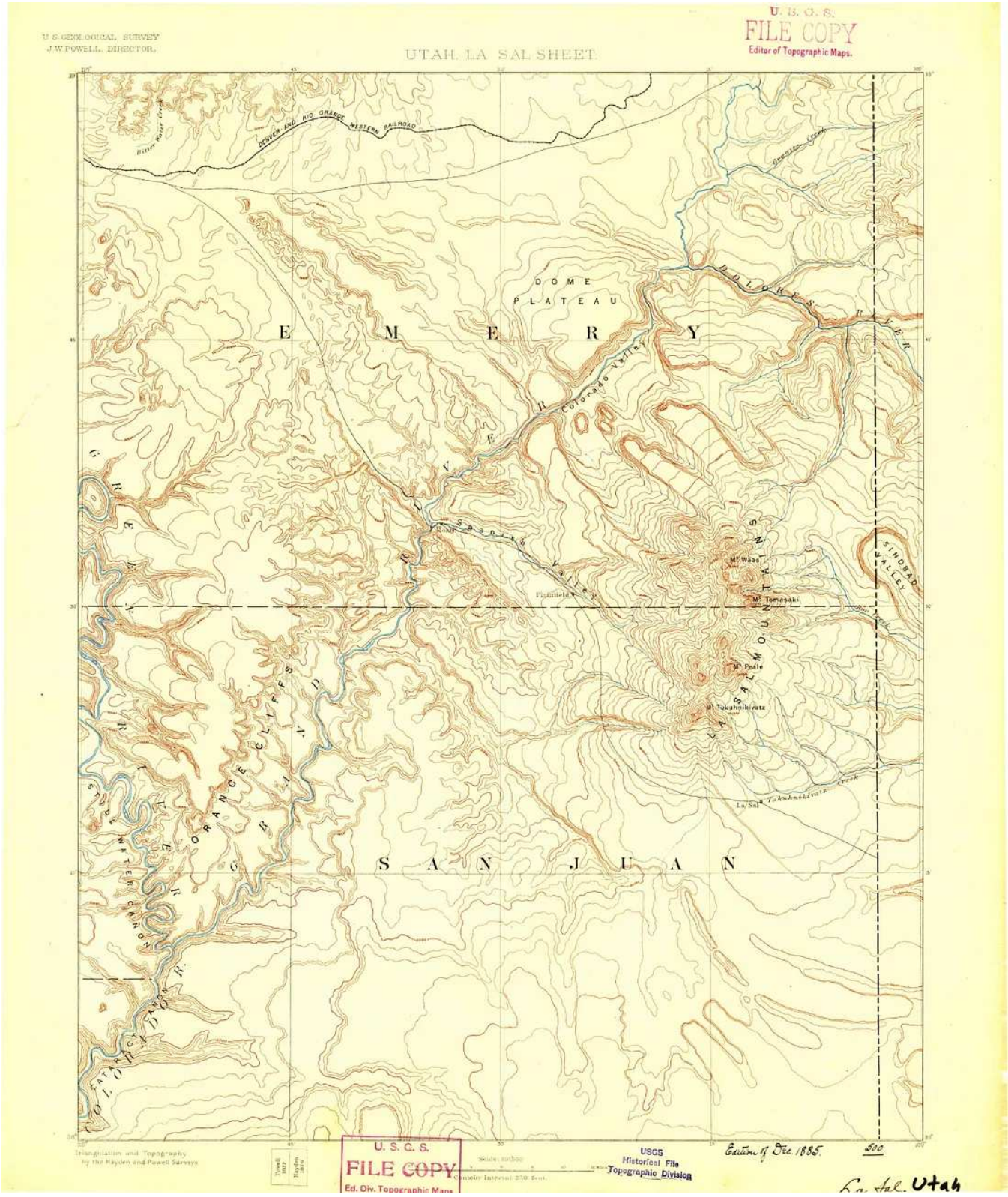
Sunset at the Dollhouse. Maze District, Canyonlands National Park. Photo courtesy National Park Service.

A Gentle Introduction to GDAL, Part 1

Robert Simmon · Follow

Published in Planet Stories
9 min read · Apr 4, 2017

In 1885 the United States Geological Survey published their first map of the Canyonlands, a remote, rugged, high, dry, and otherwise inaccessible chunk of desert in southeastern Utah.



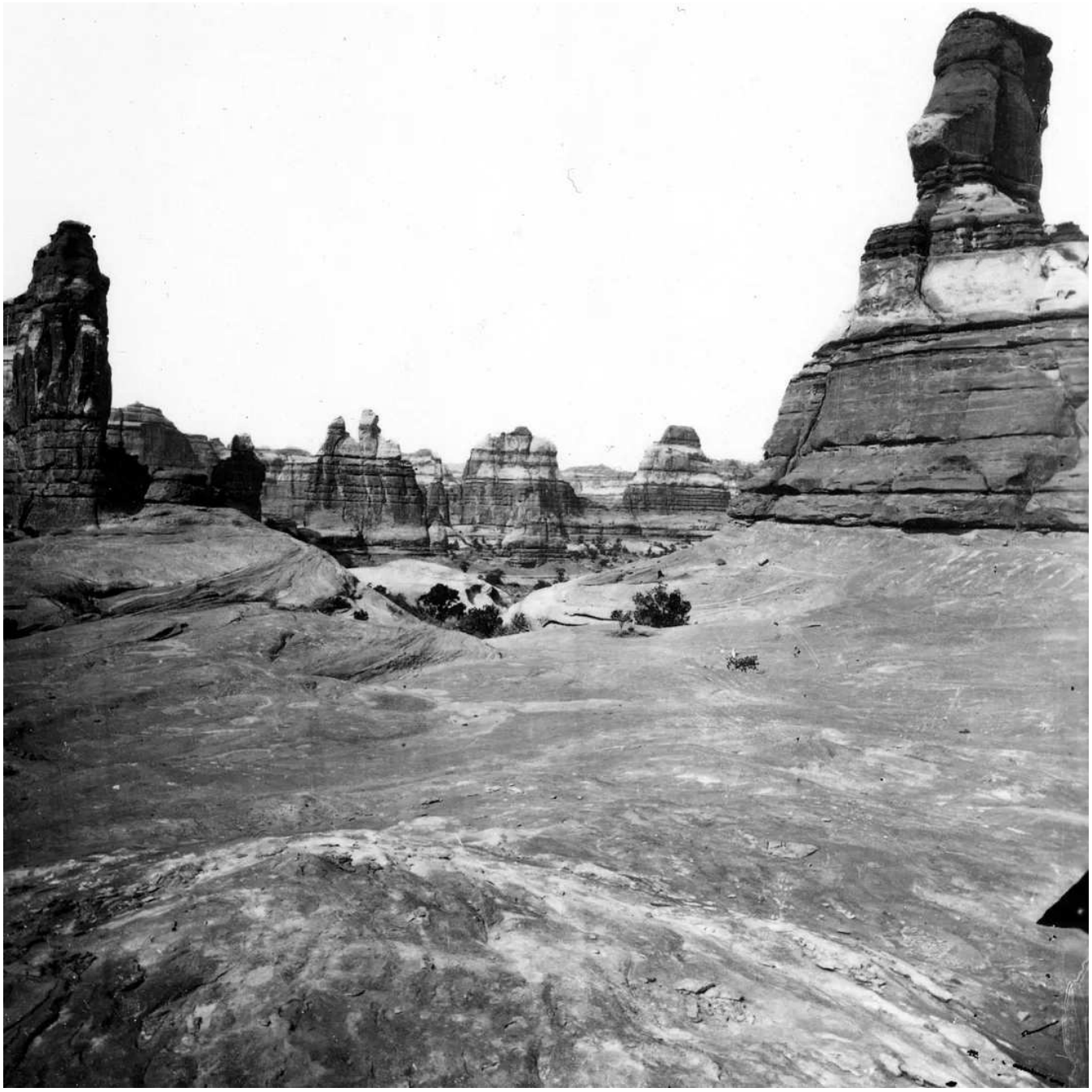
USGS 1:250000-scale Quadrangle for La Sal, UT 1885. Via the National Geologic Map Database.

These maps were based on the “triangulation and topography” recorded during a pair of expeditions lead by Major John Wesley Powell, who later become a director

of the Survey.

In a roadless landscape heavily sculpted by erosion, the easiest way through was via the rivers that wound their way through steep-walled canyons, so Powell and his men rafted down the Green and Colorado Rivers (then called the Grand River). Over the course of two journeys in 1869 and 1871, the Powell expeditions carefully mapped the landscape as they went.

Unfortunately, they were largely stuck at the bottom of those canyons: a hike to the top to get a better view was usually arduous and often impossible. Once at the rim of the canyons the men were met with inhospitable slickrock, carved into fantastic shapes, which further inhibited travel and blocked sight-lines.



On the top of the country immediately at the junction of the Grand and Green Rivers, westside. Utah. Photograph taken by the second Powell expedition, September 17, 1871. Photo courtesy USGS.

The resulting maps were largely accurate with respect to the courses of major rivers and the locations of high peaks, but were lacking in fine detail. Largely absent were the slot canyons, grabens, arroyos, arches, pinnacles, salt domes and other geological curiosities that make the area so fascinating today.

Not that any blame lies on Major Powell or his crew: the area is so inaccessible and

difficult to traverse that the USGS didn't substantially update its maps for over *65 years*—new printings were issued in 1896, 1901, 1909, 1918, and 1923 with only minor typographic changes. Large-scale maps of the Canyonlands weren't published until the 1950s—among the last parts of the continental United States to be charted in detail.

Comparison of the USGS 1:250,000-scale topographic maps of the confluence of the Green and Colorado Rivers drawn in 1885 (1923 printing, left) and 1959 (right). From the USGS National Geological Map Database.

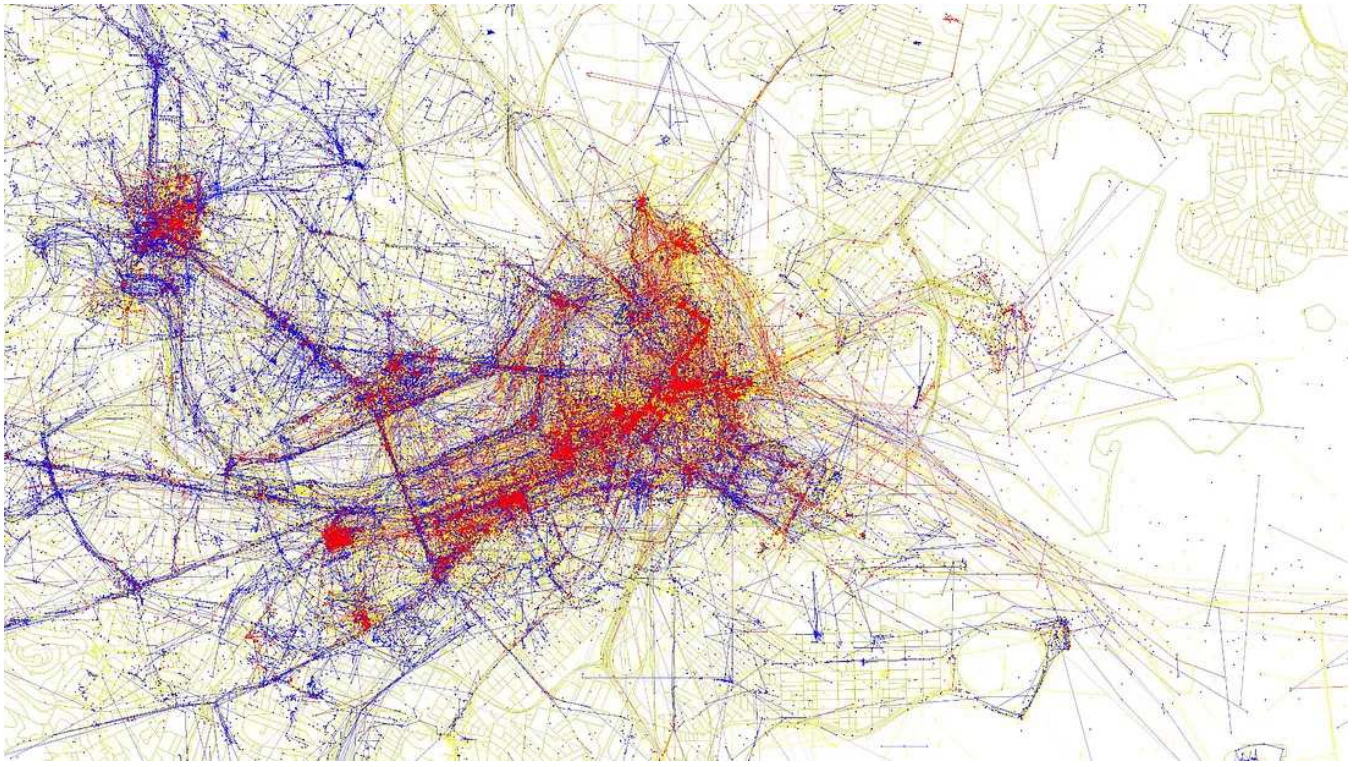
Two factors led to these radical improvements after decades of stasis—advances in photogrammetry which enabled map-makers to precisely determine topography from overlapping aerial photographs, and a post-war uranium boom that inspired prospectors to crawl over ever inch of the convoluted landscape.

Aerial photograph of the Canyonlands taken by the USGS in 1953. The Green River merges with the Colorado River in the center of the frame. Photograph from the USGS Earth Explorer.

It's been another 65 years since the terrain of the Canyonlands was finally mapped, and imagery from above is now ubiquitous. Once-remote parts of the globe are now pictured by aircraft, satellites and drones; using visible light (red, green, blue and the other colors of the rainbow), infrared light, thermal radiation, LIDAR, and radar from a scale of centimeters to kilometers. Weather satellites capture images of an entire hemisphere every 15 minutes, and Planet (my company) will soon be imaging

the entire surface of the Earth (well, the cloud-free bits at least) at about 4 meters per pixel (the size of a small building) every day.

Accompanying this explosion of raster (pixel-based) data was growth in the amount and availability of vector data—everything from county boundaries to geological maps to GIS locations attached to photographs on flickr. This bewildering array of data is accompanied by a bewildering array of data *types:* file formats, scales, map projections, datums, etc., etc., etc.



Locals and Tourists, Boston. Blue pictures are by locals. Red pictures are by tourists. Yellow pictures might be by either. ©2010 Eric Fisher, CC BY-SA 2.0.

Why GDAL?

Given that these data are (usually) much more useful combined than separate, how does one integrate disparate data into interesting and novel composites?

Broadly speaking, the answer is a **GIS**—geographic information system (perhaps the first and last time I link directly to a Wikipedia article). “A system designed to capture, store, manipulate, analyze, manage, and present spatial or geographic data.”

The bad news is that the most pervasive GIS software is expensive, difficult to learn, and won't even run on my operating system of choice. The good news is that there's an open-source alternative—GDAL—that's free, broadly supported, constantly updated, and runs on almost anything. Excepts it's also difficult to learn, especially if you're terrified of the command line, like me.

GDAL stands for "Geospatial Data Abstraction Library":

> *A translator library for raster and vector geospatial data formats that is released under an X/MIT style Open Source license by the Open Source Geospatial Foundation. As a library, it presents a **single raster abstract data model** and **single vector abstract data model** to the calling application for all supported formats. It also comes with a variety of useful command line utilities for data translation and processing.*

Even the description is complicated. Put another way, GDAL (pronounced 'gee-dal' not 'goodle' (too close to Google!)) is a collection of software that helps with the translation of data from different file formats, data types, and map projections. It's used in free software like QGIS and GRASS, and even some commercial applications like ESRI ArcGIS and Avenza Geographic Imager/MAPublisher.

Installing GDAL

As I already mentioned GDAL runs on the command line, but is only a little bit harder to install than the typical commercial app.

OSX

On a Mac, I've found the easiest approach is to download the GDAL 2.1 Complete disk image from KyngChaos (I know it sounds like a mid-80s hacker collective distributing bootleg C64 games, but the site's legit). This installer relies on the version of Python included with OS X, in my case 2.7.12, not 3.x.

On the disk image are two installers: one for NumPy (run this first) and one for GDAL Complete (run this second). You'll have to open the files with a control-click since the installers are from an "unknown developer." (Don't worry, it's still ok. (I think.)) After one last step you'll be good to go: cut and paste the following two lines (borrowed from MapBox) into the Terminal (they'll allow you to run GDAL

commands just by typing them, without specifying a path. bash_profile is a hidden file, so don't be surprised if you can't find it.)

```
echo 'export PATH=/Library/Frameworks/GDAL.framework/Programs:$PATH'
>> ~/.bash_profile

source ~/.bash_profile
```

Windows

Follow these instructions to install both Python 2.7 and GDAL 2.1 from the UCLA Institute for Digital Reseach and Education. (Be careful to set the paths correctly for a 32- or 64-bit Windows install.)

Linux

If you're running Linux you know more about this than I do—you're on your own.

An alternative which works on all platforms is to use the the Conda package manager. If you're already running Conda you can install GDAL with:

```
conda install -c conda-forge gdal=2.1.3
```

First Steps Using GDAL

To make sure GDAL is installed, open up a command line window and type:

```
gdalinfo --version
```

If you see something like:

```
GDAL 2.1.2, released 2016/10/24
```

You're good! (Hmm, it looks like I'm a tiny bit out of date.)

Now, let's download a map to work with—a shaded relief of Canyonlands, cropped from the U.S. National Park Service map (which has a fantastic cartography center, BTW).

Shaded relief map of the Canyonlands. Courtesy National Park Service Harpers Ferry Center.

gdalinfo

Now that you've got an image, navigate to the folder you downloaded to and enter this into the command line:

```
gdalinfo CANYrelief1-geo.tif -mm
```

"gdalinfo" runs the eponymous utility program (one of many included with GDAL), "CANYrelief1-geo.tif" is the name of the file you just downloaded, and "-mm" is a command that calculates and displays additional information. After hitting return, you should see a block of text, beginning with:

```
Driver: GTiff/GeoTIFF
```

This indicates that the file is a GeoTIFF, which is a special type of TIFF that stores the information necessary to place each pixel in the image on the surface of the Earth. It's an incredibly flexible and useful format, and is increasingly by adopted to store more types of data than just images.

Reading further, the next lines of text are:

```
Files: CANYrelief1-geo.tif
Size is 2800, 2800
```

These simply indicate the file name, and the size (in pixels) of the image. Here's the really important bit:

```
Coordinate System is:
PROJCS["WGS 84 / Pseudo-Mercator",
GEOGCS["WGS 84",
DATUM["WGS_1984",
SPHEROID["WGS 84",6378137,298.257223563,
AUTHORITY["EPSG","7030"]],
```

```
AUTHORITY["EPSG","6326"]],
PRIMEM["Greenwich",0,
AUTHORITY["EPSG","8901"]],
UNIT["degree",0.0174532925199433,
AUTHORITY["EPSG","9122"]],
AUTHORITY["EPSG","4326"]],
PROJECTION["Mercator_1SP"],
PARAMETER["central_meridian",0],
PARAMETER["scale_factor",1],
PARAMETER["false_easting",0],
PARAMETER["false_northing",0],
UNIT["metre",1,
AUTHORITY["EPSG","9001"]],
AXIS["X",EAST],
AXIS["Y",NORTH],
EXTENSION["PROJ4","+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0
+lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null +wktext
+no_defs"],
AUTHORITY["EPSG","3857"]]
Origin = (-12249462.599999999627471,4629559.794860946945846)
Pixel Size = (13.284000000000001,-13.285397060378999)
```

This is the information that places this image in the Canyonlands, and specifies the location of each pixel. I'll get into map projections in my next post, but if you want to learn more the USGS has a nice primer (PDF)—or just read XKCD. The last entry is the pixel size in *meters*—defined by the "unit" entry further up.

Next up is some generic metadata for the file:

```
Metadata:
AREA_OR_POINT=Area
```

```
TIFFTAG_DATETIME=2017:04:01 20:24:57
TIFFTAG_RESOLUTIONUNIT=2 (pixels/inch)
TIFFTAG_SOFTWARE=Adobe Photoshop CC (Macintosh)
TIFFTAG_XRESOLUTION=72
TIFFTAG_YRESOLUTION=72
Image Structure Metadata:
COMPRESSION=LZW
INTERLEAVE=PIXEL
```

This tells me that I made the image on April Fool's Day, I used Photoshop to crop and save the file (with a plugin that keeps the geographic info intact), the resolution (if printed) is 72 dots per inch, and the image is saved with LZW, a type of lossless compression.

```
Corner Coordinates:
Upper Left  (-12249462.600, 4629559.795) (110d 2'19.66"W,
38d21'14.51"N)
Lower Left  (-12249462.600, 4592360.683) (110d 2'19.66"W, 38d
5'29.42"N)
Upper Right (-12212267.400, 4629559.795) (109d42'16.79"W,
38d21'14.51"N)
Lower Right (-12212267.400, 4592360.683) (109d42'16.79"W, 38d
5'29.42"N)
Center      (-12230865.000, 4610960.239) (109d52'18.23"W,
38d13'22.39"N)
Band 1 Block=2800x31 Type=Byte, ColorInterp=Red
Computed Min/Max=149.000,255.000
Band 2 Block=2800x31 Type=Byte, ColorInterp=Green
Computed Min/Max=133.000,255.000
Band 3 Block=2800x31 Type=Byte, ColorInterp=Blue
Computed Min/Max=100.000,255.000
```

The last block of text displayed by gdalinfo shows the corner points of the image in

two different units (meters and minutes, degrees, seconds) and *finally* some information about each band (also called a channel)in the image. Each channel is byte (8-bit) format, there are three bands (red, green, and blue, respectively), and the minimum and maximum values in each band.

That last detail was calculated and displayed because I added "-mm" (min/max) to the command. If you throw in "-stats" you'll get additional statistics like mean, median, and standard deviation. But be careful, some software (QGIS) may write *estimates* of these values into the file header and gdalinfo will return the wrong values if you just use "stats", whereas "-mm" will force statistics to be calculated on the full dataset. Slow but thorough.

gdal_translate

Now that we have some information about the file, let's do something useful—resizing the image to a web-friendly 1,400-pixels-wide and saving it as a JPEG—with "gdal_translate". Enter these commands into the terminal:

```
gdal_translate -of JPEG -co QUALITY=70 -co PROGRESSIVE=ON -outsize
1400 0 -r bilinear CANYrelief1-geo.tif CANYrelief1.jpg
```

You'll see two lines of text as a result, and have a brand-new file, CANYrelief1.jpg, in the same directory as the TIFF. I specified the file type, size, etc. with the following commands:

```
-of JPEG -co QUALITY=70 -co PROGRESSIVE=ON
```

"-of" sets the output format, in this case JPEG. "-co QUALITY=90" and "-co PROGRESSIVE=ON" are *creation options,* a series of commands specific to each file type, in this case writing a JPEG with a quality of 90 out of 100 that will load progressively.

```
-outsize 1400 0 -r bilinear
```

These two commands indicate set the output size and interpolation method of the resizing. "-outsize" is set in pixels—x (horizontal) first and y (vertical) second. In this case I set y to zero, so the image kept it's original aspect ratio. I would have gotten the same results with "-outsize 1400 1400". "-r" specifies the resampling method, of which GDAL offers a generous selection (nearest, bilinear, cubic, cubicspline, lanczos, average, mode). "Nearest" is the default, but it leaves visible stairsteps so you'll almost always want to change it. I usually use "bilinear" for satellite imagery since it's quick and I sharpen as an additional step.

