest.

CImg is a C++ template library for image processing. As a template library, it's source code is included with Gippy and requires no additional installation. CImg allows images to be used as objects, with a variety of member mathematical functions, very much similar to NumPy. Gippy encapsulates this functionality, allowing raster bands to be used as objects, where they may be multiplied by a constant, scaled, or have some other mathematical operator applied. All of these functions are chained together and then, upon a read request, CImg is used on each chunk in turn and the processing chain applied.

CHAPTER 1

Features

Band number abstraction

When a GeoImage is opened or created bandnames can be assigned using the bandnames keyword to open, or setting them after creation. Bands can be referenced via band index or the band name and can be iterated over as a collection. This allows algorithms to written to target specific types of bands (e.g., red, nir, lwir). No more having to worry about what band numbers are what for your data, just write a custom open function for your data and set the band names when you open it, and use band names in the rest of your code.

Nodata

While typically not found in traditional image processing applications, "no data" values are extremely common in geospatial data. Projected rasters will contain nodata values outside the data boundary, but included within the bounding box in projected space. Additionally, nodata values may be used to mask out invalid pixels (such as determine from a quality band or a cloud detection algorithm), or may be due to data missing due to sensor issues (e.g., strips of nodata in Landsat7 due to the broken scan line corrector mirrors). Gippy propogates any nodata value through the chain and creates output where nodata pixels stay as nodata pixels (even if the actual nodata value is changed when writing an output file).

Process chains

Gippy supports the chaining together of processing operations (e.g., +, -, log, abs) on raster bands. These operations are stored as C++ lambda functions in the GeoRaster object and are applied each time the data is read from disk (if user requested or due to a save).

Image Chunking

Gippy is most often used to procss input data to create a new data file, such as a GeoTIFF. When using a Gippy algorithm or save() function, the image is automatically chunked up into pieces no larger than a default chunk size and the processing applied in pieces. Chunks can be used within Python code as well, so piecewise processing can be done in custom algorithms.

Algorithms

Gippy includes an algorithm module, which is a collection of functions that operate on GeoImage objects. There are currently only a handful of functions, but this will be expanded upon in future versions. Gippy algorithms take in one or more images, parameters unique to the algorithm, and an output filename and output file options. Currently the algorithms module includes indices() which can calculate a variety of indices (e.g., NDVI, NDWI, LSWI, SATVI) in one pass, acca() and fmask() cloud detection algorithms for Landsat7, cookie_cutter() for mosaicking together scenes, linear_transform() for applying basis vectors to spectral data, pansharpening, and a rxd(), a multispectral anomaly detection algorithm.

CHAPTER 2

Table of Contents

Installation

Gippy is a C++ library with Python bindings. The C++ portion, libgip.so and C++ wrappers wrappers around it, are built as extensions using Python setuptools, so do not require seperate installation. Gippy has been designed to use as few system dependencies as possible, however there are a few.s

On Ubuntu (14.04):

```
$ sudo apt-get install libgdal-dev python-setuptools g++ python-dev
# sudo easy_install pip             # if pip not already installed
$ sudo pip install numpy            # pre-install numpy
```

On OS X (using brew):

```
$ brew install gdal
```

With the dependencies met, gippy can be installed via pip from it's repository on PyPi. If installing to a virtual environment, activate the environment first. If installing system-wide pip will need to be run as sudo.

$ pip install gippy –pre

To install a beta version use the –pre switch. Without –pre, pip will install the last release version, which is currently 0.3.5.

Development Installation

For development purposes, the swig wrappers must be regenerated anytime the C++ code is modified, and thus the swig package (currently using swig2.0) must be installed.

On Ubuntu:

```
$ sudo apt-get install swig
```

On OS X:

```
$ brew install swig
```

Then install gippy as a development installation by cloing the repository. Links will be installed in the Python packages directory that point to the directory where gippy resides.

```
$ git clone http://github.com/gipit/gippy.git
$ cd gippy
$ pip install -e .
```

Docs

To generate docs:

```
$ cd docs
$ make html
```

Open *docs/_build/html/index.html* in the browser

Testing

Gippy testing is done on the Python siode sing the nosetest testing framework and the sat-testdata repository for test imagery, which is installed as a requirement. Run the tests from the test directory.

```
$ cd test
$ nosetests
```

Quickstart

Gippy includes 3 Python modules: gippy core, an algorithms module, and a small test module that has some convenience functions for testing.