Finally:

```
CANYrelief1-geo.tif CANYrelief1.jpg
```

These are just the *input* file name and the *output* file name, respectively (but be careful, gdal_translate will happily overwrite files with no warning).

gdalinfo and gdal_translate are two of the more straightforward utilities included with GDAL. And, to be honest, there are a thousand ways to convert a TIFF to a JPEG —but not nearly so many are also able to read the headers in a geotiff, translate obscure data formats (if you do data visualization long enough, you'll run into some *special* files), or transform a map from one projection to another. Next up is Part 2: Map Projections & gdalwarp, which begins to unlock the power of geospatial data, but requires some familiarity with map projections (spoiler: the Earth isn't flat).

Further Reading

- Canyonlands National Park
- GDAL.org
- GDAL cheat sheet, by Derek Watkins
- Locals and Tourists, by Eric Fischer
- USGS Earth Explorer

Astronaut photograph ISS050-E-66060 courtesy NASA Earth Observation Lab.

A Gentle Introduction to GDAL, Part 2: Map Projections & gdalwarp

Robert Simmon · Follow
Published in Planet Stories
10 min read · Apr 13, 2017

(If you're new to GDAL, you might want to start with Part 1, which covers installation, `gdalinfo`, and `gdal_translate`.)

From ground level, an unobscured view of the horizon looks ruler-flat, an illusion that persists even from the roughly 30,000-foot-altitude of an airliner or the summit of Mount Everest. It's only at twice that height, an elevation few of us will ever reach, that the first hint of a curve becomes visible.

This view was first seen in 1935 by two Army officers floating above the plains of South Dakota, aboard the Explorer II helium balloon. From a press release issued by *National Geographic:*

> *From its vantage point 72,395 feet in the air, the highest point ever reached by man, the camera registers the horizon 330 miles away, sweeping like a great arch across the*

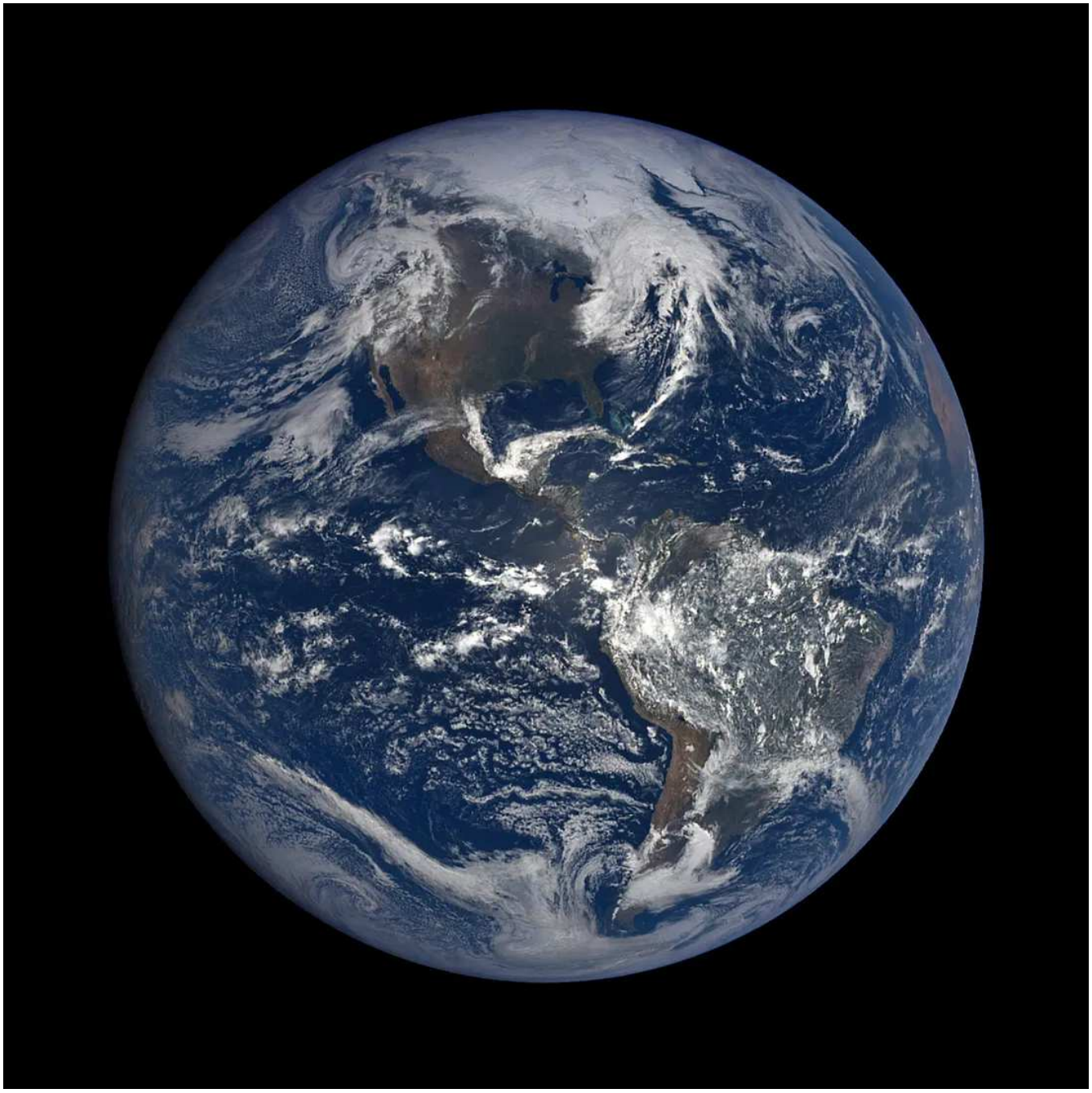
> *photograph. The straight black line ruled across the top brings out the curvature of the Earth.*

The first photograph showing a curve on Earth's horizon, taken on November 11, 1935. Courtesy Jerry Bryant, Lawrence County Historical Society.

A balloon can fly into the stratosphere, but no higher, so it wasn't until the advent of the space race that we got a better view of the edge of the Earth. From an altitude of several hundred miles one's view stretches for thousands of miles, and the curvature of the Earth is clear.

Large parts of California and Nevada are visible in this photograph taken from the Internation Space Station on October 19, 2014. Astronaut photograph ISS041-E-081461 courtesy NASA Earth Observation Lab.

From an even higher vantage point (900,000 miles), the Earth becomes a flat disk on a black field. (Even at this distance the stars are likely washed out by glare from the Earth.)



The Western Hemisphere on April 7, 2017. From DSCOVR's EPIC instrument.

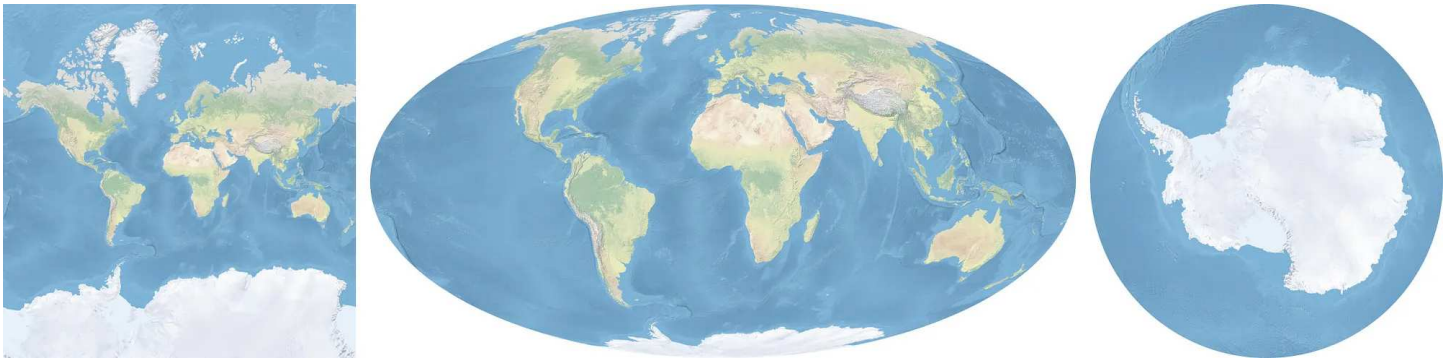
Each of these views is a distorted, two-dimensional representation of a three-dimensional surface. The appearance of the Earth is determined by the vantage point and optical characteristics of the camera taking the picture. Some features (those directly under the observer) are preserved very well, but others become quite deformed—it's impossible to take a photograph that accurately represents the distances, areas, and angles of every part of a 3D object.

This is the fundamental challenge faced by cartographers—how to represent the 3D surface of the Earth on a flat piece of paper or computer screen? It's also a challenge that mapmakers faced long before anyone ever saw the horizon as a curve.

The earliest known globe was crafted in 1492 — the same year Columbus sailed to America (but before he came back—note the lack of the Americas). To construct a globe this printed map would have been cut out and folded into a sphere, with the poles pasted on. Composite: Globe Gores 1–4. Martin Behaim's Erdapfel, 1492. Courtesy David Rumsey Map Collection.

Since a perfect representation isn't possible, cartographers have developed an incredible variety of map projections, each designed to solve a particular problem. Every map is a compromise favoring one or more projection properties: area, form (or angle), distance, and direction.

Do you need to navigate a ship across the Atlantic? Use the Mercator projection, which keeps lines of latitude parallel. Show global temperature? Use a projection that preserves area, like Mollweide. Make a reference map? Use a compromise projection like Robinson. Show Antarctica or sea ice? Use polar stereographic.



Mercator, Mollweide, and Southern Hemisphere polar stereographic maps. Made with Natural Earth.

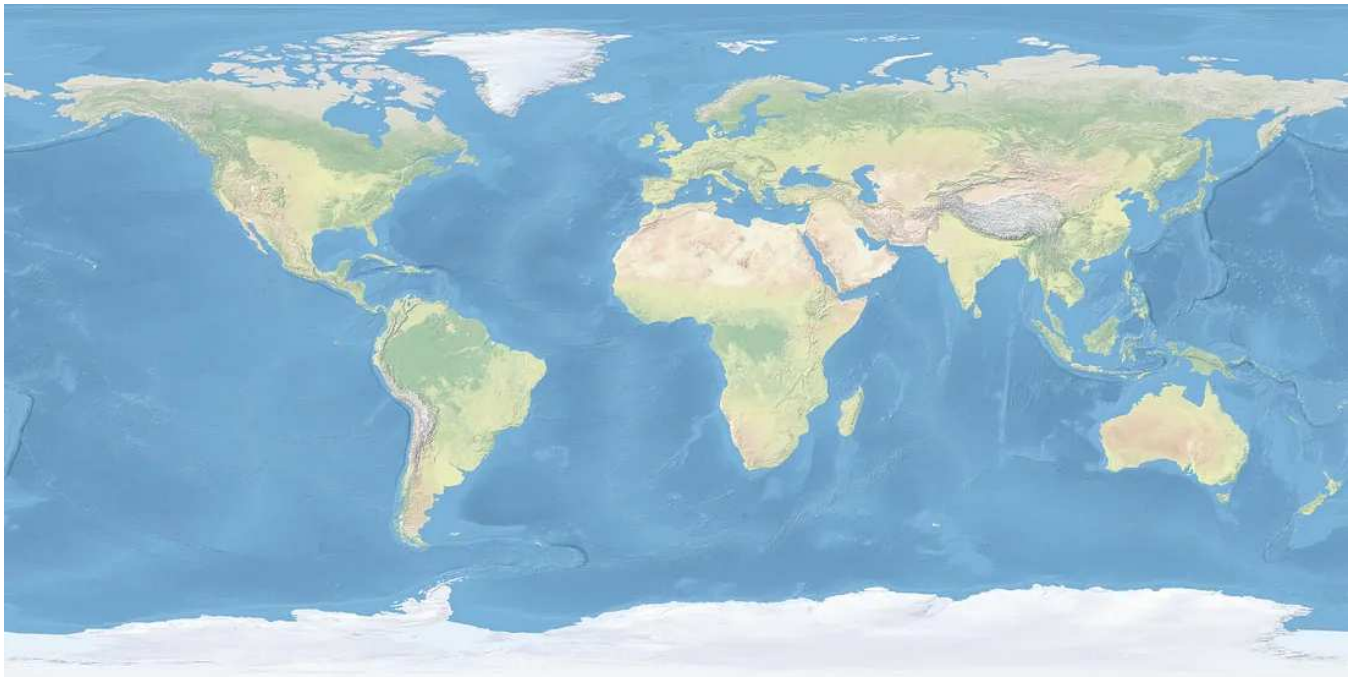
To get started, download a global geo-referenced image like Natural Earth I with Shaded Relief and Water (zip file). This is a very nice map of the world with muted colors, based on NASA's Blue Marble (yeah!) with shading for mountain ranges and the ocean floor.

Once you've downloaded and unzipped the TIFF, run `gdalinfo NE1_50M_SR_W.tif` to take a look at the metadata, particularly the coordinate system:

```
Coordinate System is:
GEOGCS["WGS 84",
DATUM["WGS_1984",
SPHEROID["WGS 84",6378137,298.257223563,
AUTHORITY["EPSG","7030"]],
AUTHORITY["EPSG","6326"]],
PRIMEM["Greenwich",0],
UNIT["degree",0.0174532925199433],
AUTHORITY["EPSG","4326"]]
```

Of particular note are the units `degrees` and the line starting with "authority" `"EPSG","4326"`. *Unit* indicates that each pixel is related to a real-world unit, not just pixels—degrees of latitude and longitude. The bracketed entries after *authority* (detailed explanation) are a shorthand reference for the map projection—EPSG:4326

—which is simply a grid of equal latitude and longitude. This results in a world map that has a 2:1 aspect ratio, and goes from 180˚ west to 180˚ east and 90˚ north to 90˚ south. Of course it's a bit more complicated than that (in ways I'll describe later) and the full definition is lengthy, so having a simple 4-digit number makes things easier to type.



Natural Earth I with Shaded Relief and Water. This is in a very common projection with way too many names: Plate Carrée, equirectangular, equidistant cylindrical, simple cylindrical, rectangular, lat-lon, geographic projection, WGS 84, or EPSG:4326. It's simply an even grid of latitude and longitude, centered at 0˚ north 0˚ south. Made with Natural Earth.

The file is big (10,800- by 5,400-pixels), so let's make it a bit smaller using gdal_translate:

```
gdal_translate -r lanczos -tr 0.1 0.1  -co COMPRESS=LZW
NE1_50M_SR_W.tif NE1_50M_SR_W_tenth.tif
```

If you read Part 1, most of this should look familiar. The only new bit is `-tr 0.1 0.1`, which sets the target resolution in real-world units. In this case these are degrees of latitude and longitude (as revealed by `gdalinfo`). Other common options are meters or feet (which you might run into with some projections in the U.S.)

gdalwarp & the Mercator Projection

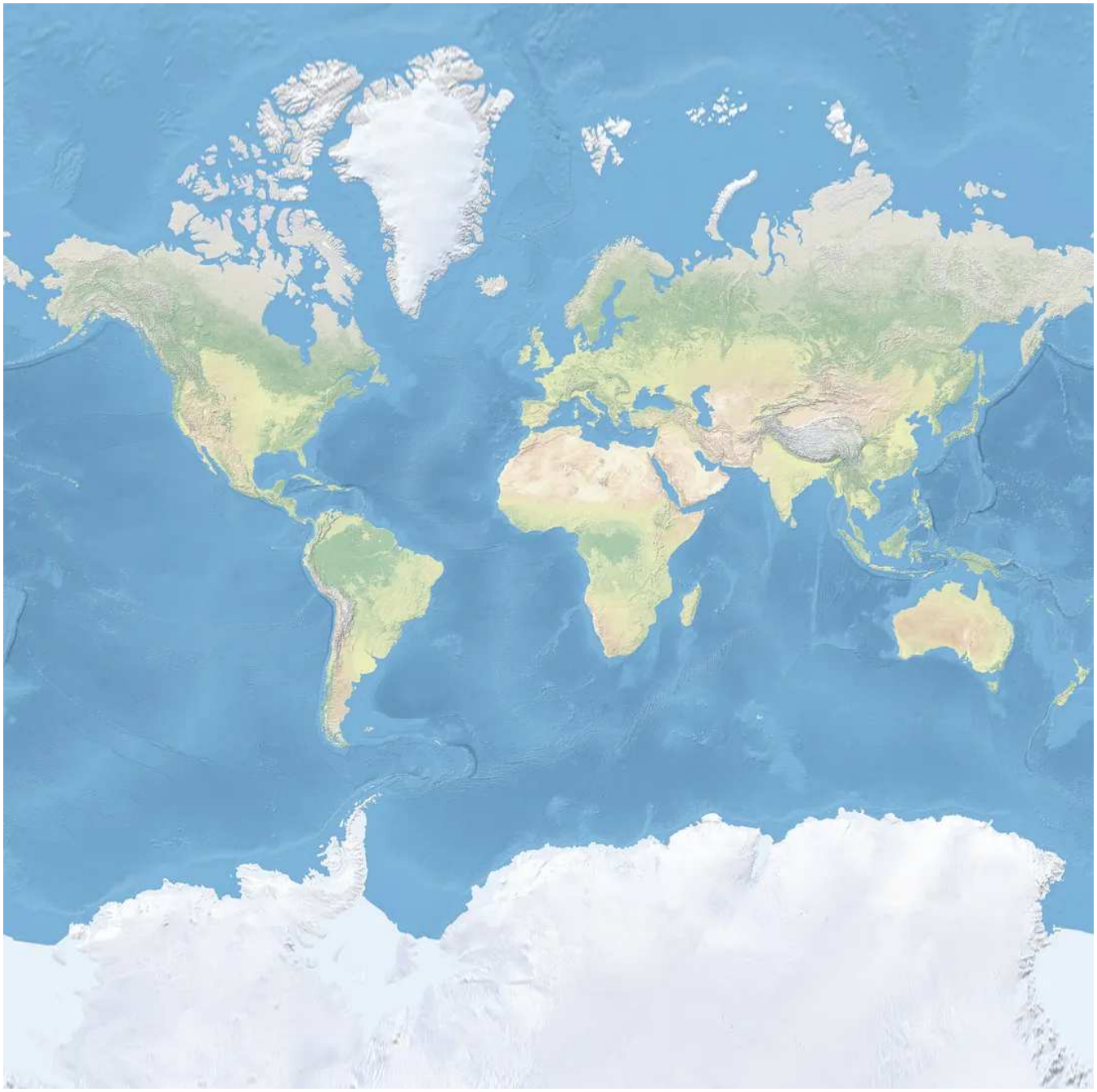
Now let's convert this map from Plate Carrée/rectangular/lat-lon/etc. to Mercator, using another GDAL utility, `gdalwarp`:

```
gdalwarp -t_srs EPSG:3395 -r lanczos -wo SOURCE_EXTRA=1000 -co
COMPRESS=LZW NE1_50M_SR_W_tenth.tif NE1_50M_SR_W_tenth_mercator.tif
```

Breaking down the code: `gdalwarp` invokes the command, while `-t_srs EPSG:3395` sets the *target source reference system* to EPSG:3395, which is the catalog number for Mercator (I'll go into other ways to do this in a bit). There are a few ways to find these, spatialreference.org is especially helpful because it displays the description of a map projection in several formats.

`-r lanczos` defines the resampling method, with a few more options than `gdal_translate`. Lanczos is slow but high quality. `-wo SOURCE_EXTRA=1000` is an example of a *warp option*—advanced parameters that determine how the reprojection is calculated. `SOURCE_EXTRA` adds a buffer of pixels around the map as it is reprojected, which helps prevent gaps in the output. Not all reprojections require it, but it doesn't hurt to add the option to be on the safe side. `-co COMPRESS=LZW` works just the same as it does in `gdal_translate`, and `NE1_50M_SR_W_tenth.tif NE1_50M_SR_W_tenth_mercator.tif` are the input and output filenames, respectively.

Run the command, and you should see a map that looks like this (but slightly larger):



An (almost) global map using the Mercator projection. Since the Mercator projection stretches to infinity at the poles, the highest northern and southern latitudes are automatically clipped via mysterious GDAL guesstimates. Made with Natural Earth.

Success!

You may notice that the resulting map doesn't go all the way to 90˚ north or 90˚ south—the Mercator projection becomes infinitely large at the poles, so GDAL is making some guesses about how big the reprojected map should be. It's part of the

magic of GDAL! Which isn't without its downside. GDAL often behaves unpredictably—in this case, if you reproject the *original* Natural Earth TIFF instead of the smaller one the resulting Mercator will be a bit taller, stretching from 88.2˚ north to 88.2˚ south. I'll cover this more in part 3, but add the commands `-te -180 -88.2 180 88.2 -te_srs EPSG:4326` to force a taller map. It's often good practice to specifiy output extents, even for a global map, since it can save GDAL some guesswork (which can be slow, or in some cases, end badly).

Unfortunately, the Mercator projection has a glaring error at global scale: the further north or south you go the bigger things get. Greenland appears as big as Africa, while in reality it's only about 7 percent as large. And I won't even mention Antarctica, which is absolutely *huge* despite missing its interior.

So instead of making a map that preserves angles, let's make a map that preserves area. These are especially useful for *thematic maps*—i.e. maps that display data geographically, not just geography.

The Mollweide Projection

Mollweide is one of many equal-area projections, but has the additional nice property of maintaining straight lines of latitude (also called *parallels* because they run directly east-west and never meet). We could try to re-reproject the Mercator map to Mollweide, but it's best to minimize the transformations you apply to a dataset, so let's work with the rectangular map again. Run:

```
gdalwarp -t_srs '+proj=moll +lon_0=0 +x_0=0 +y_0=0 +ellps=WGS84
+datum=WGS84 +units=m +no_defs' -r lanczos -dstalpha -wo
SOURCE_EXTRA=1000 -co COMPRESS=LZW NE1_50M_SR_W_tenth.tif
NE1_50M_SR_W_tenth_mollweide.tif
```

(Note: this will generate *lots* of errors, it's just GDAL trying to figure out what to do with all the empty space in the corners.)

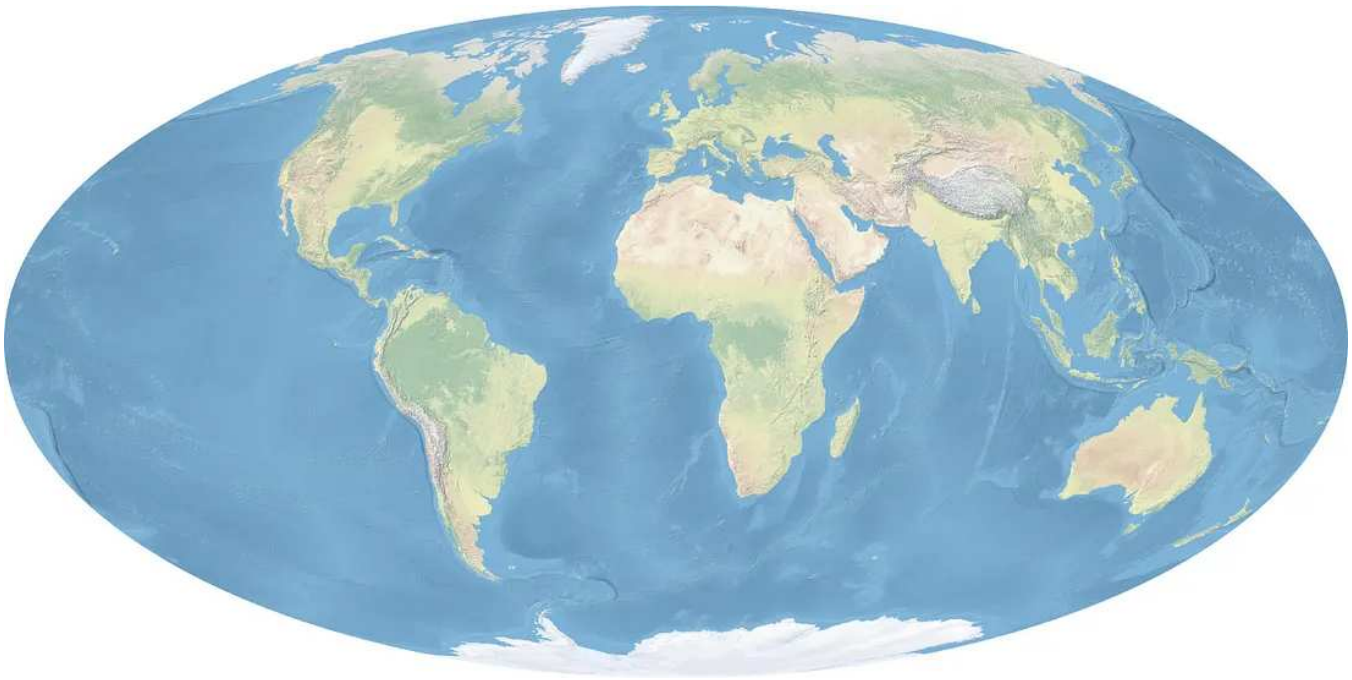
This is largely the same as the series of commands to make a Mercator map, but I've specified the target spatial reference system, `-t_srs` , piece by piece instead of relying on an EPSG code. For reasons I don't understand, some projections are un-

loved by the standards bodies, so they don't have EPSG codes. Mollweide is one of these, so you need to set the parameters manually, using the proj.4 syntax.

In this case `+proj=moll` sets the projection to Mollweide, which is followed by a string of settings that define parameters like the center of the map (`+lon_0` (try setting this to 175 if you're upset that New Zealand always gets short shrift on world maps)).

Rather than describing each property, I'll just recommend that you find the definitions for projections on the Spatial Reference site. It's got a nice search function, and defines each projection more than a dozen different ways. On the Mollweide page, for example, you'll see a list with "proj4" on it—click that link to get the full definition: `+proj=moll +lon_0=0 +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m +no_defs` . Enclose it in single quotes so it's parsed correctly.

Finally, I've added the option `-dstalpha` so the areas that are (literally) off the map are made transparent, instead of given a fill color (black by default).



The Mollweide projection, which is equal-area and has the bonus feature of maintaining straight lines of latitude. Made with Natural Earth.

Perfect!

The Polar Stereographic Projection

Well, no, not perfect—every map is a compromise, remember? What if we wanted to focus on the poles? They're undefined in Mercator, and shunted off to the edges on most other global maps, including Mollweide. Fortunately, there's a special map projection designed specifically for the Arctic and Antarctic: polar stereographic.

Some pre-processing is required to create a polar stereographic map using GDAL, which also gives me the opportunity to introduce the concept of a *virtual dataset* (VRT). A vrt is a text file that functions exactly like a georeferenced data file, but it's much **much** smaller. Polar stereographic maps become undefined towards the other pole, so the source data needs to be cropped down to the latitude that will become the edge of the map. I'll again use `gdal_translate`, but with some new options:

```
gdal_translate -of VRT -projwin -180 -60 180 -90
NE1_50M_SR_W_tenth.tif NE1_50M_SR_W_SH60.vrt
```

Instead of letting `gdal_translate` write the default GeoTIFF, I've used `-of` to specify the output format as a VRT. I've then cropped out Antarctica with `-projwin` specifying the boundary in the following format: upper left x, upper left y, lower right x, lower right y (but without the commas). In the Southern Hemisphere upper left longitude is -180˚, upper left latitude is -60˚, lower right longitude is 180˚, and lower right latitude is -90˚ (the South Pole).

With a nice compact (and quick to make) VRT, here's the command to make a map of Antarctica:

```
gdalwarp -t_srs EPSG:3976 -ts 7200 0 -r near -dstalpha -wo
SOURCE_EXTRA=1000 -co COMPRESS=LZW NE1_50M_SR_W_SH60.vrt
NE1_50M_SR_W_sh60_polarstereo.tif
```

It starts like the command for Mercator, specifying the *target spatial reference system* with an EPSG code: `EPSG:3976`. Then it gets (very slightly) weird. I've forced the *target size* to be 7,200 pixels with `-ts 7200 0` and then set *resampling* to nearest neighbor with `-r near`. This gets around a nasty issue that was blurring pixels along

the +/−180˚ line, at the expense of making the map look pixellated (at least if you zoom in). The rest mirrors the previous commands.

I set the target size to be so much larger than the resolution needed for this post (only 1,400 pixels across) so that I could reduce the resolution with a resampling method that averages pixels—bilinear. Here's a `gdal_translate` command to shrink the image and save it as a PNG, a nice lossless compression method (like GIF) that also retains full color (like JPEG) and displays on the Web.

```
gdal_translate -of PNG -outsize 1400 0 -r bilinear
NE1_50M_SR_W_sh60_polarstereo.tif
NE1_50M_SR_W_sh60_polarstereo_1400.png
```

A polar stereographic projection, centered on the South Pole and extending to 60˚ south. Notice how razor-sharp the edge is—that's the result of the render large then downsize technique, combined with lossless compression. Made with Natural Earth.

Huzzah!

A nice map of Antarctica extending from the South Pole to 60˚ north. I'll leave it as an exercise for the reader to make a similar map of the Northern Hemisphere.

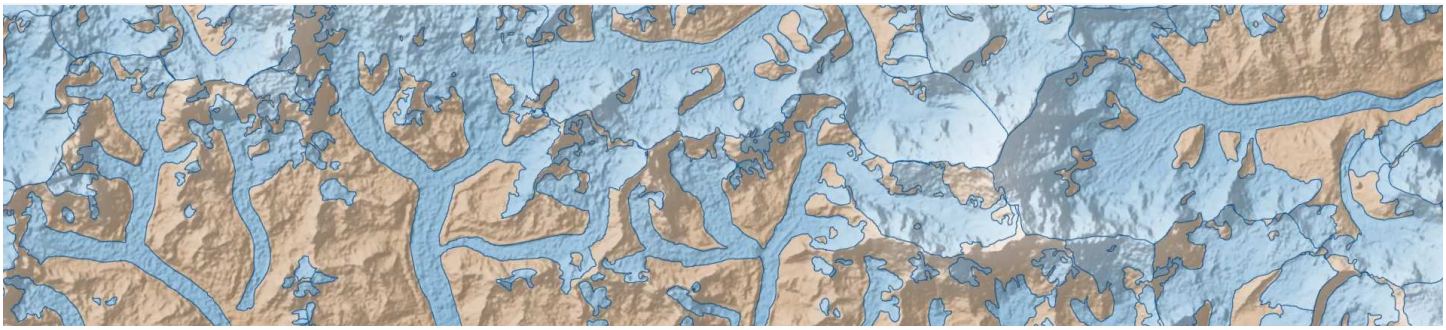
One final note: although GDAL is very powerful and very flexible, it can't do everything in mapping. Some projections, even some of my favorites like Hammer-Aitoff, Winkel-tripel, and Natural Earth, are not fully supported. Before trying out your favorite projection check the Spatial Reference database and if the proj.4 entry is blank, it probably won't work. The good news is support for some new map projections is underway.

Phew. Next up: geodesy, local map projections & `gdal_merge.py`.

Further Reading

- NASA's DSCOVR: EPIC page with real-time views of the sunlit Earth.
- JSC's Earth Observation Lab for a closer view.
- The David Rumsey Map Collection for a historical perspective.
- Natural Earth for comprehensive raster & vector data.
- Axis Maps' Thematic Cartography Guide.
- Spatial Reference.
- GDAL.org.
- Hands on with GDAL/OGR: a gentle introduction to command line GIS.
- Even Rouault's GDAL workshop.
- XKCD on Map Projections.

1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp (you are here)
3. Geodesy & Local Map Projections
4. Working with Satellite Data
5. Shaded Relief



Glaciers of the Everest region. NASA Earth Observatory image by Robert Simmon, using data from ASTER.

A Gentle Introduction to GDAL, Part 3: Geodesy & Local Map Projections

Robert Simmon · Follow

Published in Planet Stories

12 min read · Apr 17, 2017

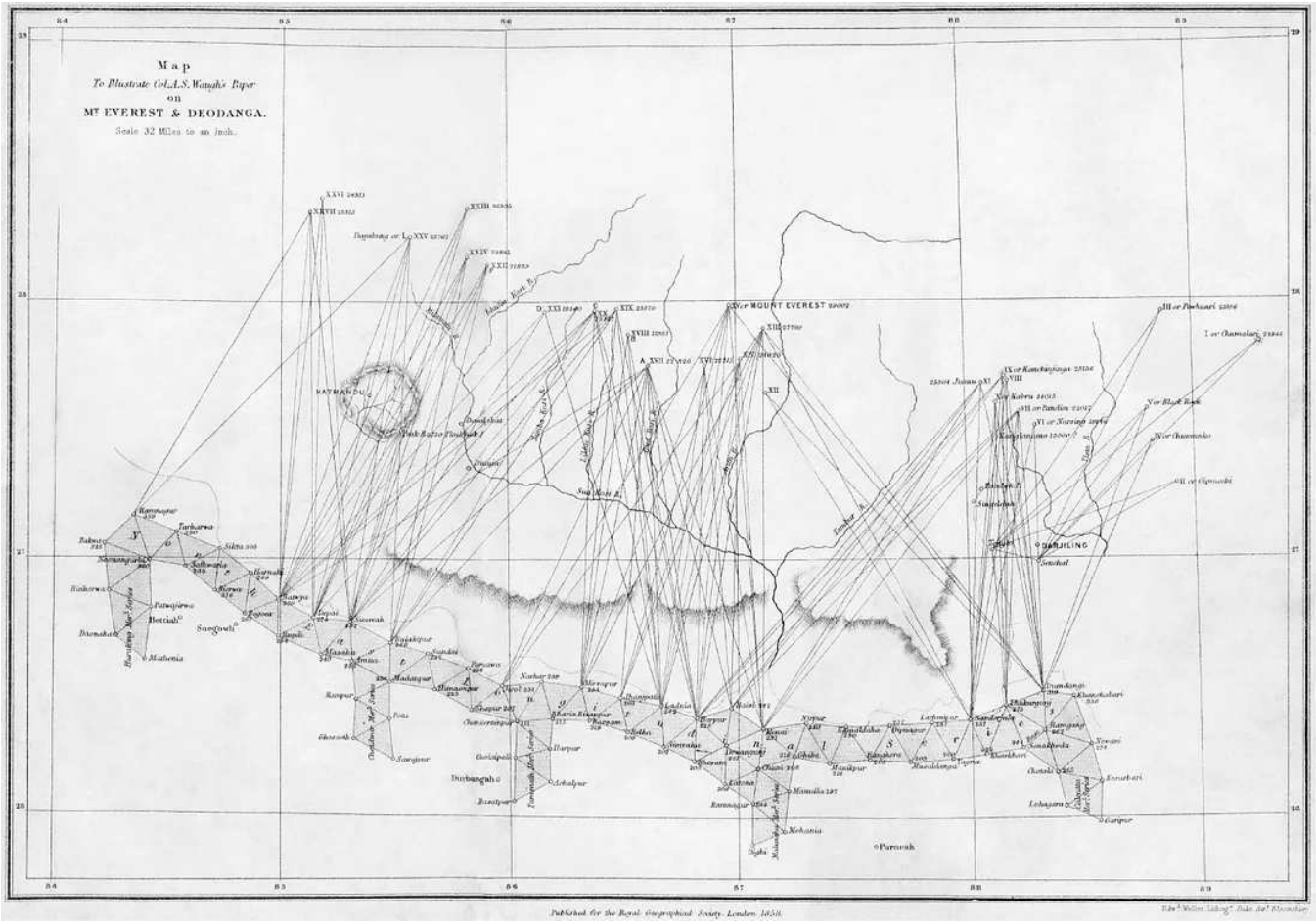
(If you're new to GDAL, you might want to start with Part 1, which covers installation, `gdalinfo`, & `gdal_translate`, or Part 2 Map projections & `gdalwarp`.)

In the spring of 1802, the British East India Company embarked on a detailed, 5-year-long survey of the Indian Subcontinent. The plan was to measure the arc of the meridian (length along a line of longitude—in this case 78˚ east) from Cape Comorin (the southern tip of India) to the Himalaya.

45 years and 3 directors later the Great Trigonometric Survey (as the endeavor became known) reached the glacier-capped mountains. About a decade after that, in 1956, Andrew Scott Waugh declared Peak XV (known by locals as Chomolungma and the rest of the world as Mount Everest) the world's highest. And in 1870 the first *Index Chart to the Great Trigonometrical Survey of India* was published.

1882 edition of the Index Chart to the Great Trigonometrical Survey of India, printed in 1885. Map courtesy Mountains of Central Asia Digital Dataset.

The Survey's primary objective wasn't to measure the altitude of the world's highest peaks (although they more-or-less succeeded at that) or make a map of India (ditto)—but to measure the size and shape of the Earth itself. It turns out that not only is the Earth not flat, but it's not exactly round, either. It's what's technically (and charmingly) known as an *oblate spheroid*—a ball with a bulge in the middle.



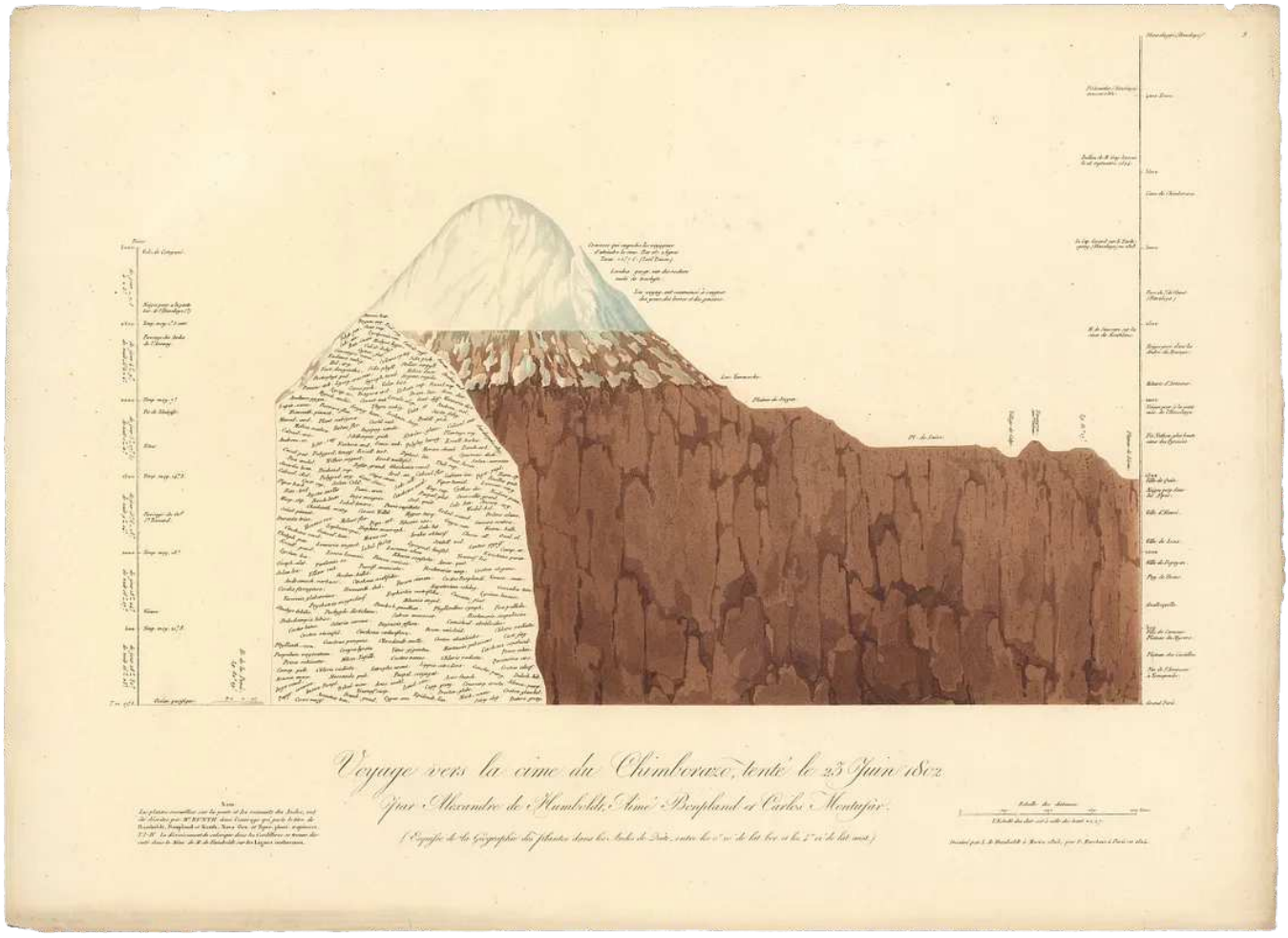
A map of the triangulations used to determine the height of Mount Everest and other Himalayan peaks. From "On Mounts Everest and Deodanga, Proceedings of the Royal Geographical Society of London, Vol. 2, №2 (1857–1858), pp. 102–115".

By taking precise measurements along one very long north-south line, and comparing that length to equally precise east-west measurements, it's possible to determine the shape of that squashed ball, a mathematical surface called a reference ellipsoid. And, in fact, the ellipsoid obtained by the Survey's second director—George Everest—remains in use in South Asia and is even incorporated in a handful of EPSG codes.

The main cause of the imperfect shape of the Earth is simply rotation—it stretches a bit along the equator. This asymmetry is handled perfectly well by an ellipsoid—but there are also local variations, caused by differences in density and thickness of the Earth's crust, and the water and ice piled on top of it. These variations change the overall shape of the Earth's gravity field. A *geoid* accounts for these local variations

in shape. (The geoid is sometimes described as mean sea level, but there are subtle differences which I don't quite understand.) The gravitational discrepancies are generally quite subtle, but still large enough to measurably deflect a weight hung on the end of a string. This is why the Great Trigonometric Survey took so many decades—to be useful, the measurements needed to be incredibly precise.

It's worth pointing out that shape in this context is the theoretical shape of the Earth if it was entirely covered in water—differences in elevation between mountains and valleys, plateaus and canyons, are defined relative to this surface.



Chimborazo Volcano, Ecuador. If surface elevation was defined relative to the center of the Earth rather than sea level, Chimborazo, which lies near the equator, would win by (more than) a mile. Drawing courtesy David Rumsey Historical Map Collection.

A *geodetic datum* is used to link a reference ellipsoid with precise coordinates for latitude and longitude, and there is a bewildering array of them ranging from continental (NAD27) to local (Old Hawaiian Datum). Each is optimized to minimize

errors—which can be several hundred meters—for particular locations.

Prior to the Space Age datums were defined in isolation, and could only be joined by carefully matching the measurements from adjacent surveys—which ranged from extremely difficult to downright impossible. For example, traditional surveying techniques rely on line of site—if you can't see it, you can't plot it. This means that the coordinate systems for far-flung islands, or separate continents, could never be precisely aligned. During the Cold War the U.S. Air Force bounced radio waves off planes to link far-flung datums, but these efforts were soon supplanted by satellites. Many modern geospatial datasets are defined in terms of the World Geodetic System 84 (WGS84), which you've seen any time you've used a GPS.

By now you may be wondering why I've gone into a long and somewhat technical digression — I certainly was frustrated by the extended discussion of such things during my (extremely limited) formal training in GIS. It turns out that the differences between ellipsoids, geoids, and datums don't matter so much at a global scale—but they *do* matter if you're trying to drill an oil well, survey a property line, or drop a missile on someone's head from the other side of the world. (A discouraging number of advances in cartography were developed for the military.) A working understanding of these concepts will help you understand some of the more esoteric aspects of manipulating maps and satellite data.

Crafting Local Maps

There are two fundamental ways to make a map of a specific region: start with a large dataset and cut out the bit you're interested in, or build the map from smaller pieces. I'll start with a large map, the high-res version of the Natural Earth raster dataset I used in part 2. If you're not up for the 300MB download, just use the smaller one and change the file names to match.)

Transverse Mercator map of India. Made with Natural Earth.

I could just use `gdal_translate` with `-projwin` option to crop out a latitude and longitude, but, as I've discussed, cylindrical equirectangular isn't a great projection

for global maps, and it's worse for most local and regional maps. Which begs the question—how to choose a better one?

Hacks borrow, artists steal.