- gippy: The main library that contains the GeoImage and GeoRaster classes.
- gippy.algorithms: Functions that operate on GeoImage objects.
- gippy.test: Utility files for testing or examples. Currently just contains the get_test_image() function.

```
import gippy
import gippy.algorithms
import gippy.test
```

Gippy uses several global options, each one accessible with gippy.Options.[*name*]() and each one has a set_[*name*](newvalue) function.

- *defaultformat* - the [GDAL format code](http://www.gdal.org/formats_list.html) used for new files if not otherwise specified.
- *verbose* - the verbosity level for any messages printed, from 0 (nothing) to 5 (full debug).
- *chunksize* - the default chunk size in MB
- *cores* - the number of cores to use for those parts of Gippy that utilize multi-processing. For now the only thing that uses it is warping operations.

The two main classes in GIPPY are GeoImage and GeoRaster. A GeoRaster is a single raster band, and a GeoImage is a collection of GeoRaster bands that cover the same spatial footprint and have the same resolution, although could be different datatypes.

Working with images

A GeoImage in Gippy is the main class used and is a collection of raster bands covering the same footprint and spatial resolution. The raster bands are most often spectral, but could be a time series. The individual raster bands could be from different files, but all are contained in a GeoImage.

GeoImage has factory functions that are used to open or create new images: GeoImage.open(), GeoImage.create(), and GeoImage.create_from().

Open existing image

A GeoImage can be opened from a single file or multiple files, and bands can be added with add_band() or removed by using select() to get desired bands.

```
from gippy import GeoImage

# open an existing RGB image with a nodata value of 0
geoimg = GeoImage.open(filename, bandnames=(['red', 'green', 'blue']), nodata=0)

# Open up multiple files as a single image
geoimg = GeoImage.open(filenames=[filename1, filename2, filename3],
                       bandnames=(['red', 'green', 'blue']), nodata=0)

# add a band from another image
geoimg.add_band(geoimg2[0])

# get a GeoImage with just the red and nir bands
rn_image = geoimg.select(['red', 'nir'])
```

Create new images

To create a new image use the create() or create_from functions. When creating from scratch a geolocated image will need to be provided a bounding box and projection (default projection is EPSG:4326). Not all keywords are required. A newly created imge will default to a dataypt of byte, and a format of the defaultformat global option.

Values such as bandnames and nodata can be set with the set_[*name*] functions.

```
# Create new int16 image
bounding_box = numpy.array([xmin, ymin, width, height])
geoimg = GeoImage.create(filename, xsz=1000, ysz=1000, nb=3, proj='EPSG:3857',
                         bbox=bounding_box, dtype="int16", format="GTiff")
geoimg.set_bandnames(['red', 'green', 'blue'])

# Create new image with same properties (size, metadata, SRS) as an existing GeoImage
geoimg2 = GeoImage.create_from(geoimg, filename)

# Create new image with same footprint as geoimg, but different datatype and 4 bands
geoimg2 = GeoImage.create_from(gimage, filename, nb=4, dtype='byte')
```

Temporary files can be created by not providng a filename (or giving filename as an empty string), or by providing the *temp* keyword as True. Temporary files are created with a random filename in a temporary folder depending on the

OS (on Linux it is in /tmp), and are automatically deleted when the last reference to it is gone. As an example, see the code below that tests this feature:

```
geoimg = GeoImage.create(filename, xsz=1000, ysz=1000, nb=5, temp=True)
assertTrue(os.path.exists(filename))
# keep a band
band = geoimg[1]
geoimg = None
# band still references file
assertTrue(os.path.exists(filename))
band = None
# file should now have been deleted
assertFalse(os.path.exists(filename))
```