If you're making a map of a place that's already been mapped by experts, just look up what they did and copy it! In this case, I just found a *National Geographic* wall map of India and read the projection off the description: Transverse Mercator. Professional cartographers will almost universally include projection and scale info on their maps. They also overlay maps with *graticules* (lines of latitude and longitude) so I estimated the proper extents from those. Here's the code:

```
gdalwarp -t_srs '+proj=tmerc +lat_0=0 +lon_0=84 +k=0.9996
+datum=WGS84 +units=m +no_defs ' -te 66 6 100 41 -te_srs EPSG:4326 -
ts 1400 0 -r bilinear NE1_HR_LC_SR_W.tif india_tmerc.tif
```

Most of this should look familiar. I've written out the target spatial reference system, `-t_srs`, since I couldn't find a ready-made EPSG. Transverse Mercator is specified by `+proj=tmerc` followed by a few more variables:

```
+lat_0=0 +lon_0=84 +k=0.9996 +datum=WGS84 +units=m +no_defs
```

`+lat_0=0` specifies the latitude of the origin (the Equator), `+lon_0=84` specifies the longitude of origin—the only meridian that will be vertical (84˚). `+k=0.9996` is a scaling factor that helps spread the distortion across the map. Set to 1, There's no distortion along the central meridian, but distortion increases out towards the edges. With a value less than 1, there are two meridians on either side of the center with no distortion, and a little bit of distortion at the center and a little bit at the edges, but limits total distortion. It's a compromise. I've defined the datum as WGS84 with `+datum=WGS84`, which matches the source dataset and prevents having to worry about datum switching, which can be painful. Finally `+units=m` defines the units (Transverse Mercator is a projected coordinate system so it needs linear units) and `+no_defs` prevents proj.4 from using defaults settings, which could theoretically

cause problems.

The other interesting part of the command is this:

```
-te 66 6 100 41 -te_srs EPSG:4326
```

This sets the *target extent* in units defined by the spatial reference system specified with `-te_srs`, our good friend EPSG:4326, or simple latitude and longitude. (This is one of the new features in GDAL 2.) If you set `-te` without `-te_srs` you need to figure out what the boundaries of your map should be in the target SRS, which is often meters that are specific to an origin point that can be mysterious. This is often hard (for me, at least), so I find it easier to define my boundaries in latitude and longitude. The final few commands I've shown before:

```
ts 1400 0 -r bilinear NE1_HR_LC_SR_W.tif india_tmerc.tif
```

These set the *target size* in pixels (by setting height to 0 GDAL will automatically figure out how tall the map should be), *resampling* method to bilinear in the names of the *input file* and *output file.*

But what happens if you're making a unique map, not of something that's been done a million times before like a country or a province? How do you pick a map projection then? With the fantastic Projection Wizard, by Bojan Šavrič.

The Projection Wizard interface.

All you have to do is drag a box around the area of interest, select the type of map you want; equal-area, conformal, or equidistant; then hit the "PROJ.4" link for the projection you want (there are options), which will spit out a text string to drop in `-t_srs`. So good.

```
+proj=eqdc +lat_1=38.02777777777778 +lat_2=38.47222222222222
+lon_0=-109.875
```

My only quibble is that the interface uses degrees, minutes, seconds, and GDAL uses decimal degrees. You may want to trim those to round numbers, and likewise for the extents. Like so:

```
gdalwarp -t_srs '+proj=eqdc +lat_1=38.025 +lat_2=38.470
+lon_0=-109.875' -te -110.5 37.75 -109.25 38.75 -te_srs EPSG:4326 -
ts 1400 0 -r bilinear NE1_HR_LC_SR_W.tif
```

```
NE1_HR_LC_SR_W_canyonlands_eqdc_1400.tif
```

Oops. Not very sharp, is it?

A map of the Canyonlands that doesn't have remotely enough resolution. Made with Natural Earth.

It's not that useful to make a large-scale (local) map with a small-scale (global) dataset. Looks like I need another data source. Conveniently, there's a high-res (1:100,000-scale) version of Natural Earth covering the United States. Not so

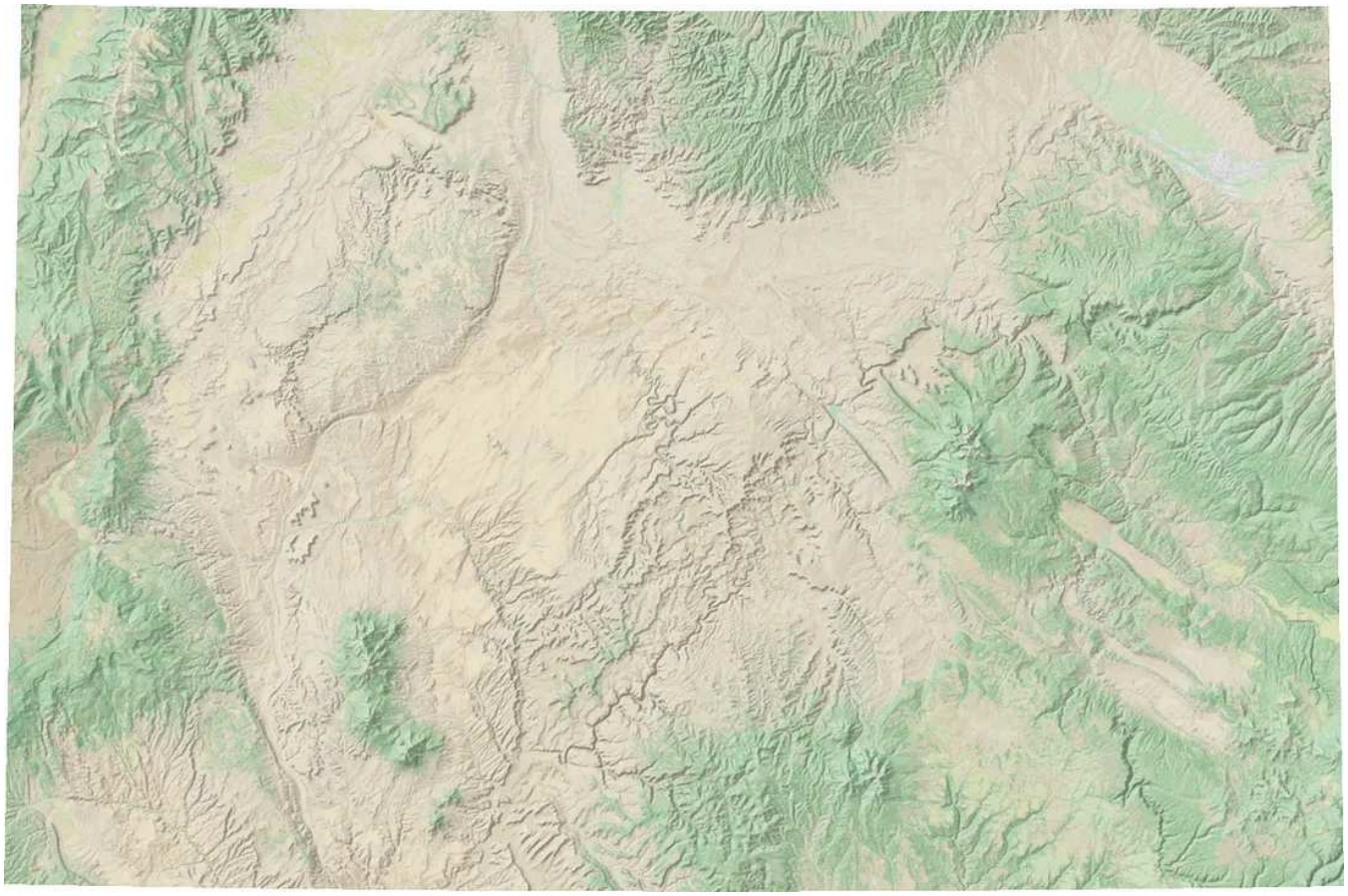
conveniently, it's really, *really* big—4.72 GB big, to be precise. To save everyone a long download I've cropped it to the Canyonlands and chopped it into pieces (download here), which, not coincidentally, also allows me to demonstrate `gdal_merge.py`.

`gdal_merge` is a helper utility written in Python—it's not a core part of GDAL. In fact, Frank says he wrote it as a demo (which is why the feature set may seem limited and syntax inconsistent) but people found it useful so it stuck around. And yes, it is useful. Unzip the download, navigate to the `natual_earth_100k_canyonlands` directory in your command line, and run the following command:

```
gdal_merge.py -o canyonlands_merged.tif *.tif
```

That's it. `gdal_merge.py` invokes the script, `-o canyonlands_merged.tif` specifies the name of the output file, and `*.tif` is a wildcard that opens up every file in the directory ending with .tif. What `gdal_merge` *doesn't* do is any type of reprojection (but it can crop with `-ul_lr` and resize with `-ps`). So the output file is Web Mercator, just like the input files. Equidistant conic is more appropriate for this region, so use the same `gdalwarp` command as before (with one change) to reproject the data:

```
gdalwarp -t_srs '+proj=eqdc +lat_1=38.025 +lat_2=38.470
+lon_0=-109.875' -ts 1400 0 -r bilinear -dstalpha
canyonlands_merged.tif NE1_HR_LC_SR_W_canyonlands_ne_eqdc_1400.tif
```



Map of the southeastern Utah, from the 100-meter resolution Natural Earth of the United States, equidistant conic projection.

I've omitted the `te` and `te_srs` options— `gdalwarp` is smart and matches the extents of the output file to the input file, which can be convenient, and results in the subtle cuve along the edges of the map above. To match the boundary of the original (blurry) map, use:

```
gdalwarp -t_srs '+proj=eqdc +lat_1=38.025 +lat_2=38.470
+lon_0=-109.875' -te -110.5 37.75 -109.25 38.75 -te_srs EPSG:4326 -
ts 1400 0 -r bilinear canyonlands_merged.tif
canyonlands_ne_eqdc_te_1400.tif
```

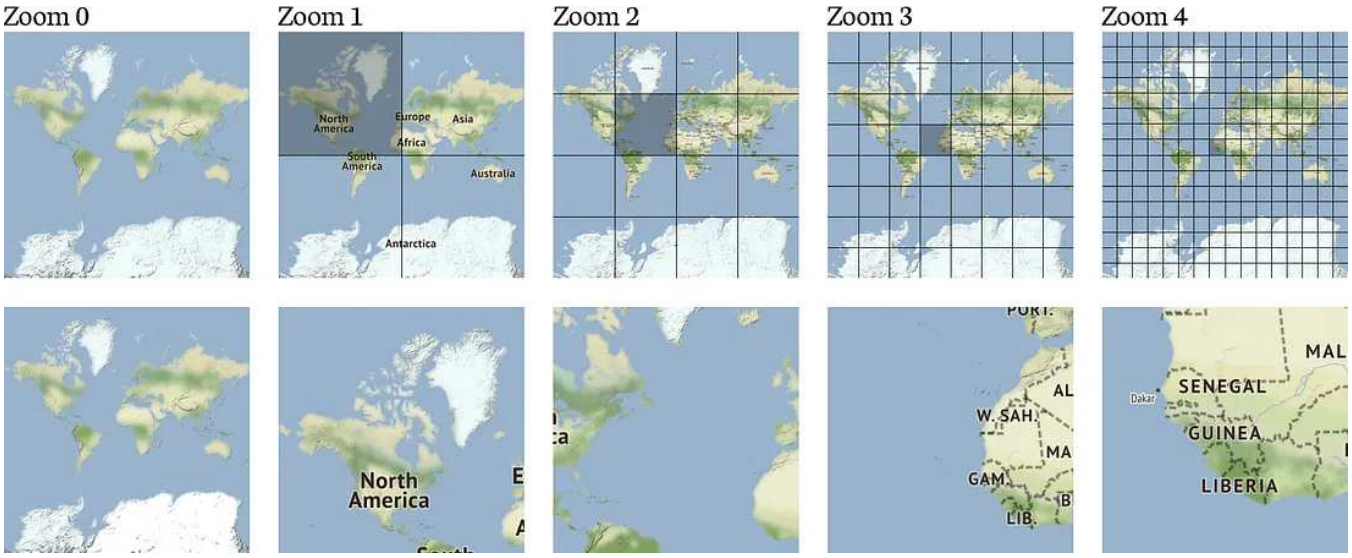
Map of the Canyonlands, from the 100-meter resolution Natural Earth of the United States, equidistant conic projection.

Looking carefully, it's evident that even this map isn't *quite* detailed enough to display at this size—it needs a slightly higher-resoluton data source. In the not-so-distant past, you'd probably be limited to 7.5 minute USGS topographic maps (or their international equivalents), or custom made maps for specific locations (like U.S. national park maps). But in the past decade or so there's been an explosion of

mapping on the web, both commercial (Google Maps, MapBox) and open-source (Open Street Map). Typically they're limited to display in a browser, or on a mobile device (that's what these maps are made for, after all).

Georeferencing Web Tiles

The maps you see on the web are composed of many different tiles, each 256- by 256-pixels, scaled to fit the display size. They're not individually georeferenced, but each fits into a system that allows their relative scale and placement to be derived.



A typical web-tiling scheme, from zoom level 0 to 4. Terrain tiles ©Stamen Design, cc by s.a. 3.0.

The most common web mapping tiling scheme consists of a a zoom level, an x coordinate, and a y coordinate. Zoom levels typically range from 0 (156,412 meters per pixel, global) to about 19 (about 0.3 meters per pixel—as detailed as the highest-resolution unclassified satellite data). At higher zoom levels, the map is subdivided into two equal vertical and horizontal slices. Every slice is given an x and y coordinate, starting with 0 in the upper-left-hand corner. Mapbox has an excellent and much more detailed description of how web maps work.

This consistent scheme provides a mechanism for GDAL to decode, and the ability to convert web tiles into a georeferenced file. The code to generate a map of the Canyonlands is deceptively simple:

```
gdal_translate -projwin -110.75 39 -109 37.5 -projwin_srs EPSG:4326
-outsize 4096 0 frmt_wms_stamen_terrain_tms.xml
```

```
canyonlands_terrain_4096.tif
```

It *looks* just like a normal use of `gdal_translate`, but instead of pointing to an image, it's pointing to an XML file:

```
<GDAL_WMS>
<Service name="TMS">
<ServerUrl>http://b.tile.stamen.com/terrain-background/${z}/${x}/${y}.jpg</ServerUrl>
</Service>
<DataWindow>
<UpperLeftX>-20037508.34</UpperLeftX>
<UpperLeftY>20037508.34</UpperLeftY>
<LowerRightX>20037508.34</LowerRightX>
<LowerRightY>-20037508.34</LowerRightY>
<TileLevel>18</TileLevel>
<TileCountX>1</TileCountX>
<TileCountY>1</TileCountY>
<YOrigin>top</YOrigin>
</DataWindow>
<Projection>EPSG:3857</Projection>
<BlockSizeX>256</BlockSizeX>
<BlockSizeY>256</BlockSizeY>
<BandsCount>3</BandsCount>
<ZeroBlockHttpCodes>302</ZeroBlockHttpCodes>
<Cache />
</GDAL_WMS>
```

A full description of the process is in the GDAL Web Map Services documentation, but the basic idea is to open up the XML file in a text editor and point the

`<ServerUrl>` line at the map server hosting the tiles you want to download, stitch, and georeference. (I also added a little buffer around the edges with `-projwin -110.75 39 -109 37.5` so my subsequent reprojection step wouldn't be cut off at the edges.)

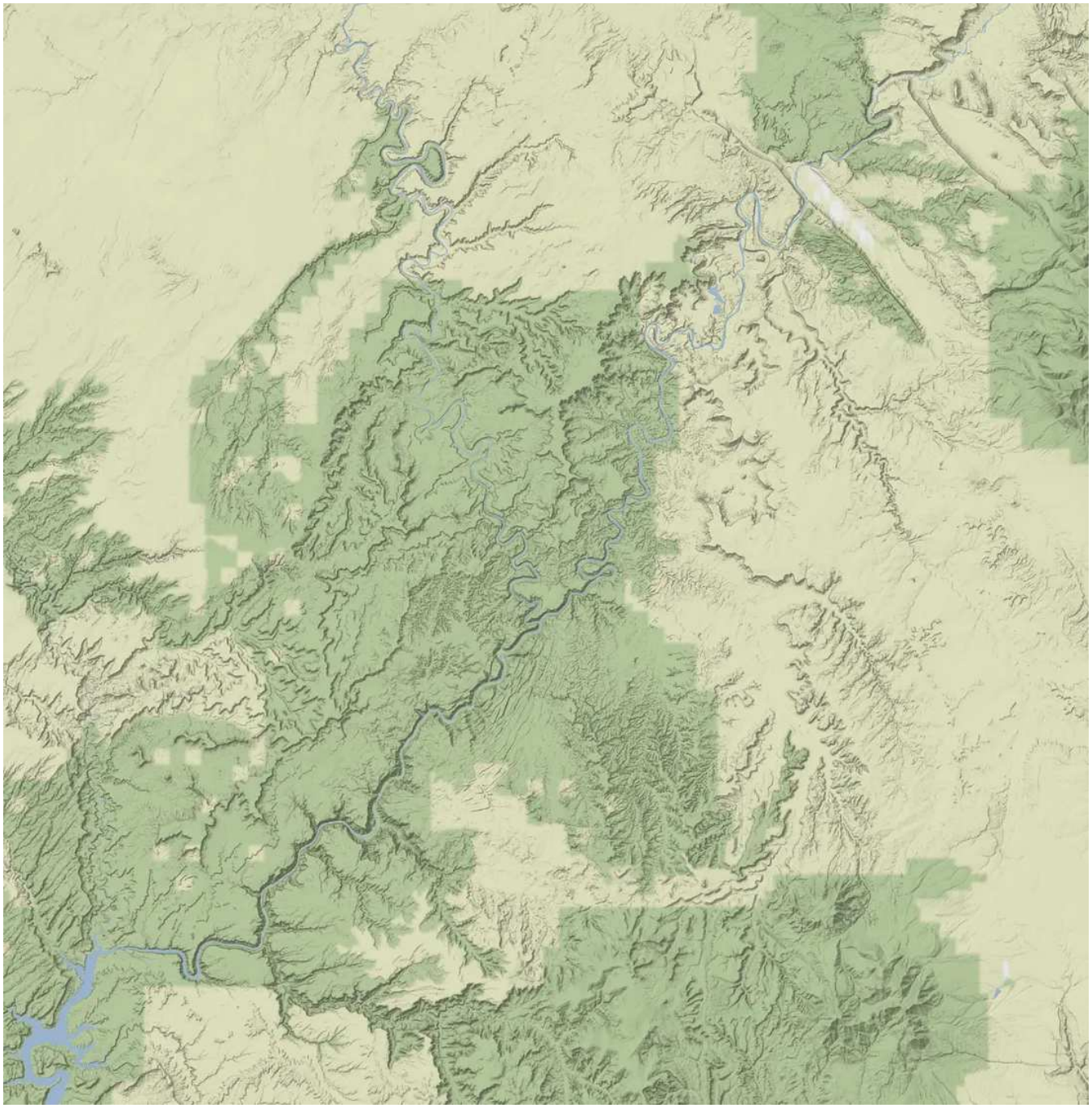
There are several examples using different types of map servers in the documentation, but if you are viewing a slippy map you can often right-click (command-click on a mac) on the map and it will give you the option to open the image in a new tab or window — that will give you the URL to paste into `<ServerUrl>`. Replace `/terrain/` with `/watercolor/` to generate stylized maps. You can access Open Street Map tiles by changing the URL to `http://tile.openstreetmap.org/` and file format from `.jpg` to `.png`:

```
<ServerUrl>http://tile.openstreetmap.org/${z}/${x}/${y}.png</
ServerUrl>
```

BTW—Stamen's map tiles are built on Open Street Map and licensed under creative commons—make sure you follow any licensing requirements for the data you use, and please don't abuse your access by downloading tens of thousands of tiles.

The final step is to convert the map from Web Mercator to equidistant conic with what should be familiar `gdalwarp` command (although I admit to need to have the documentation open more often then not to make sure I get the syntax right):

```
gdalwarp -t_srs '+proj=eqdc +lat_1=38.025 +lat_2=38.470
+lon_0=-109.875' -te -110.5 37.75 -109.25 38.75 -te_srs EPSG:4326 -
ts 1400 0 -r bilinear canyonlands_terrain_4096.tif
canyonlands_terrain_eqdc_1400.tif
```



Map of the Canyonlands, using terrain tiles ©Stamen Design, cc by s.a. 3.0. Notice the presence of fine details missing in the Natural Earth map above. In fact, one could argue there is an overwhelming amount of detail. One important part of making maps is finding the balance between precision and readability.

With these tools the wide variety of government and open-source data available, I hope you'll be able to get started making your own maps. But what if you want to go beyond idealized base maps, and explore a photo-realistic view of the world, or show change over time? That will be the topic of my next post—processing satellite data, including Planet, Landsat, and Sentinel, with GDAL.

Further Reading

- The Great Trigonometric Survey of India Geospatial World
- What do the terms geoid, ellipsoid, spheroid and datum mean, and how are they related? ESRI
- Natural Earth
- Map Projections Used by the US Geological Survey (PDF) USGS
- Projection Wizard Bojan Šavrič
- Maps Stamen Design
- How Web Maps Work Mapbox

1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp
3. Geodesy & Local Map Projections (you are here)
4. Working with Satellite Data
5. Shaded Relief
6. Visualizing Data
7. Transforming Data

Special thanks to Frank Warmerdam and Tim Schaub for proofreading and technical support.

Maps GIS Everest Gdal Web Mapping

Intensely-colored wildflowers covered the Carrizo Plain in the spring of 2017. Photograph ©2017 Steve Corey. cc by-nc-nd 2.0.

A Gentle Introduction to GDAL Part 4: Working with Satellite Data

Robert Simmon · Follow
Published in Planet Stories
11 min read · Apr 22, 2017

Maps are great for plotting a route, finding yourself when you're lost, exploring a distant land, or discovering relationships between areas, but they're more-or-less by definition a static, idealized depiction of place. What happens if you want to see the surface of the Earth in near-real time, watch the seasons change, or study a long-term trend?

Launch a satellite! Or, failing that, download some data from someone who has.

A very wet winter brought an incredible blanket of flowers to California in the spring of 2017. These Planet RapidEye images show how the flowers transformed Carrizo Plain National Monument. (Lindsay Hoshaw of KQED found these data in the Planet archive & wrote about the super bloom — thanks!) ©2017 Planet Labs Inc., cc-by-sa 4.0.

The images above, from Planet's RapidEye constellation, demonstrate one of satellite data's greatest strengths: observing change. Orbiting robots can monitor the Earth 24/7 (although this depends on the orbit, there are many options), 365.25 days a year. They also have super-powers, like being able to eat sunlight, infra-vision, perfect memory, and virtual time-travel (some weather satellite datasets go back to the early '60s.) This allows unparalleled study of seasonal and long-term change.

On the flip side, they can also be inscrutable, like an Alan Moore superhero. The data are rarely packaged neatly, and even when they are it can take some work to make a usable image (or set things up for processing by a computer). Which is where GDAL comes in. Again.

A typical workflow to create an image from raw satellite data would be:

1. Download data.
2. Re-order or assemble bands into the desired order (red, green, blue; or near-infrared, red, green; etc.)
3. Increase the resolution with pan-sharpening, if desired.
4. Contrast-stretch and color-correct the imagery, either algorithmically or by hand.
5. Restore georeferencing information, if necessary.
6. Crop, re-project, and re-size image to merge with other data.

Here's how to do it.

Using gdal_translate to Re-order Bands

Let's start with the data that went into the March 31 image of the Carrizo Plain. Download a single scene here, or sign up for a Planet Explorer account. (It's free, and you'll be able to access PlanetScope and RapidEye data from California. Navigate to Carrizo Plain National Monument (about -119.725, 35.120) and download the *RapidEye ortho tile* with this ID: `20170331_190720_1155205_RapidEye-3` and its neighbors.) Now open the file `1155205_2017-03-31_RE3_3A.tif` with your TIFF viewer of choice.

Blue, green, red RapidEye image of the Carrizo Plain. ©2017 Planet Labs Inc., cc-by-sa 4.0.

Oh. That doesn't look right.

Fire up `gdalinfo` to figure out why.

```
gdalinfo 1155205_2017-03-31_RE3_3A.tif
```

Which gives a block 'o text, ending with:

```
Band 1 Block=5000x1 Type=UInt16, ColorInterp=Red
NoData Value=0
Band 2 Block=5000x1 Type=UInt16, ColorInterp=Green
NoData Value=0
Band 3 Block=5000x1 Type=UInt16, ColorInterp=Blue
NoData Value=0
Band 4 Block=5000x1 Type=UInt16, ColorInterp=Undefined
NoData Value=0
Band 5 Block=5000x1 Type=UInt16, ColorInterp=Undefined
NoData Value=0
```

There are **5** bands! RapidEye detects light in 5 separate wavelengths: blue, green, red, red-edge, and infrared (full image specs are in the Planet Imagery Product Specification (PDF)). Each is assigned as a *band* in the image. We can see the first 3 (the additive color primaries!), kinda see red-edge, and decidedly can't see near infrared. Read Why is that Forest Red and that Cloud Blue? (which I helped create in my previous gig at NASA) to learn more about what these extra wavelengths (and more) are for.

You may have noticed I listed the bands as blue, green, red... That's because they're ordered from short (blue) to long (infrared) wavelengths. It's just a convension in the remote sensing community, although there are exceptions (*cough* Landsat 4, 5, & 7 *cough*). They really shouldn't be displayed as an RGB image at all, but 5 separate grayscale channels. To reorder the bands and view the image correctly, run `gdal_translate` with a few new options:

```
gdal_translate 1155205_2017-03-31_RE3_3A.tif
1155205_2017-03-31_RE3_3A_rgb.tif -b 3 -b 2 -b 1 -co
COMPRESS=DEFLATE -co PHOTOMETRIC=RGB
```

Adding the option `-b 3 -b 2 -b 1` assigns band 3 to blue, band 2 to green, and band 3 to red. `-co COMPRESS=DEFLATE` enables an efficient lossless compression scheme to keep the file size from getting outrageous, and `-co PHOTOMETRIC=RGB` ensures any image viewer will display the bands as red, green, blue (instead of an alpha channel or something).

Unstretched red, green, blue RapidEye image. ©2017 Planet Labs Inc., cc-by-sa 4.0.

Better! (No really, it's better.) But still dark. Time for `gdalinfo` again, but this time with `-mm` to compute image statistics and see *why* it's so dark.

```
gdalinfo -mm 1155205_2017-03-31_RE3_3A_rgb.tif
```

Which should spit out:

```
Band 1 Block=5000x1 Type=UInt16, ColorInterp=Blue
Computed Min/Max=1422.000,41645.000
NoData Value=0
Band 2 Block=5000x1 Type=UInt16, ColorInterp=Green
Computed Min/Max=2108.000,49122.000
NoData Value=0
Band 3 Block=5000x1 Type=UInt16, ColorInterp=Red
Computed Min/Max=3482.000,49572.000
NoData Value=0
```

There's a few important things in here: `Type=UInt16` and the `min/max` for each band. `Type=UInt16` means the data is in unsigned 16-bit integer format—twice as many bits, but 256 times the information as a normal 8-bit image channel. Theres 65,536 possible values in each band. This gives a lot of flexibility in image processing at the expense of a little bit of storage. The other value of the interest is the `Computed Min/Max` for each channel which are 14,22–41,645 for blue, 2,108–49,122 for green, and 3,482–49,572 for red. That's not bad, considering the available range in a 16-bit file is from 0–65,535. So what's wrong, and how can the image be made better?

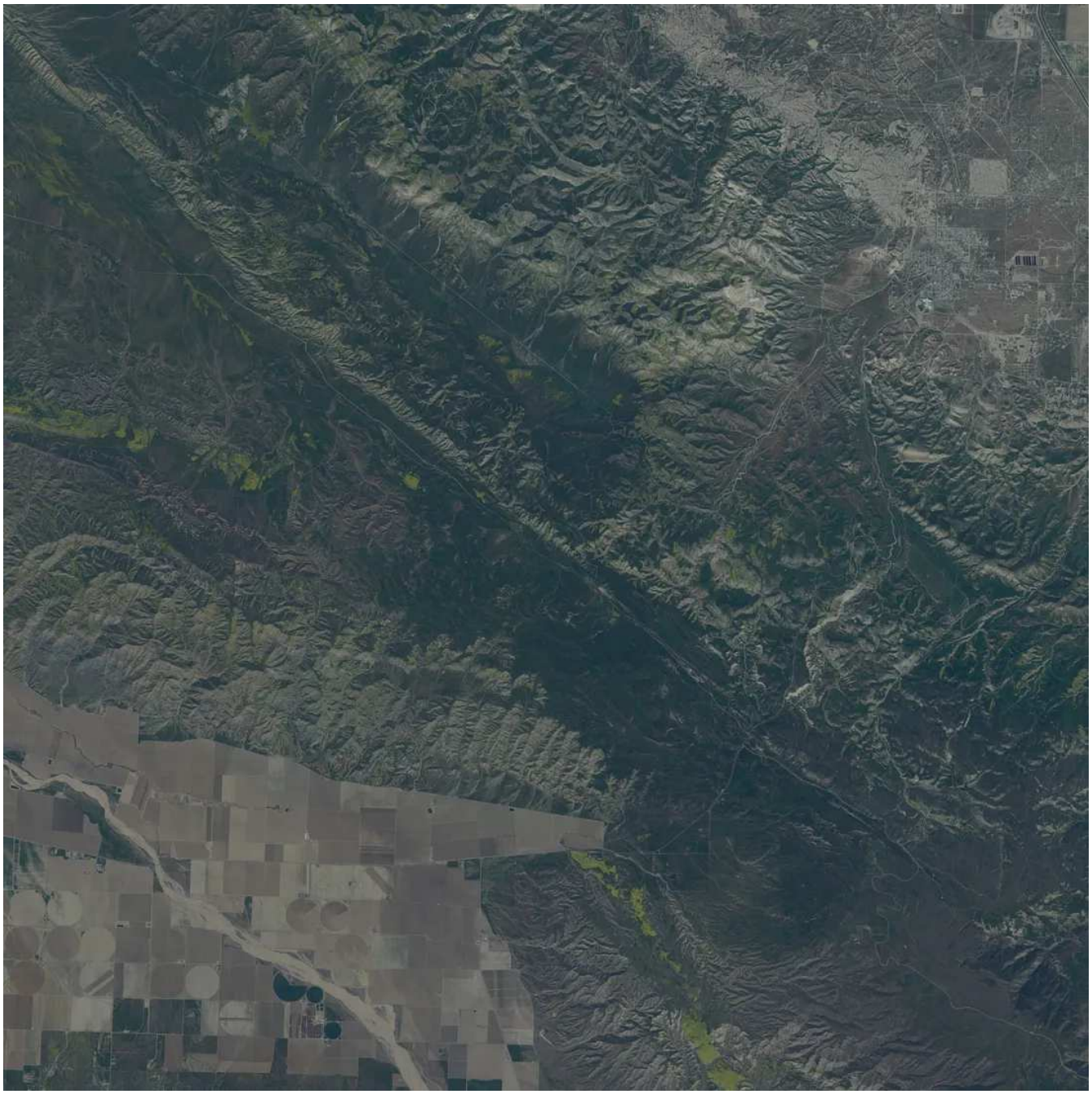
Algorithmic Image Enhancement

Since our eyes sense light proportionally, the data still need to be stretched to compensate, despite filling most of the available range. Like so:

```
gdal_translate 1155205_2017-03-31_RE3_3A_rgb.tif 1155205_2017-03-
31_RE3_3A_rgb_scaled.tif -scale 1422 49572 0 65535 -exponent 0.5 -co
COMPRESS=DEFLATE -co PHOTOMETRIC=RGB
```

This does two important things. `-scale 1422 49572 0 65535` stretches each band equally from from a range of 1,422–49,572 (first pair of numbers) to a range of 0–

65,535 (second pair of numbers). I could have scaled each band separately to its extents (which is an extremely common image processing technique) but that would likely engender a hue shift. Equal scaling leaves things looking more natural. `-exponent 0.5` raises each band to the power of ½—the square root. It's a quick and dirty way to get a preview image.



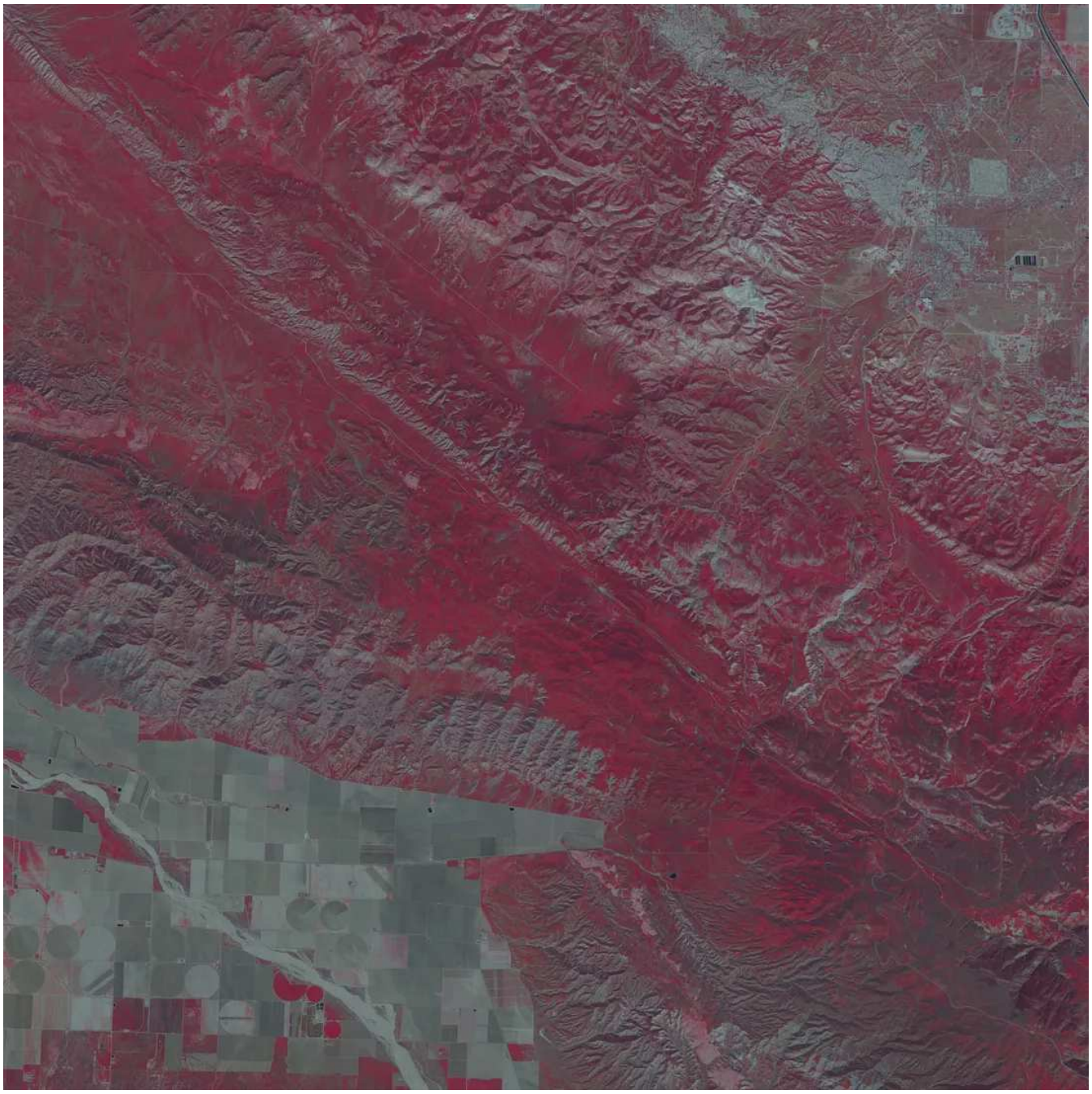
Histogram stretched and square-root scaled red, green, blue RapidEye image. ©2017 Planet Labs Inc., cc-by-sa 4.0.

As I said, this is quick and dirty, and is going to make every scene look different.

Color correction of satellite imagery is a *vast* topic. I've written two guides, one with Landsat 8 and one with PlanetScope data—they're the process for both works fairly well for RapidEye. Tom Patterson of the National Park Service and Mapbox have both written guides that I recommend, too.

That's an RGB image—how is the infrared band used? By convention the longest wavelength (furthest in the infrared) replaces the red band, while the other (shorter wavelength) bands are pushes towards blue. A "standard" false-color infrared image is near-infrared=red, red=green, and green=blue. There's only 3 bands to play with, so blue goes away entirely. Make a false-color image like this:

```
gdal_translate  1155205_2017-03-31_RE3_3A.tif
1155205_2017-03-31_RE3_3A_nir.tif -b 5 -b 3 -b 2 -scale 1051 49122 0
65535 -exponent 0.5 -co COMPRESS=DEFLATE -co PHOTOMETRIC=RGB
```



Histogram stretched and square-root scaled near-infrared, red, green RapidEye image. ©2017 Planet Labs Inc., cc-by-sa 4.0.

I've done the band re-ordering & scaling with one line of code (having run `gdalinfo -mm` on the original 5-band file to get the range).

Combining Bands with gdal_merge.py

So far so good. These images show the Carrizo Plain in the relatively recent past (at least at the time I'm writing this)—what did the region look like last year, after a

merely rainy (not biblically wet) winter?

Digging through the Open California RapidEye archive you'll find—nothing. Well, not exactly nothing. Clouds. Not everywhere is as cloudy as Borneo, but clouds are still surprisingly common, with about 2/3 of the Earth's surface covered at any given time. Fortunately, the RapidEye satellites aren't the only ones in the sky, and Landsat 8 collected a nice, cloud-free scene of the area just over a year before, on March 25, 2016.

There's roughly a dozen different Landsat viewers out there (and you can add Planet Explorer to the list), but the most straighforward for retrieving a single scene might be the AWS Landsat archive (if you want to use Earth Explorer, the official USGS access point, I wrote a guide back when it was the only option). Navigate to LC80420362016085LGN00 to download the data from last spring (learn how to decode the scene IDs here).

Wow. That file list is a mess. Instead of having multiple bands in a single file, each band is stored in its own TIFF, and there's a bunch of other files with cryptic extensions like "IMD" and "OVR". Just ignore them, and download the TIFFs for band 2 (blue), band 3 (green), band 4 (red) and band 8 (panchromatic).

Combining the separate bands is just as straightforward as re-ordering bands, but uses `gdal_merge.py` instead of `gdal_translate`.

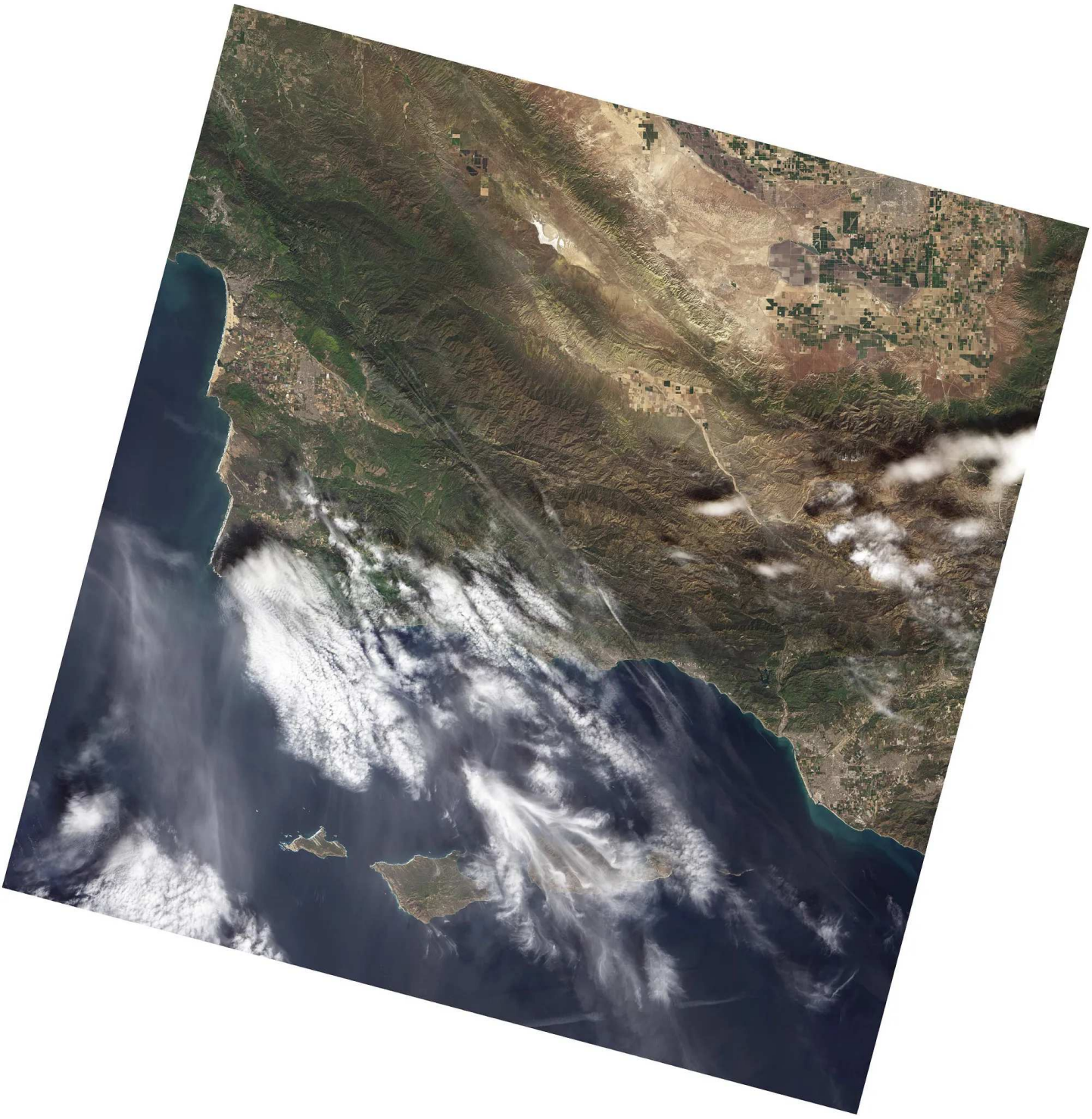
```
gdal_merge.py -o carrizo-20160325-oli-rgb.tif -separate
LC80420362016085LGN00_B4.TIF LC80420362016085LGN00_B3.TIF
LC80420362016085LGN00_B2.TIF -co PHOTOMETRIC=RGB -co
COMPRESS=DEFLATE
```

The flag `-separate` followed by the filenames for the bands we want `LC80420362016085LGN00_B4.TIF LC80420362016085LGN00_B3.TIF LC80420362016085LGN00_B2.TIF` stack the channels in the order they're written. The rest of the code I've described before: `gdal_merge.py` calls the script and `-o carrizo-20160325-oli-rgb.tif` creates the output file (be careful: It will overwrite existing files!) The command options `-co PHOTOMETRIC=RGB` and `-co COMPRESS=DEFLATE`

make sure TIFF readers interpret the colors correctly and that the file is as small as possible without throwing away any data.

By the way, GDAL is *very* flexible about the order of commands—it usually doesn't matter. I try to be consistent just to keep myself sane, but don't always manage it (either consistency or sanity.) The order of the bands after `-separate` does matter, however. For most conventional band combinations, make sure longer wavelengths are listed first.

Here's the output file after color correction:



True-color Landsat 8 image collected on March 25, 2016. Data courtesy NASA/USGS Landsat.

Landsat scenes cover a *much* wider area than a RapidEye tile. You can see all the way from Vandenberg Air Force Base (where Landsat 8 was launched!) to Oxnard, California. But Landsat has a lower resolution—30 meters per pixel instead of 5 meters per pixel. This is a common tradeoff in satellite imaging: area covered vs.

resolution. Landsat 8 and several other satellites (the SPOT series, Digital Globe's constellation) carry a special band, the *panchromatic* band, to get extra resolution from their sensors.

Pan sharpening essentially swaps the luminance information in a low resolution full-color image with a high resolution grayscale image. Since our eyes are more sensitive to changes in luminance than changes in color, a pan-sharpened image is almost as good as a full resolution multi-spectral (the fancy term for color) image. Here's how you apply pan sharpening with gdal_pansharpen.py, available in GDAL 2.1 and later (the writing of this tutorial was in *no way* affected by my UNIX environment's insistence that python try to import GDAL 1.11):

```
gdal_pansharpen.py LC80420362016085LGN00_B8.TIF carrizo-20160325-
oli-rgb.tif carrizo-20160325-oli-pan.tif -r bilinear -co
COMPRESS=DEFLATE -co PHOTOMETRIC=RGB
```

Like most GDAL commands, it may be intimidating at first but is fairly straightforward once you get the hang of it. The script requires the input and output files to be in a specific order: *panchromatic band*, followed by the *multispectral image*, ending with the *output filename*.

Pan-sharpening a Landsat image bumps the resolution from 30 meters per pixel (left) to 15 meters per pixel (right). Images based on data courtesy NASA/USGS Landsat.

It's also possible to generate a pan-sharpened image from multiple separate bands, instead of a pre-combined file, in which case the command would look like:

```
gdal_pansharpen.py LC80420362016085LGN00_B8.TIF
LC80420362016085LGN00_B4.TIF LC80420362016085LGN00_B3.TIF
LC80420362016085LGN00_B2.TIF carrizo-20160325-oli-pan.tif -r
bilinear -co COMPRESS=DEFLATE -co PHOTOMETRIC=RGB
```

Longer to type, but ultimately fewer steps.

Unlike many other GDAL scripts, `gdal_pandsharpen` uses a smoothing resampling algorithm by default (cubic), but I often use `-r bilinear` instead because it creates images suitable for additional custom sharpening.