Band names & Colors

The band names assigned when opening an image or creating a new image can be used to reference that raster band. Bands can be referenced via band index or the band name and can be iterated over as a collection. This allows algorithms to written to target specific types of bands (e.g., red, nir, lwir). The code below illustrates using band names instead of indexes and that band 0 and the red band are the same band.

```
from gippy import GeoImage

geoimg = GeoImage.open(filename, bandnames=(['red', 'green', 'blue']))

# add band from another image
geoimg.add_band( geoimg2['lwir'])

arr1 = geoimg[0].read()
arr2 = geoimg['red'].read()

print(geoimg.nbands())

> 4

print(numpy.array_equal(arr1, arr2))

> True
```

Many algorithms can be applied to different sensors because they use standard band colors. For instance NDVI uses red and near-infrared bands and can be calculated regardless of the sensor specs as long as they have been calibrated into reflectance units. As such, the standard names below should be used for bands, where appropriate.

Band Name	Band Range	Landsat 5	Landsat 7	Landsat 8	Sentinel 2	MODIS
Coastal	0.40 - 0.45			1	1	
Blue	0.45 - 0.5	1	1	2	2	3
Green	0.5 - 0.6	2	2	3	3	4
Red	0.6 - 0.7	3	3	4	4	1
Pan	0.5 - 0.7		8	8		
NIR	0.77 - 1.00	4	4	5	8	2
Cirrus	1.35 - 1.40			9	10	26
SWIR1	1.55 - 1.75	5	5	6	11	6
SWIR2	2.1 - 2.3	7	7	7	12	7
LWIR	10.5 - 12.5	6	8			
LWIR1	10.5 - 11.5			10		31
LWIR2	11.5 - 12.5			11		32

Nodata values

Gippy propogates nodata value through any processing done so that nodata pixels stay as nodata pixels (even if the actual nodata value is changed when writing an output file). Every datatype has a default value used for nodata, but it can be set for each raster band, or the entire image. Normally this is done when creating an image or explicitly setting with the one of the set_(*name*) commands.

```
geoimg = GeoImage.open(filename, bandnames=['red', 'green', 'blue'], nodata=255)
# or
# geoimg.set_nodata(255)
# or
# geoimg['red'].set_nodata(255)
```

Gain and offset

An image gain and offset is a linear function that is applied to the data. This may be to convert digital counts to radiance or other units, or to store floating point data in the -1.0 to 1.0 range as Int16 by using a gain of 0.0001 (this takes up half the disk space and is common for storing indices). While these operations are basically the same as multiplying a band by a constant and adding another:

```
toa = geoimg['red'] * gain + offset
```

instead, the gain and offset can be set explicitly, and are applied first before any other processing:

```
geoimg['red'].set_gain(gain)
geoimg['red'].set_offset(offset)
```

The advantage of using the gain() and offset() functions are that they use the GDAL gain and offset functions, thereby setting them on the data source (i.e., GeoTiff) and able to be used by other programs such as QGIS which will apply the gain and offset automatically.

GeoImage and GeoRaster functions

GeoImage and GeoRaster both inherit from a GeoResource class, and thus share many common characteristics, such as both being aware of what their projection and affine transformation are, and containing other similar functions: xsize(), ysize(), filename(), extent(), meta(). There are two types of GeoImage and GeoRaster functions, those that return a new object of that type, and those that operate directly on itself. For instance.

```
rgbimg = geoimg.select(['red', 'green', 'blue'])
```

returns a new image, rgbimg, which is a GeoImage that contains the red, green, and blue bands, in that order. However geoimg still contains all the original bands, in whatever order they were in.

On the other hand, some functions, specifically functions starting with "**set_**", "**add_**", and "**clear_**" all operate on the calling object itself, but they also return a reference to itself, so that functions can be chained together.

```
# this sets nodata on all bands in geoimg
geoimg.set_nodata(255)

# this sets nodata and adds a metadata item to the first band
geoimg.set_nodata(255)[0].add_meta("key", "value")
```

Saving images

Images can be saved as new files using the save() function. The resulting file will have all the same metadata and number of bands, but with all processing applied. The datatype of the new image will be the same as the old one unless the dtype keyword is provided. Note that providing the dtype keyword does not scale the values however, it is up to the user to scale values to the desired range to match the output file created. Use the GeoImage.autoscale() function to automatically scale all bands, or use the GeoRaster.scale() function on each band to specify the input and output ranges.

```
# scale image and save to new file
geoimg.autoscale(1, 255).save(filename, dtype='byte')
```

Images can also be warped to a new projection, which is saved as a new file. The provided proj string can be anything recognized by GDAL: an EPSG code, a proj string, or even a filename containing WKT of the projection. Resolution is in the same units as the new projection and must be provided (they default to 1.0 which will likely not be what is wanted).