Pansharpening works best when the panchrimatic band spans similar wavelengths to the multispectral bands. Some older satellites like Landsat 7 and Ikonos had a lot of infrared in the panchromatic band, so they always looked a little funny. If you're working with these data try out the somewhat esoteric -b (*bands*) and -w (*weights*) options. Landsat 8 and most high-resolution satellites have panchromatic data purely in the visible spectrum, and look great out of the box.

Now that I have images from similar dates in two different years, how do I compare them? More GDAL magic, of course. That's what you're here for, right?

The Landsat scene and RapidEye tile have both different extents (the area covered by each image) and resolutions, and both need to be matched. I'll start with the extents, using `gdaltindex` to create a *shape file*—a file format that stores vector information, in this case the corner points of the RapidEye image.

```
gdaltindex 1155205_2017-03-31_RE3_3A_extent.shp
1155205_2017-03-31_RE3_3A.tif
```

Once there's a shapefile that defines the desired shape use `gdalwarp` with the `-cutline` option to crop the pan-sharpened Landsat file and `-tr 5 5` (*target resolution* in the units defined by the coordinates system of the file (meters in the Universal Transverse Mercator projection used by both Landsat and RapidEye)) to resize to match RapidEye's resolution:

```
gdalwarp -tr 5 5 -cutline 1155205_2017-03-31_RE3_3A_extent.shp -
crop_to_cutline carrizo-20160325-oli-pan-8bit.tif carrizo-20160325-
oli-pan-crop-5m.tif -co COMPRESS=LZW -co PHOTOMETRIC=RGB
```

This creates a perfectly matched pair of images.

Although there were some flowers in the spring of 2016, they can't hold a candle to 2017's bumper crop. Images based on data courtesy NASA/USGS LAndsat (left) and ©2017 Planet Labs Inc., cc-by-sa 4.0.

Last but not least, here's a trick to add georeferencing information into a file that doesn't have any. This is especially useful if you process images in Photoshop (without Avenza's Geographic Imager, which I generally like) or other photo editors that don't natively support GeoTIFF. Just remember to keep the original image size. To strip the headers, open one of the GeoTIFFs in Photshop or another image viewer and re-save it. (Or download this color-corrected copy of the March 25, 2016, Landsat scene.)

First, you'll need to install a helper python script that's not included in many (most?) default GDAL installations, `gdalcopyproj.py` . The script is hosted on Github, and should be downloaded into the directory that your command line environment stores programs. In my case: `/Users/myusername/miniconda2/bin/` . To make it run navigate to that directory and type: `chmod +x gdalcopyproj.py` .

To restore georeferencing run (make sure to use the right file name if you created your own TIFF):

```
gdalcopyproj.py carrizo-20160325-oli-rgb.tif carrizo-20160325-oli-rgb-corrected-nogeo.tif
```

This copies the information from `carrizo-20160325-oli-rgb.tif` into `carrizo-20160325-oli-rgb-corrected-nogeo.tif`. Confirm with `gdalinfo carrizo-20160325-oli-rgb-corrected-nogeo.tif`, which should print a bunch of text listing the coordinate system, etc. that looks similar to what I showed way back in part 1. Maybe rename the file from -nogeo.tif to -geo.tif, just to keep track.

Hopefully this is enough info for you to get started working with satellite imagery. There's a large (and growing!) collection of data from free and commercial sources, of ever-increasing accessibility and quality.

My next post will show some examples of using GDAL to translate and crop vector data with `ogr2ogr`, and some data processing techniques. with `gdal_calc`.

(Err, things didn't quite go as planned so the next post ended up being on shaded relief. I do intend to cover both vectors and data visualization, hopefully sooner rather than later.)

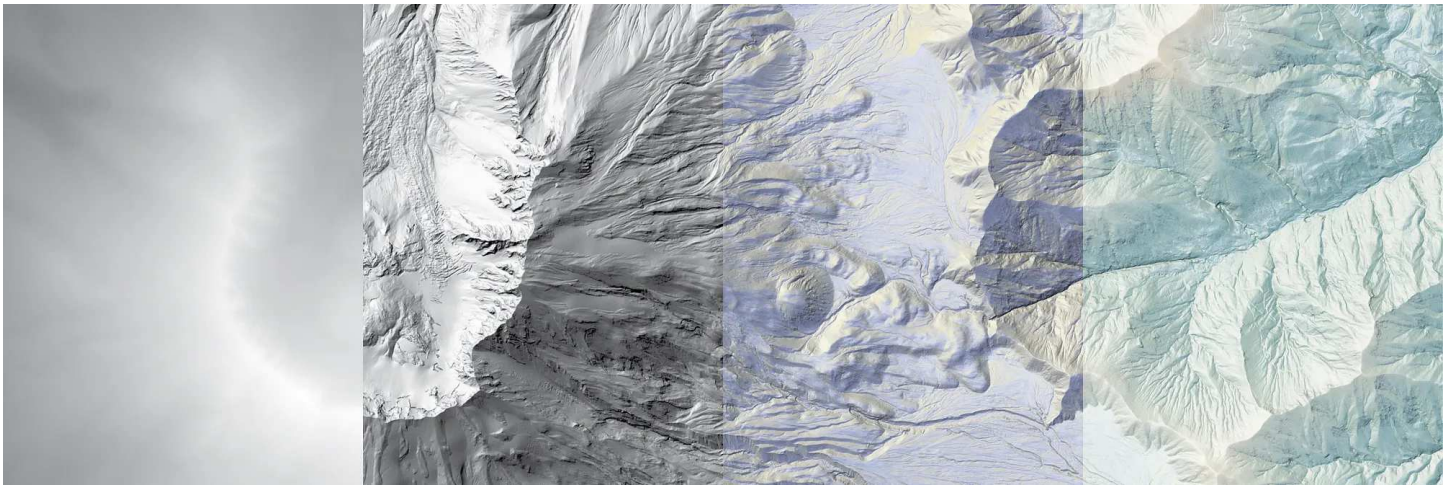
1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp
3. Geodesy & Local Map Projections
4. Working with Satellite Data (you are here)
5. Shaded Relief
6. Visualizing Data
7. Transforming Data

A Gentle Introduction to GDAL Part 5: Shaded Relief

Robert Simmon · Follow

17 min read · Sep 10, 2023

In my previous posts on GDAL (written more than five years ago!) I covered how to open and interpret maps and images with embedded geographic information; how to transform maps from one projection to another; some of the complexities introduced working with highly detailed maps; and how to read and manipulate satellite imagery. This post and the next one will cover using GDAL for visualizing other types of data: measurements like elevation, cloud cover, city lights, and vegetation.



Maps of Mount St. Helens generated from a single digital elevation model with GDAL. From left to right: grayscale elevation, hillshade, hillshade + aspect, and color elevation + blended hillshades + aspect + roughness. Images derived from the USGS National Map.

I'll start with elevation, showing off some of the specialized tools GDAL has for rendering shaded relief maps, like multidirectional shading, the ability to calculate the direction a slope faces, and the ability to color-code by altitude (a function that can be appropriated for other types of data, as well).

But first, a slight digression. In the first part of this series I recommended a now-outdated method for installing GDAL. These days (late summer, 2023) I think the best way to install GDAL is through Conda. If you're not familiar with it, Conda is a tool that helps one set up and maintain a programming environment and all the myriad libraries (like GDAL) that perform specific tasks. Installing Conda is beyond the scope of this post, but it shouldn't be too complicated if you're familiar with the command line on Windows, MacOS, or LINUX. If you're just getting started I'd recommend the regular installation (which comes with lots of bundled libraries) and installing GDAL from conda-forge.

With that out of the way, go ahead and download an elevation dataset. If you don't have one of your own handy, the following examples were all created from a high resolution digital elevation model of Mount St. Helens in Washington state. (That *should* be a direct link, if it doesn't work and you didn't have anything in mind you can find & download data from the USGS National Map, the Copernicus Land Monitoring Service, the ALOS Global Digital Surface Model, or a similar archive.) A digital elevation model (DEM) is a technical name for a file containing topographic data. You might also run into the terms digital surface model (DSM), which is the elevation of the surface as seen from above, including things like tops of trees and buildings; and digital terrain model (DTM) which is the elevation of the bare earth. Any of these variations will allow you to make a hill shaded representation of the topography, which is a great foundation to build a map on.

If you grabbed the Mount St. Helens data, you'll have a file called `USGS_1M_10_x56y512_WA_FEMAHQ_2018_D18.tif` which I renamed `mount_st_helens_USGS_1m_dem.tif` for my own sanity.

Once you have a dataset, navigate to the directory where you have it stored in a command line window, make sure you have a Conda environment with GDAL

running, and use `gdalinfo` to take a look at the metadata:

```
gdalinfo mount_st_helens_USGS_1m_dem.tif
```

This command will display metadata for the file, including details like the map projection, resolution, number of bands, etc.

```
Driver: GTiff/GeoTIFF
Files: mount_st_helens_USGS_1m_dem.tif
Size is 10012, 10012
Coordinate System is:
PROJCRS["NAD83 / UTM zone 10N",
    BASEGEOGCRS["NAD83",
        DATUM["North American Datum 1983",
            ELLIPSOID["GRS 1980",6378137,298.257222101,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4269]],
    CONVERSION["UTM zone 10N",
        METHOD["Transverse Mercator",
            ID["EPSG",9807]],
        PARAMETER["Latitude of natural origin",0,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8801]],
        PARAMETER["Longitude of natural origin",-123,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8802]],
        PARAMETER["Scale factor at natural origin",0.9996,
            SCALEUNIT["unity",1],
            ID["EPSG",8805]],
        PARAMETER["False easting",500000,
            LENGTHUNIT["metre",1],
            ID["EPSG",8806]],
        PARAMETER["False northing",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8807]]],
    CS[Cartesian,2],
        AXIS["(E)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["(N)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    USAGE[
```

```
        SCOPE["Engineering survey, topographic mapping."],
        AREA["North America - between 126°W and 120°W - onshore and offshore. C
        BBOX[30.54,-126,81.8,-119.99]],
    ID["EPSG",26910]]
Data axis to CRS axis mapping: 1,2
Origin = (559993.999978157342412,5120006.000014467164874)
Pixel Size = (1.000000000000000,-1.000000000000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=LZW
  INTERLEAVE=BAND
  LAYOUT=COG
  PREDICTOR=3
Corner Coordinates:
Upper Left  (  559994.000, 5120006.000) (122d13'19.06"W, 46d13'51.54"N)
Lower Left  (  559994.000, 5109994.000) (122d13'23.64"W, 46d 8'27.18"N)
Upper Right (  570006.000, 5120006.000) (122d 5'31.69"W, 46d13'48.09"N)
Lower Right (  570006.000, 5109994.000) (122d 5'37.03"W, 46d 8'23.74"N)
Center      (  565000.000, 5115000.000) (122d 9'27.85"W, 46d11' 7.71"N)
Band 1 Block=512x512 Type=Float32, ColorInterp=Gray
  NoData Value=-999999
  Overviews: 5006x5006, 2503x2503, 1251x1251, 625x625, 312x312
```

Yeah, I know it's a lot. The important things here are that the `Coordinate System` (map projection) is `UTM Zone 10N` (you can look up EPSG:26910 for more info) with units in `metres` (note the international spelling), `Pixel Size` is `1 by 1 meters` (with lots of significant figures) and that `Band 1` (the only band in the dataset) is `Type=Float32`.

Why are these details important? Firstly, GDAL needs to know how the horizontal units in the projection relate to the vertical units of the elevation data. Many projections appropriate for large scale (high resolution) maps will have units of feet or meters. This is convenient because in that case the elevation data in a file is *probably* going to match the coordinate system (yes, you read that right, some U.S. datasets still use feet, so their could be a mismatch between imperial and metric units). For small scale (low resolution, wide area) datasets the coordinate system is likely to use angular units (degrees) which need to be scaled to generate accurate shaded relief. For data with horizontal units in degrees (like the equirectangular projection commonly used for global topographic & bathymetric data) you'll need to

include `-s 111120` in your GDAL command. (Don't worry, I'll show an example later.)

Secondly, the data within a digital elevation model can come in a few flavors, so it's good to know the data type. In addition to `Float32` (floating point — 32 bit data that can be positive or negative), common data types for elevation are `UInt16` (unsigned integers — 16 bit data with only positive values) or `Int16` (signed integers — 16 bit data with negative values representing bathymetry or land elevations below sea level). Encoding the elevation with one of these data types means there doesn't need to be any scaling to convert from data to real-world units, and there's usually plenty of precision to avoid banding in the data (which would look like terraces in shaded relief — yuck).

Unfortunately, floating point and signed integer data probably won't display properly in consumer image processing applications like Photoshop or GIMP, but require something like QGIS, ArcGIS, ENVI, or the Geographic Imager plugin for Photoshop. If you don't have any of those installed you can generate a simple grayscale image of the DEM with GDAL. You can get the minimum and maximum values from the file by including the flag `-mm` ("Force computation of the actual min/max values for each band in the dataset.") when you run `gdalinfo`, like so:

```
gdalinfo -mm mount_st_helens_USGS_1m_dem.tif
```

This command will return `Computed Min/Max=550.701,2537.783` near the bottom of the info dump.

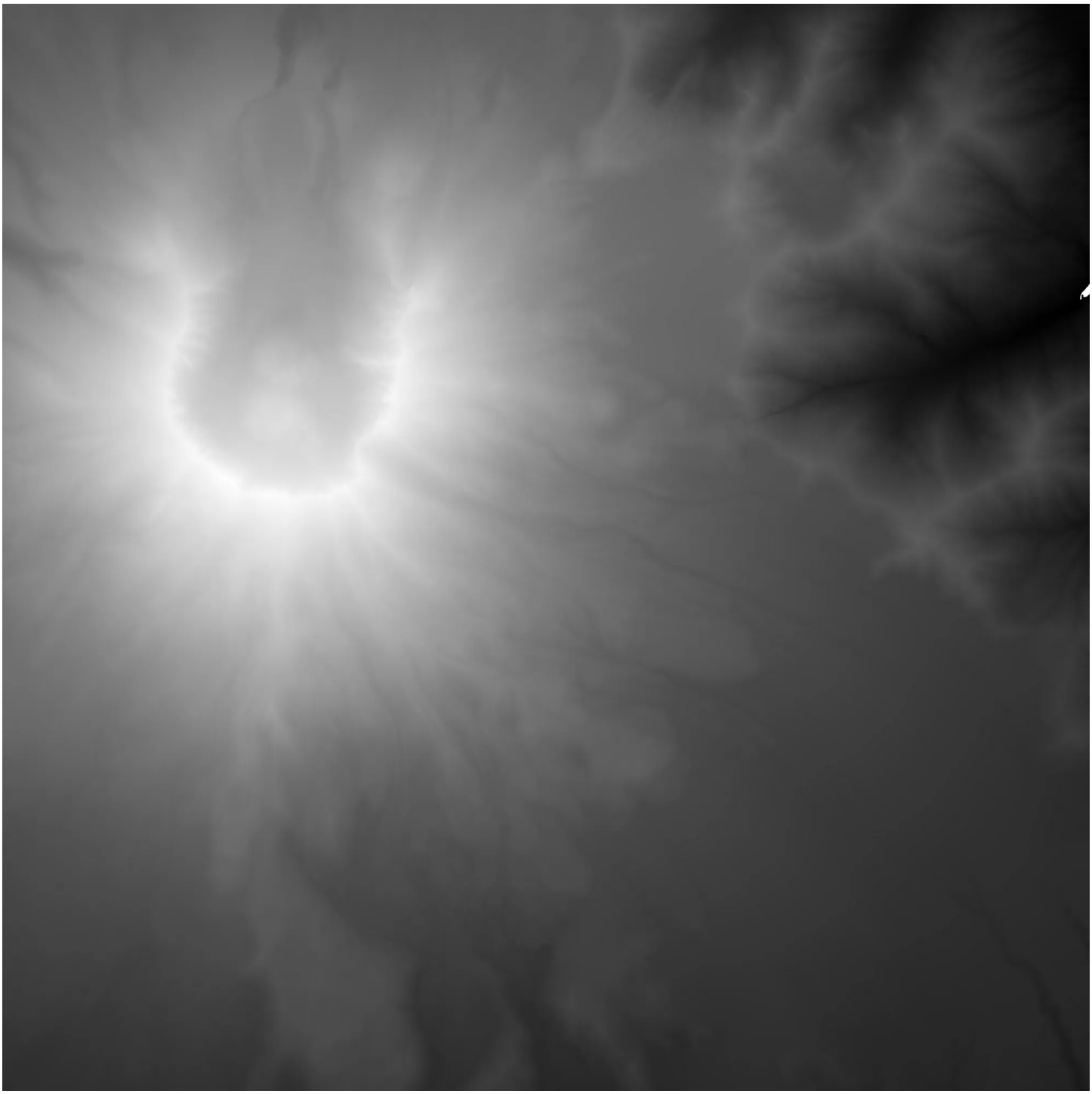
Then convert the data into an 8-bit grayscale file with `gdal_translate`. Use the `-scale` parameter with the values for max and min in the data and max and min of the output data type in the format: `-scale original_minimum original_maximum scaled_minimum scaled_maximum`. You'll probably want the minimum and maximum to span the full range of the output data, which will be 0 to 255 for a typical single-channel 8-bit image. You *can* scale from 0 to 65535 and use `-ot Uint` to convert to a16 bit unsigned integer, but your display is probably only 8 bits anyways, so why bother? Here's the command I used:

```
gdal_translate -scale 550.701 2537.783 0 255 -ot Byte
```

```
mount_st_helens_USGS_1m_dem.tif mount_st_helens_USGS_1m_dem_8bit_gray.tif -co
COMPRESS=LZW
```

As a reminder, `gdal_translate` takes the input filename first, output filename second, and the options can got pretty much anywhere. The “creation option” `-co COMPRESS=LZW` losslessly compresses the data to minimize the file size while preserving 100% of the information. (Creation options are specific to the export file format, so if you’re exporting as something aside from TIFF keep that in mind.)

The resulting image looks like this:



8-bit grayscale representation of Mount St. Helens elevation data, derived from a 1-meter-per-pixel DEM from the USGS National Map.

If the data is formatted properly (the value for “no data” should be specified) and you don’t want to muck about running `gdalinfo` and copy/pasting some numbers by hand, `gdal_translate` will scale the maximum and minimum values in the data to 0

to 255 automatically. Use `-scale` and omit all the max and min values afterwards. You just have to remember to set the output data type to byte with `-ot Byte`, or else you'll end up with data that remains in the original format, but no longer in real world units.

```
gdal_translate -scale -ot Byte mount_st_helens_USGS_1m_dem.tif
mount_st_helens_USGS_1m_dem_8bit_gray_auto.tif -co COMPRESS=LZW
```

That was a lot of steps to get to a kinda plain grayscale image, but datasets will come in a variety of data types and file formats and it's good to know your way around them. Now it's time for the fun stuff.

GDAL includes a tool — `gdaldem` — that creates hill shaded images from DEMs (which can also be appropriated to apply color palettes to numerical data — but I'll get to that later). Here's the generic command to create a grayscale hill shade with the default options:

```
gdaldem hillshade input_dem output_hillshade
```

`gdaldem` is the program (equivalent to `gdalwarp` & `gdal_translate`)

`hillshade` is the mode (I'll describe a few other modes later)

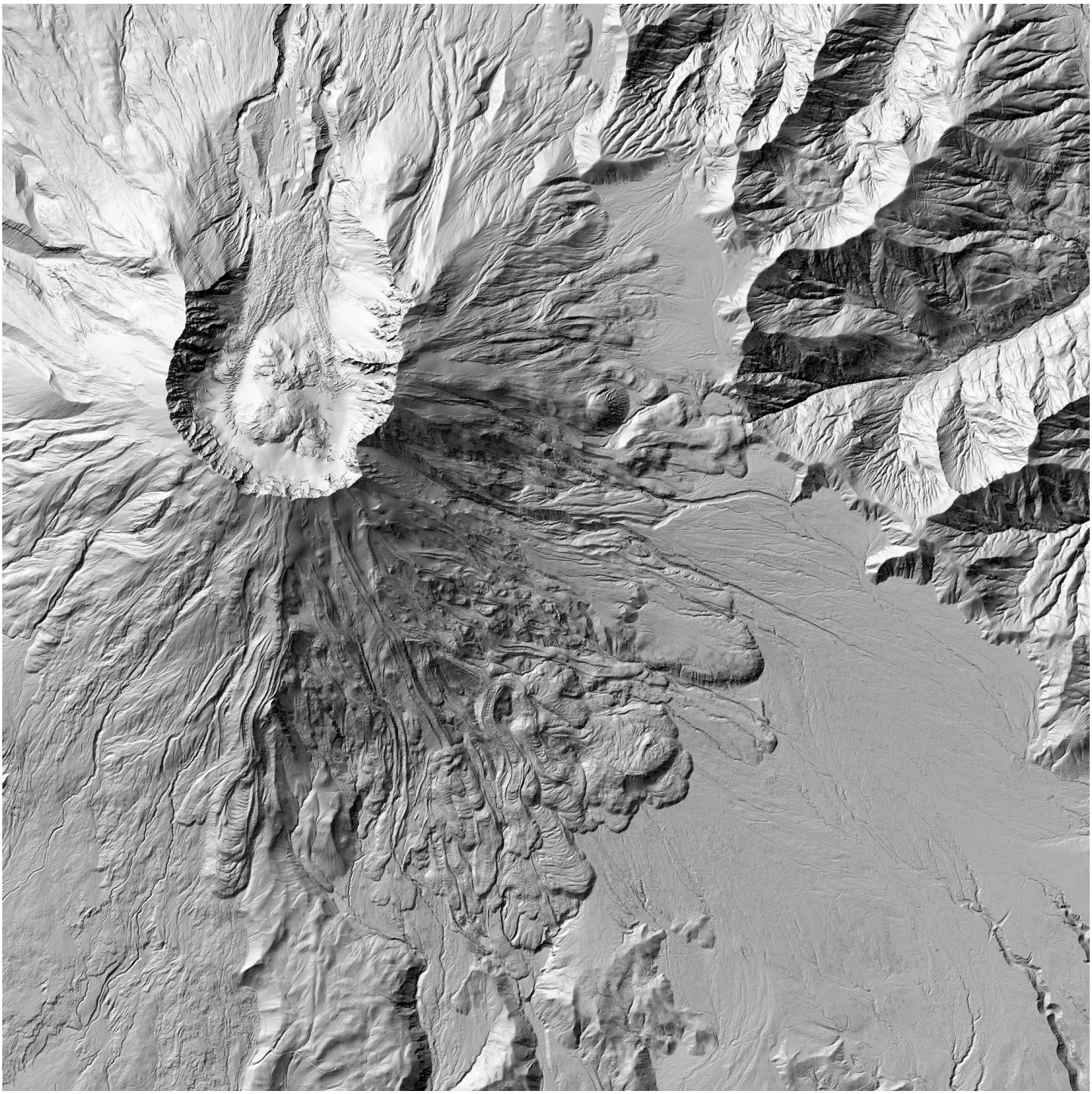
`input_dem` is the source data

and `output_hillshade` is the resulting image (format is guessed from the extension on the output filename — I usually create GeoTIFFs by using `.tif`).

Here's the command with the Mount St. Helens elevation data:

```
gdaldem hillshade mount_st_helens_USGS_1m_dem.tif
mount_st_helens_USGS_1m_dem_hillshade.tif -co COMPRESS=LZW
```

Here's what that looks like:



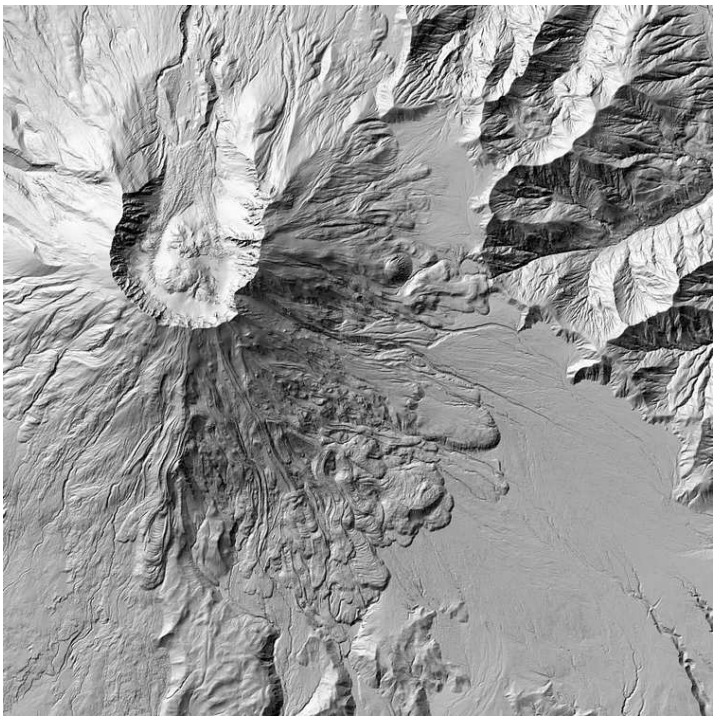
Hillshaded Mount St. Helens elevation data with GDAL's default parameters — 315˚ azimuth and 45˚ elevation angle. The 315˚ azimuth takes advantage of the human brain's preference for interpreting shape and texture when lighting is coming from the upper left, and the 45˚ elevation angle is a good compromise that highlights both steep and shallow slopes. Image derived from the USGS National Map.

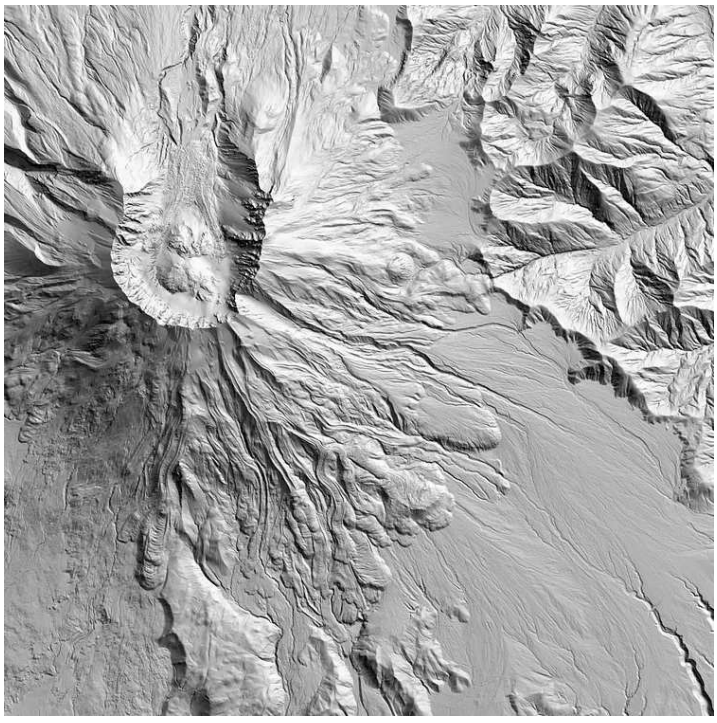
That's a plain hill shaded version of the Mount St. Helens elevation data with GDAL's defaults illumination angles — a 315˚ azimuth (the terrain is lit as if the light source

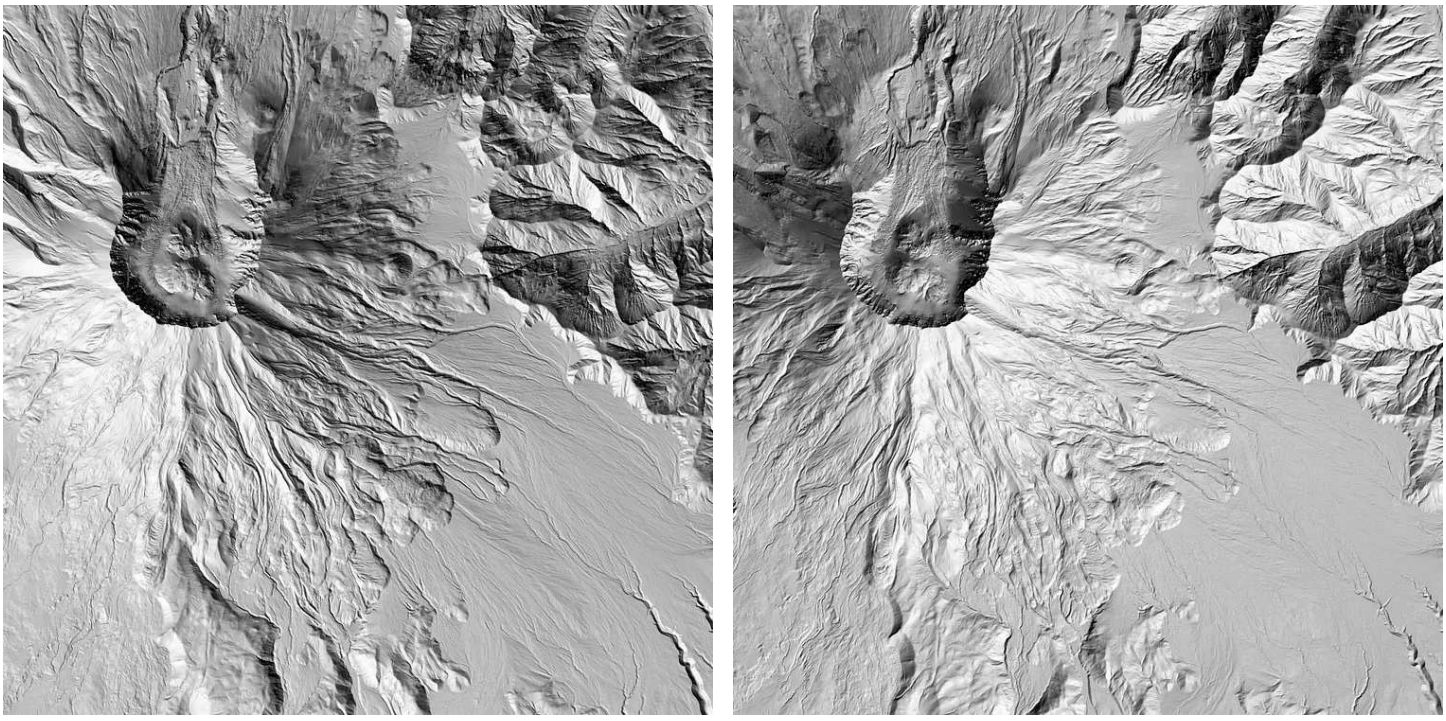
is coming from the upper left (45˚ counter-clockwise from the top of the image)) and a 45˚ altitude (the light is pitched 45˚ (midway) between the horizon and directly overhead). The 315˚ azimuth takes advantage of the human brain's preference for interpreting shape and texture when lighting is coming from the upper left, and the 45˚ elevation angle is a good compromise that highlights both steep and shallow slopes. Note that natural illumination from these angles is impossible at this latitude! Sunlight will always be coming from the south this far north (46.2˚) of the equator.

GDAL allows you to set these angles directly for different effects (or to match the sun's position in a satellite image) with the `-az` (azimuth) and `-alt` (altitude) parameters. Azimuth is measured clockwise from the top of the image, and altitude is measured from the horizon (0˚) to the directly overhead (90˚).

Here's what the terrain looks like with the sun coming from the upper left (315˚), upper right (45˚), lower right (135˚), and lower left (225˚):

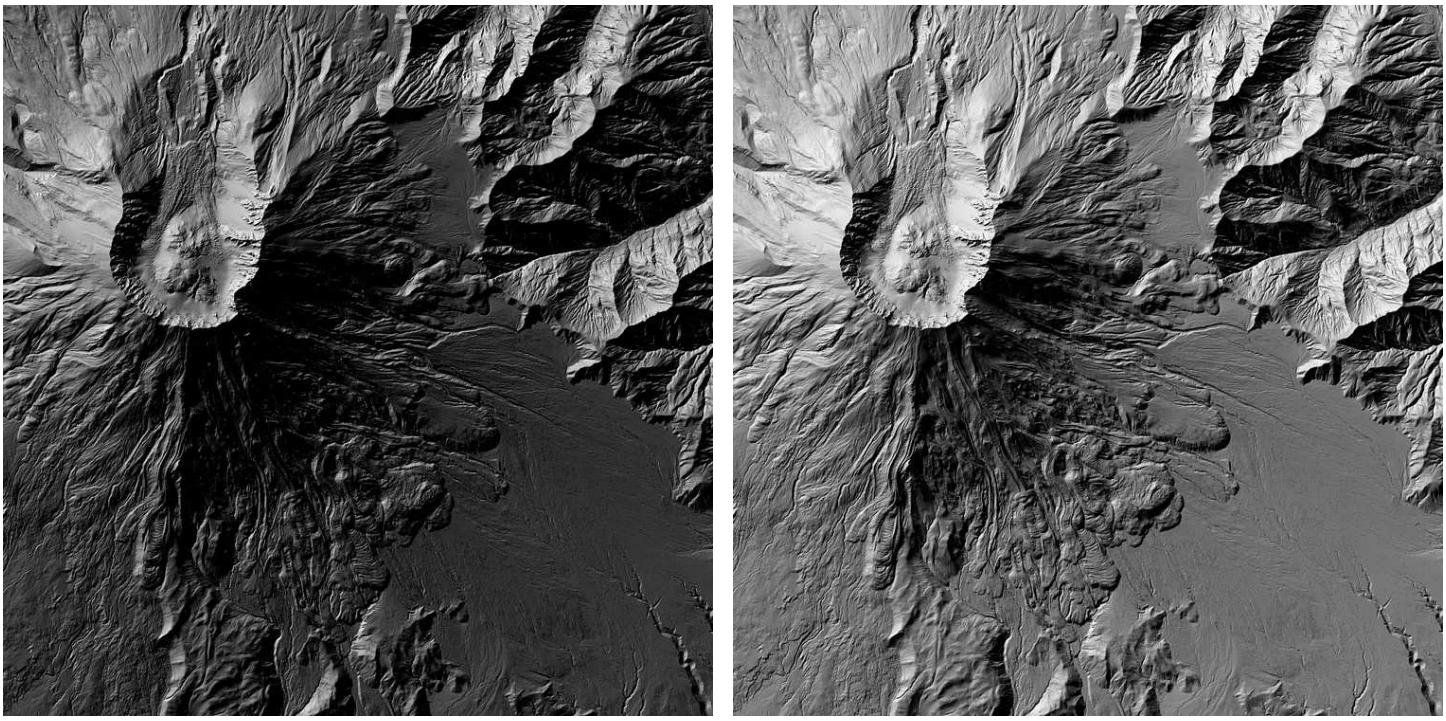


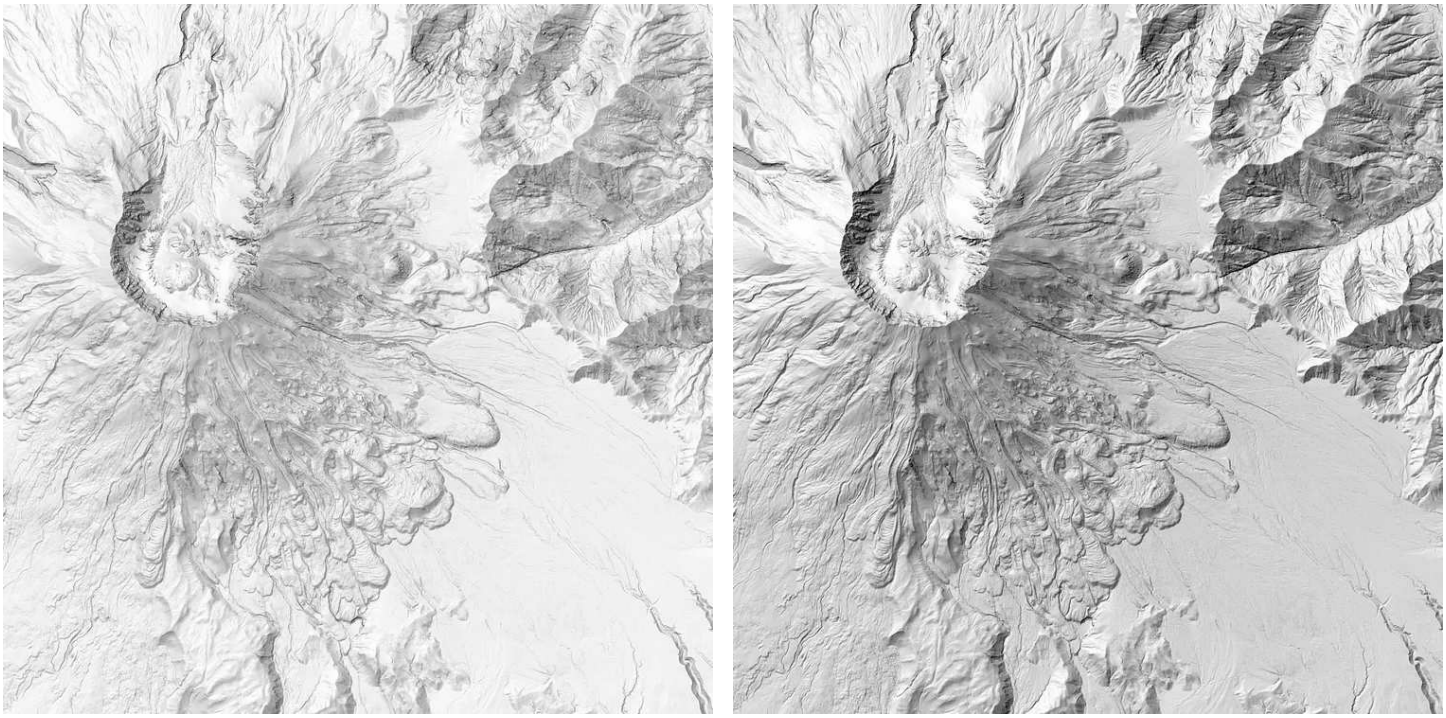




Four hillshaded images with varying azimuth. Images derived from the USGS National Map.

And here's what it looks like at four different altitudes — 15˚, 30˚, 60˚, and 75˚ above the horizon (clockwise from upper left). Azimuth is constant at 335˚. Notice how the overall image gets brighter as the illumination source gets closer to the zenith. The brightest slopes will be perpendicular to the light source, and the darkest will be angled 90˚ or more away.





Four hillshaded images with varying altitude. Images derived from the USGS National Map.

This is all well and good, but what if you want to implement some of the more advanced techniques described by Tom Patterson on the Shaded Relief site? GDAL's got you (partially) covered. There are a few alternate shading methods available for the `hillshade` mode that you can select by adding a flag into the `gdaldem` command:

`-multidirectional` blends light sources from several different angles, clustered around the default 315˚. This helps make sure that linear features aren't over- or under-estimated if they are aligned perpendicular or parallel to the light source.

`-combined` is "a combination of slope and oblique shading" (from the gdaldem documentation) I'm not quite sure what this is doing, but to my eye looks like it's emphasizing texture more than slope.

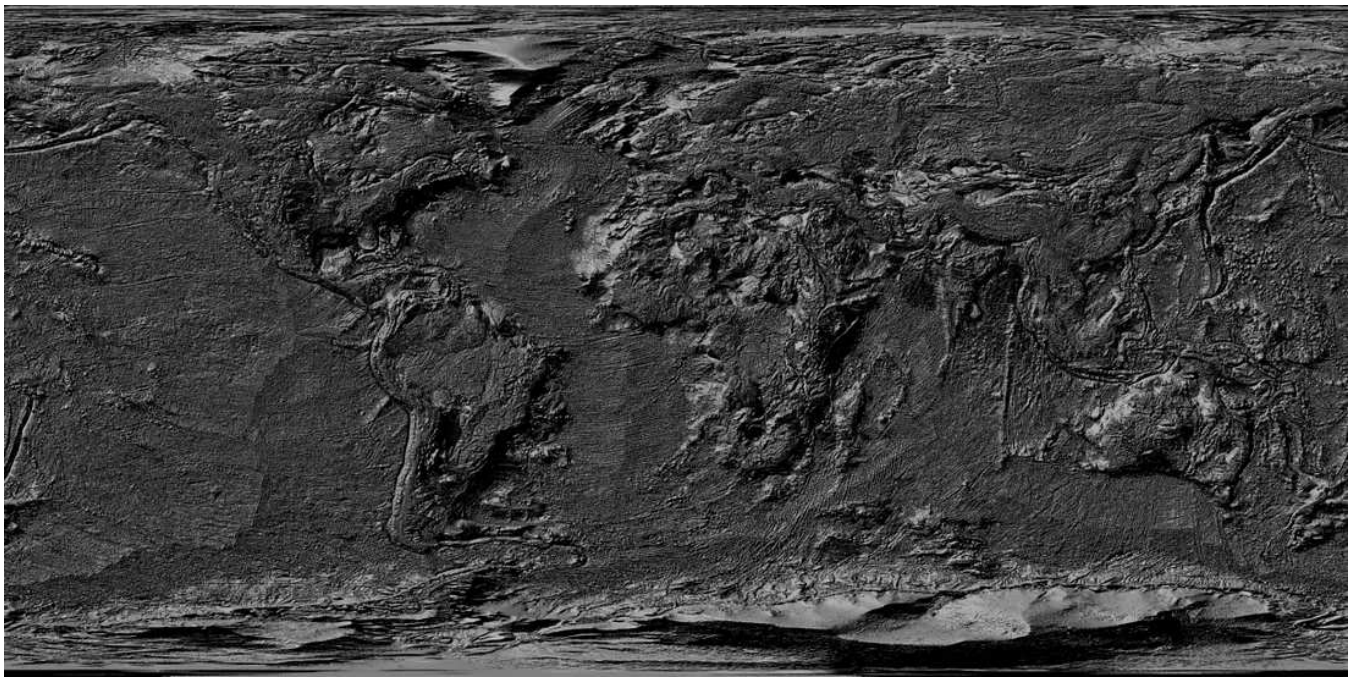
`-igor` is a more subtle type of hillshading designed to be used in combination with additional layers of data. It's similar to `-combined` but lower-contrast.

GDAL's alternate hillshade algorithms — from left to right: multidirectional, combined, and igor. Images derived from the USGS National Map.

In practice, I find these different shading algorithms work best blended together in image processing or GIS software, with brightness and contrast adjustments tweaked to suit the specific map I'm making. But that's probably the topic of another tutorial.

So far I've only shown data with matching horizontal and vertical units. What happens when you try to make a map with units in degrees, which is typical of most global datasets, like GEBCO combined topography & bathymetry?

```
gdaldem hillshade GEBCO_7200px.tif GEBCO_7200px_hillshade.tif -co COMPRESS=LZW
```



Global shaded relief with default scaling — not exactly right! Derived from the GEBCO 2023 Grid.

Oh … oh no. The Earth isn't *nearly* that bumpy.

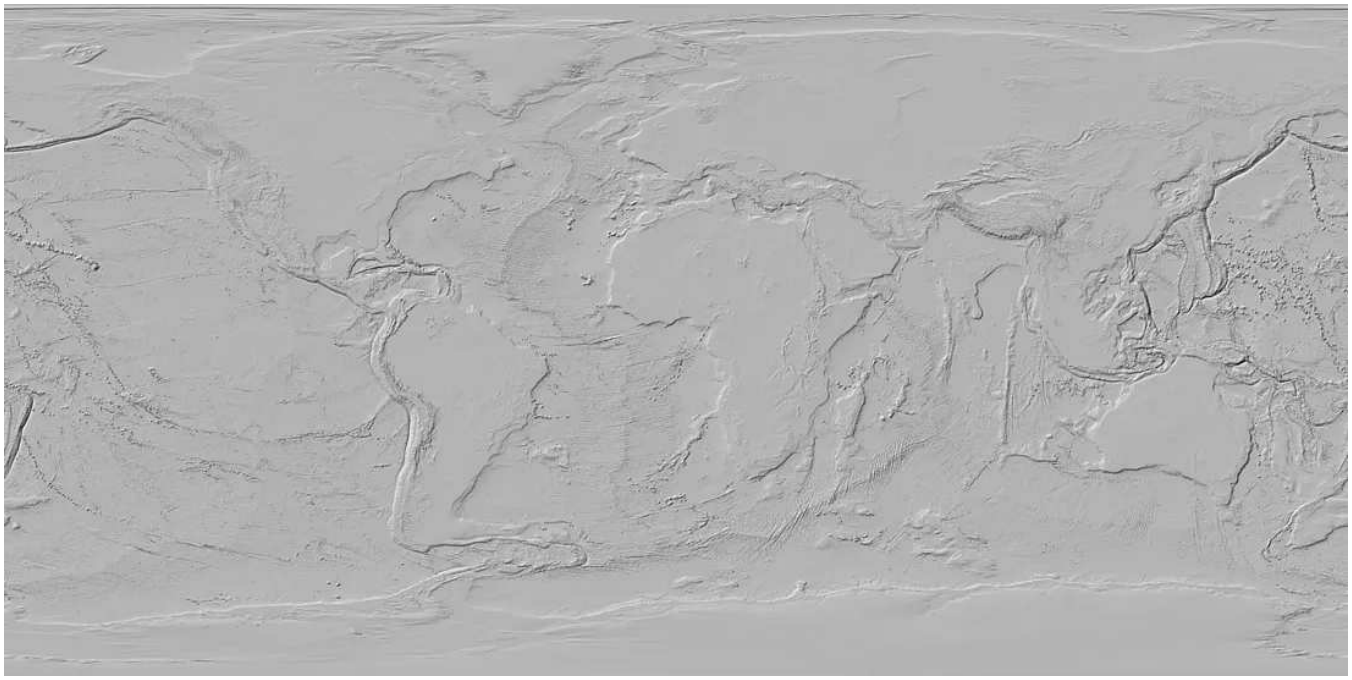
Fortunately there's a relatively easy solution, *if* you know the magic number. Add `-s 111120` to the command, like this:

```
gdaldem hillshade -s 111120 GEBCO_7200px.tif GEBCO_7200px_hillshade_scaled.tif -co
COMPRESS=LZW
```

Global shaded relief with -scale set to 111120 to convert from degrees to meters. Realistic, but underwhelming. Derived from the GEBCO 2023 Grid.