If a geometric feature (see Working with vectors) is provided, the resulting file will be clipped to that feature or, if crop set to True, it will be clipped to the intersection of the feature and current extent.

Interpolation can either be 0-nearest neighbor (default), 1-bilinear, 2-cubic convolution.

Working with bands

Most processing occurs on individual bands, which as seen above, can be referenced by band index (0-based) or by band name. A GeoRaster cannot be created on it's own, it is always a member of a GeoImage. GeoRaster has a variety of processing functions, all basic mathematical operators, logical and bitwise operators, among others.

Bands can be treated as any other single entity, and processing applied to a raster band, as per the convention explained above, returns a new raster band. To apply processing to a band directly in an image, simply replace that band with the returned band with processing applied:

```
geoimg['red'] = geoimg['red'] + 7
geoimg['red'] = geoimg['red'].sqrt()
```

The red band is now processed by adding 7, then taking the square root. As another example, to convert a landsat-7 ETM+ image from radiance to top of the atmosphere reflectance, where theta is the solar zenith angle and sundist is the earth-sun distance:

> green_toa = img['green'] * (1.0 / ((1812.0 * numpy.cos(theta)) / (numpy.pi * sundist * sundist)))

The 1812.0 is the exoatmospheric solar irradiance for the green band, as given in the [Landsat handbook](http://landsathandbook.gsfc.nasa.gov/data_prod/prog_sect11_3.html).

This green_toa band can then be further processed, but none of the calculations anywhere in the chain will be performed until a read is requested through the read() or save() functions.

> # get a numpy array of the byte scaled green TOA reflectance green = green_toa.autoscale(1, 255).read().astype('byte')

Masks

A mask is not a special entity, any GeoRaster can act like a mask where 0 is off (invalid, or masked), and non-zero is on (valid). Masks can be added to a GeoRaster, in which case any subsequent processing will respect the mask values.

```
# calculate the stats of the blue band where red > 0.5
stats = geoimg['blue'].add_mask(geoimg['red'] > 0.5).stats()
```

Multiple masks can be added using the add_mask() function, while all masks can be cleared using clear_masks(). The mask can itself have other processing applied to it, which will be done, in chunks, when it is read for the purpose of applying the mask.

GeoRaster processing functions

These processing functions all operator on a raster band and return a raster band. They are pointwise operations, where the output, if read, will be of the same size as the input.

Arithmetic +, -, *, /

Logical >, >=, <, <=, ==

Bitwise bxor (XOR)

Filters convolve, laplacian

Exponential pow, sqrt, log, log10, exp

Trigonometric cos, sin, tan, acos, asin, atan, cosh, sinh, tang, sinc

Other min (pointwise min), max (pointwise max), abs, sign

Statistical functions

In addition to the processing functions, there are several functions for calculating band statistics.

stats() Calculate and return array of image statistics: min, max, mean, stddev, skewness, kurtosis

histogram(bins=100, normalize=true, cumulative=false) The histogram, or cumulative histogram. If normalized the sum of all frequencies will be 1.0

percentile(p) Calculate the pixel value for this percentile (uses the cumulative distribution histogram)

Reading and Chunking

The read function can be called on a GeoRaster, in which case a 2d array is returned, or on a GeoImage, which will return a 3d array. The read() function takes in an optional chunk, and if not provided will default to the entire image.

When using a Gippy algorithm or save() function, the image is automatically chunked up into pieces no larger than a default chunk size and the processing applied in pieces. To write code that uses numpy but also utilizes chunking, use the GeoImage.chunks() function to generate a collection of chunks that cover the image. The total number of chunks can be specified, as can a padding factor to mitigate border effects in the case of local area operations.

```
for ch in geoimg.chunks(numchunks=100):
    arr = geoimg['red'].read(chunk=ch)
    # do something with this chunk and write it back
    geoimg['red'].write(arr, chunk=ch)
```

When data is actually read by the user it is returned as a numpy array, and numpy arrays may be passed into the GeoImage.write() function. If the number of chunks is not specified they are calculated to be no larger than a default chunk size, given in MB. This chunk size can be set in gippy's global options.

There are some other read type functions that also take in a chunksize. The data_mask() and nodata_mask() return an array of 0 and 1 indicating where data is and where nodata is, respectively. The read_raw() function reads the data as it is stored, but without applying any gain, offset, or processing.

Working with vectors

Gippy provides basic support for vector data, following the same concept as a GeoImage, where a GeoVector is a container class for GeoFeature objects. Any OGR datasource can be opened as a GeoVector, is iterable, and features can be indexed.

```
from gippy import GeoVector

geovec = GeoVector(filename)

# this is the extent of all features
full_extent = geovec.extent()

# this is the extent of the first feature
f0_extent = geovec[0].extent()
```

The GeoVector and GeoFeature are not fully featured, and are more designed as supporting classes for raster operations (e.g., warping), rather than providing extensive vector support on their own, as there are other vector libraries for more general use.

Algorithms

Gippy includes an algorithm module, which is a collection of functions that operate on GeoImage objects. There are currently only a handful of functions, but this will be expanded upon in future versions. Gippy algorithms take in one or more images, parameters unique to the algorithm, and an output filename and output file options. Currently the algorithms module includes indices() which can calculate a variety of indices (e.g., NDVI, NDWI, LSWI, SATVI) in one pass, acca() and fmask() cloud detection algorithms for Landsat7, cookie_cutter() for mosaicking together scenes, linear_transform() for applying basis vectors to spectral data, pansharpening, and a rxd(), a multispectral anomaly detection algorithm.

```
from gippy.test import get_test_image
import gippy.algorithms as alg

geoimg = get_test_image()

index_image = alg.indices(geoimg, products=['ndvi', 'ndwi'])

print(index_image.bandnames())

> ['ndvi', 'ndwi']
```

acca(geoimg, filename) The ACCA algorithm operates on a GeoImage of Landsat7 data and must contains bands for Red, Green, NIR, SWIR1, and LWIR. Because it utilizes empirically derived constants it is currently only suitable for Landsat7.

fmask(geoimg, filename) Like ACCA, Fmask is currently only suitable for Landsat7 data. It requires bands for Blue, Red, Gree, NIR, SWIR1, SWIR2, and LWIR.

cookie_cutter([geoimgs], filename, feature, crop, proj, xres, yres, interpolation) The cookie_cutter algorithm takes in a list of GeoImage objects, and optionally a feature representing the region of interest. A mosaic will be created from all input images and cutting to the feature. If the crop keyword is set to True, the extent of the output will be the intesection of the feature and all the images, otherwise, it will be the extent of the feature.

indices(geoimg, products, filename) This calculates the desired indices given by the products list, and creates a single file with one index for each band. Indices currently supported: NDVI, EVI, LSWI, NDSI, NDWI, SATVI, MSAVI2.

linear_transform(geoimg, coef, filename) This applies the matrix of coefficients to create a new series of bands. The coef matrix must be of size #bands x #bands and will create the band outputs that are linear combinations of the inputs. Useful for transforming images into principal components by providing the Eigenvectors.

pansharpen(geoimg, panimg, weights, filename) This performs a Brovey pansharpening algorithm on the input image. First geoimg is upsampled and panimg shifted so that they both cover the same footprint. Pansharpening then uses the panimg values, and the weights (if provided) to get the pansharpened multispectral input. The input image must contain 3 or 4 bands, and if provided weights must be the same length.

rxd(geoimg, filename) The RX Detector algorithm is a simple anomaoly detector that reduces a multiband image into a single band that indicates how the spectral signature of pixels deviates from an average.

Indices and tables

- search