Better! But since the scale of the Earth is a lot bigger than the highest relief on land or in the oceans, a realistic shaded relief map at global scale is pretty underwhelming. As with `-s`, there's a single-command that will adjust vertical exaggeration: `-z`.

```
gdaldem hillshade -s 111120 -z 8 GEBCO_7200px.tif
GEBCO_7200px_hillshade_scaled.tif -co COMPRESS=LZW
```



Global shaded relief with correct scale and 8× vertical exaggeration — better! Derived from the GEBCO 2023 Grid.

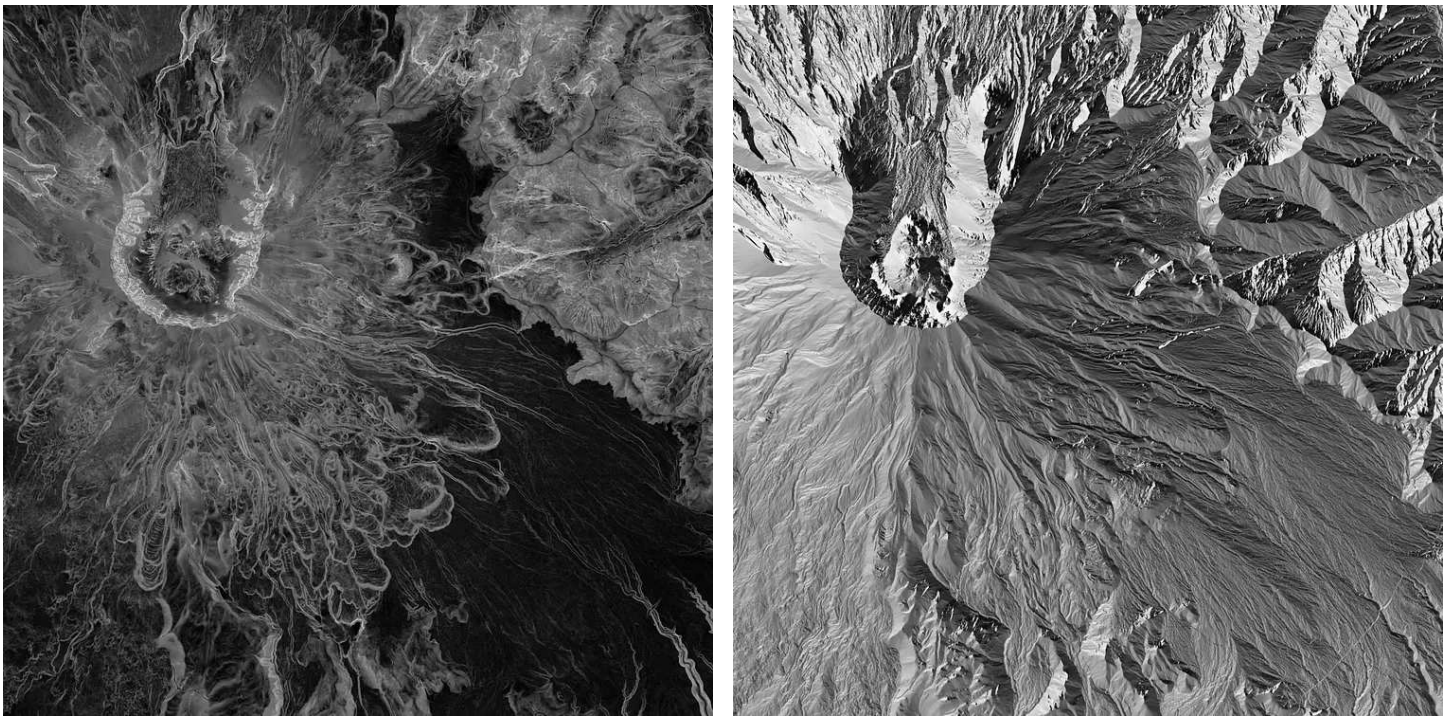
For a global map displayed on screen, a vertical exaggeration value of 8 looks pretty good to me, but the exact value will depend on what a map is trying to show and how it will be displayed. It might even be beneficial to vary the vertical exaggeration between land and oceans to balance the apparent relief. Maps are, after all, tools, and it's best to create maps that communicate effectively rather than maps that are 100% literal but misleading or hard to interpret.

Aside from hill shading, what else can `gdaldem` do with topographic data? This is where the different *modes* come in. `Slope` and `aspect` are relatively straightforward, in my opinion.

`Slope` is the steepness of the terrain, by default calculated in degrees with a range of 0˚ — 90˚, but can be set to percent with the -p flag, in which case the values will be go from 0% (flat) to up to 100% (vertical).

`Aspect` is the direction the slope is facing. It ranges from 0˚ (directly north) clockwise — 45˚ is northeast, 90˚ east, 135˚ southeast, etc. Keep in mind that the numbers wrap, so a slope of 359˚ is oriented mostly north, angled just a *tiny* bit to the west.

Notice that both slope and aspect return floating point results, not 8-bit (grayscale) like the images from hillshade. Here's what maps of these two parameters look like (at least after they've been scaled and converted from data to imagery):



Slope (left) and aspect (right) maps of Mount St. Helens. In the slope map relatively flat areas are dark, steep areas are light, and the brightest sections are nearly vertical. In the aspect map, northeast-facing slopes are the darkest, with southeast-, southwest-, and northwest-facing slopes getting progressively brighter. Images derived from the USGS National Map.

So far the examples I've shown have been a little, well gray. What if I wanted to bring a little flair? A bit of pizzazz? A dash of color? `Color-relief` will convert a plain old DEM into a map with hypsometric tints, cartographic jargon for color-by-numbers. With `color-relief` you'll need *two* input files. The digital elevation model and a text file that tells the algorithm which colors go with which numbers. The text file is formatted like so:

```
elevation red green blue alpha(optional)
```

Put the elevation in the first column, then the value for red, then green, then blue, with an optional column for alpha (transparency). Elevation can be an integer or floating point value, and each color component is an integer from 0 to 255 (one byte's worth). Alpha also ranges from 0–255, with 255 being fully opaque. (The default is opaque if there's no value for alpha.) Column separators are fairly flexible — commas, spaces, tabs, or colons. Here's an example, modified from the "cold humid" color palette in The Development and Rationale of Cross-blended Hypsometric Tints, by Patterson & Jenny:

```
500 112 147 141
600 120 159 152
700 130 165 159
900 145 177 171
1100 180 192 180
1300 212 201 180
1500 212 184 163
1800 212 193 179
2100 212 207 204
2400 220 220 220
3000 235 235 237
4000 245 245 245
```

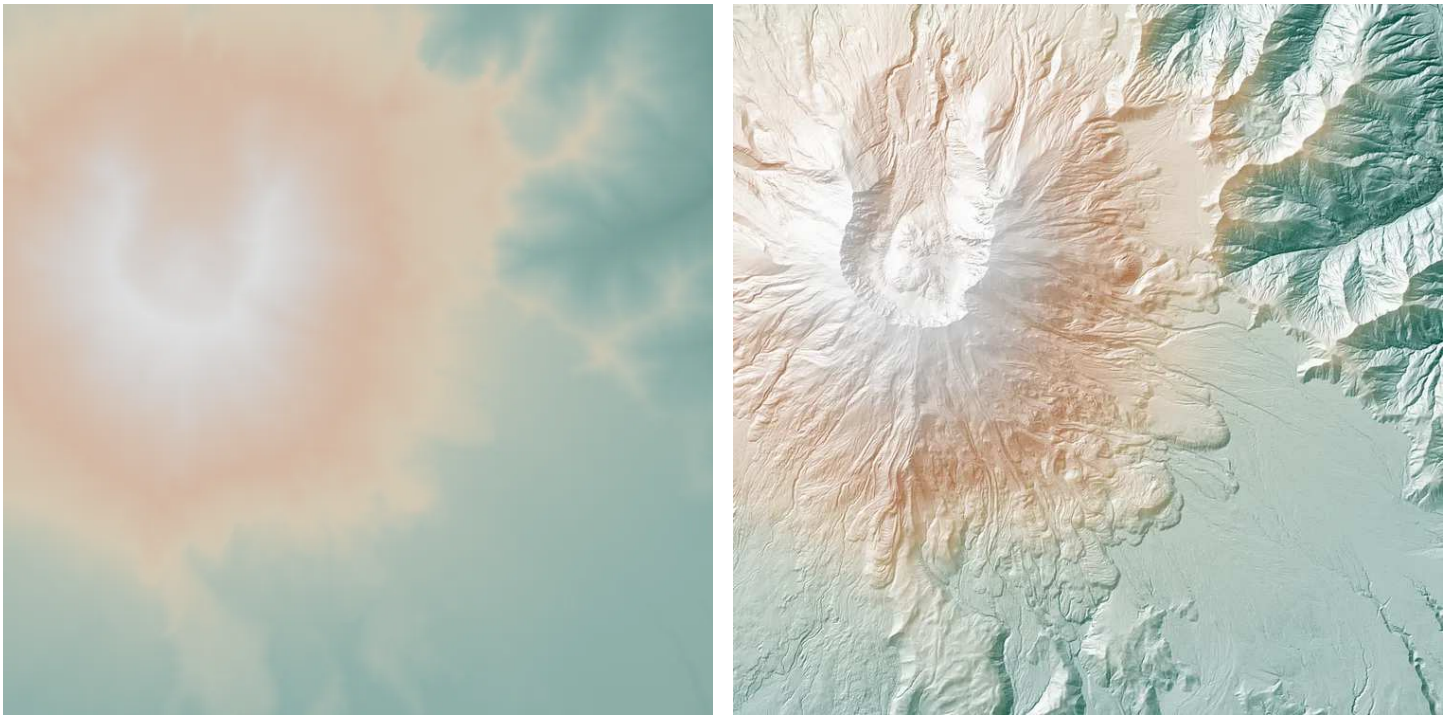
Here's how to run the generic version of the command:

```
gdaldem color-relief input_dem color_text_file output_color_relief_map
```

Substituting the elevation data for Mount St. Helens and my custom color palette file (all on one line, Medium is wrapping the text):

```
gdaldem color-relief mount_st_helens_USGS_1m_dem.tif st_helens_hypsometric.txt
mount_st_helens_USGS_1m_cold_humid_hypsometric.tif -co COMPRESS=LZW
```

Which yields a color image (below, left). By itself it's not all that interesting — it may even be less informative than a plain grayscale image of the mountain's elevation (like the image towards the top of this post). But combined with shaded relief, even a simple hillshade with a 335˚ azimuth, the map starts to come alive (below, right).



Color-coded elevation map of Mount St. Helens (left). Green indicates low elevations, beige and light orange indicate medium elevations, and light gray indicates high elevations. Elevation map combined witha simple hillshade (right). Images derived from the USGS National Map.

Note: I made the combined elevation/shaded relief image by using blend modes in QGIS, accessible from the `Layer Rendering` options. The `overlay` blend mode applied to the hillshade layer makes areas of the layer below *lighter* when brightness is above 50% and *darker* when brightness is below 50%. As a result, slopes facing towards the illumination source get lighter, while those facing away get darker.

Color-coded elevation data plus a grayscale hillshade are the fundamental components of a shaded-relief map. But there are a wealth of additional techniques cartographers use to illustrate topography — and it's possible to replicate some of these advanced techniques with the tools provided by GDAL.

For example, Swiss style cartography and other elegant topographic maps often

employ subtle shifts in hue that emulate the contrast between directly sunlit surfaces — which exhibit warm (yellow) hues — and shadowed surfaces — which are tinted with cool (blue) colors. Conveniently enough, the output from GDAL's `aspect` mode has the information needed to replicate this effect, and the `color-relief` mode can be applied to any numerical data — not just elevation!

As I mentioned earlier, a slope's aspect is merely the direction it faces. So if one color-codes aspect so that slopes facing the illumination source are given a slight yellow tint, and slopes facing away from the illumination are colored a bit blue, the resulting map takes on a nice touch of realism.

Just like creating colors from elevation data, both a data file (in this case floating point values generated from the digital elevation model with the `aspect` mode of `gdaldem`) and a text file (that specifies what colors go with which values) are necessary to run `color-relief`.

The tricky part is that aspect varies from 0˚ to 360˚, wrapping around so that both 0˚ and 360˚ should be represented by the same color, or else there will be an obvious discontinuity east and west of north. (I mapped the elevation data sequentially from low to high, so those colors are intentionally distinct.) With a typical direction of illumination of 315˚, the slope colors should look something like this:

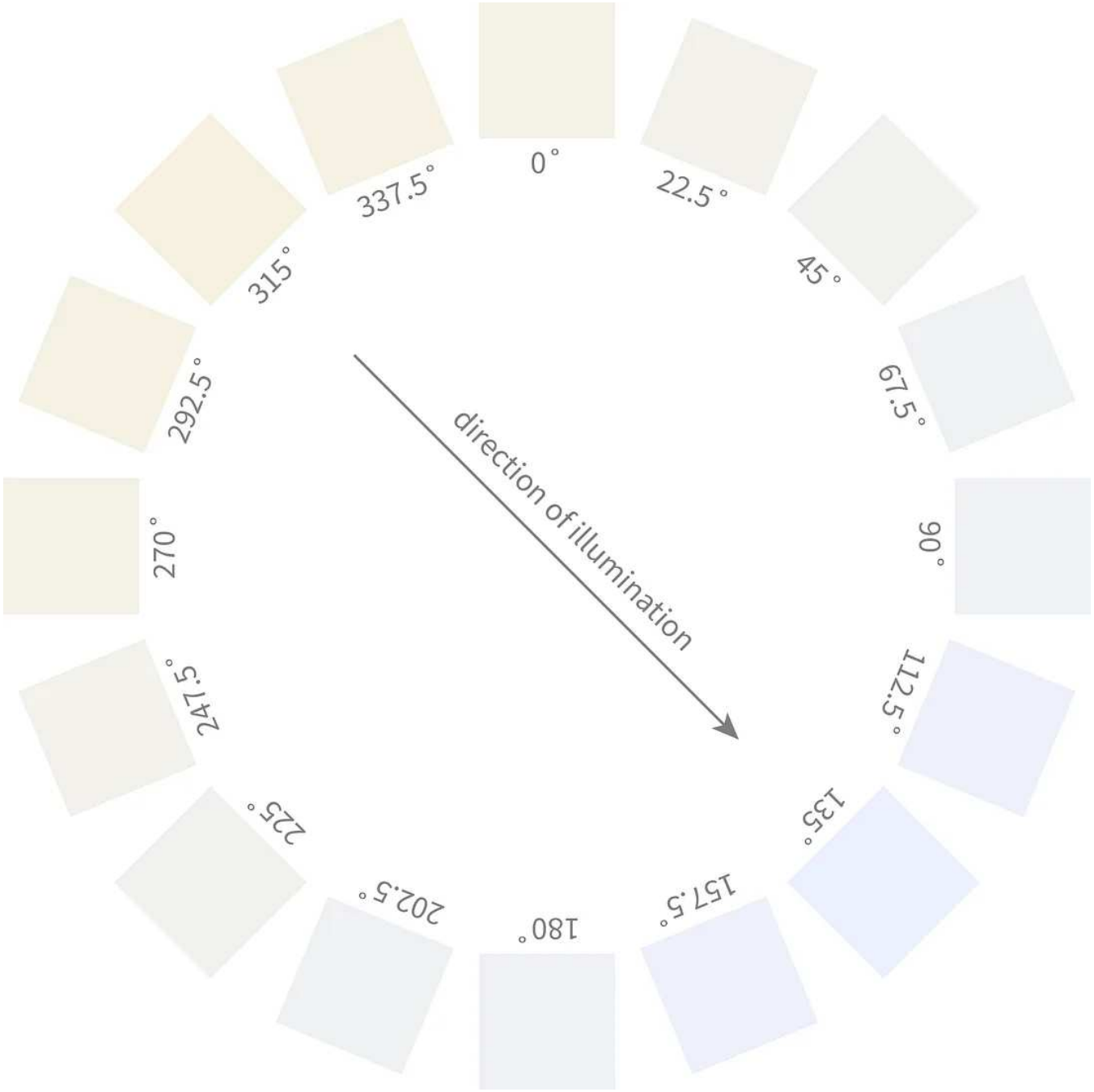


Illustration of colors applied to an aspect map that give a touch of realsim to a shaded relief map. Note that the sequence of colors is the same going from 315˚ to 135˚ in both the clockwise and counter-clockwise directions. Illustration by me.

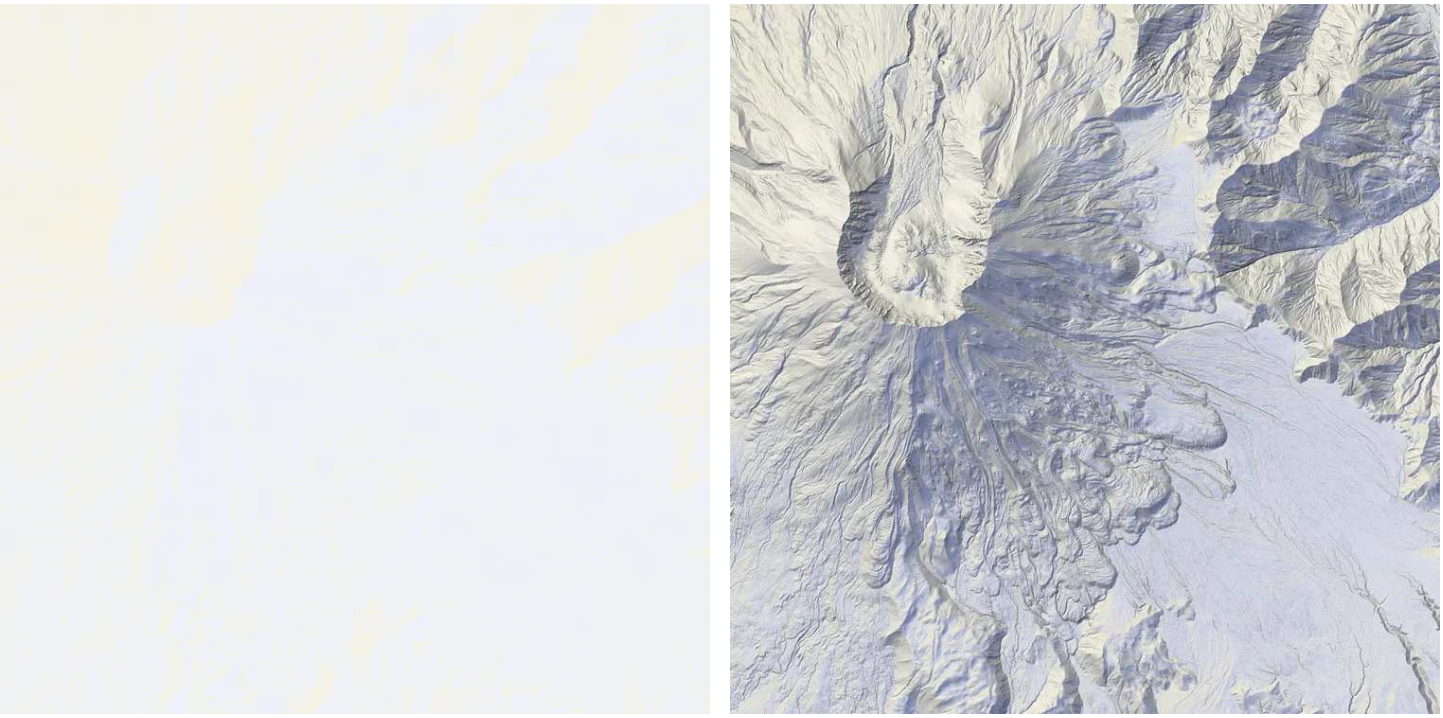
The variation in hue is symmetric, so that colors going clockwise from 315˚ to 135˚ are the same as those going counterclockwise. One thing to note is that the lightness of all these colors should remain constant while hue and saturation vary — so I used the HCL Wizard Palette Creator to make a color ramp. (If you want to learn more about human perception, color models, and how they apply to data visualization you

can read Lisa Charlotte Muth's superb posts on color on the Data Wrapper blog, or my own series Subtleties of Color.)

Here's what the text file associating colors with aspect angle looks like:

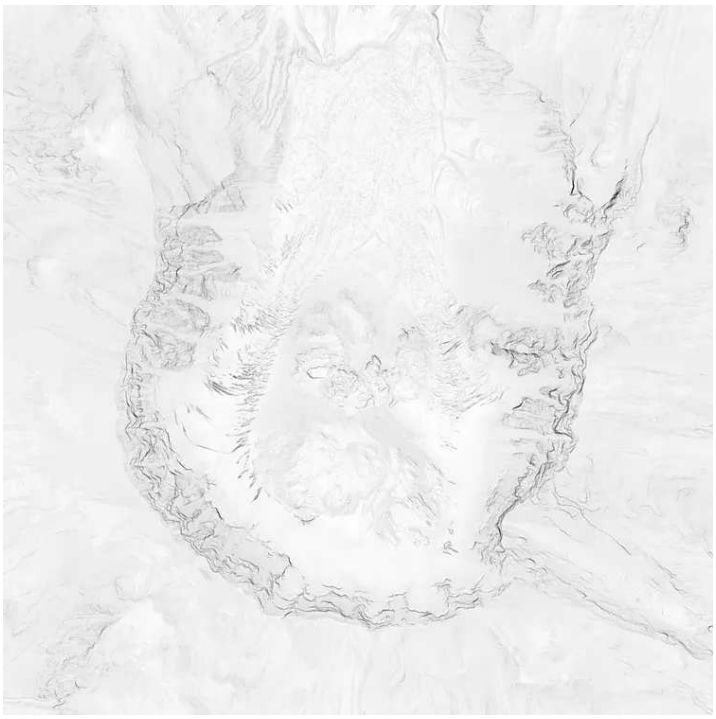
```
0  243 241 231
22.5  242 241 235
45  241 241 239
67.5  240 241 243
90  239 241 246
112.5  238 240 250
135  236 240 254
157.5  238 240 250
180  239 241 246
202.5  240 241 243
225  241 241 239
247.5  242 241 235
270  243 241 231
292.5  245 241 227
315  246 241 222
337.5  245 241 227
359.999 243 241 231
```

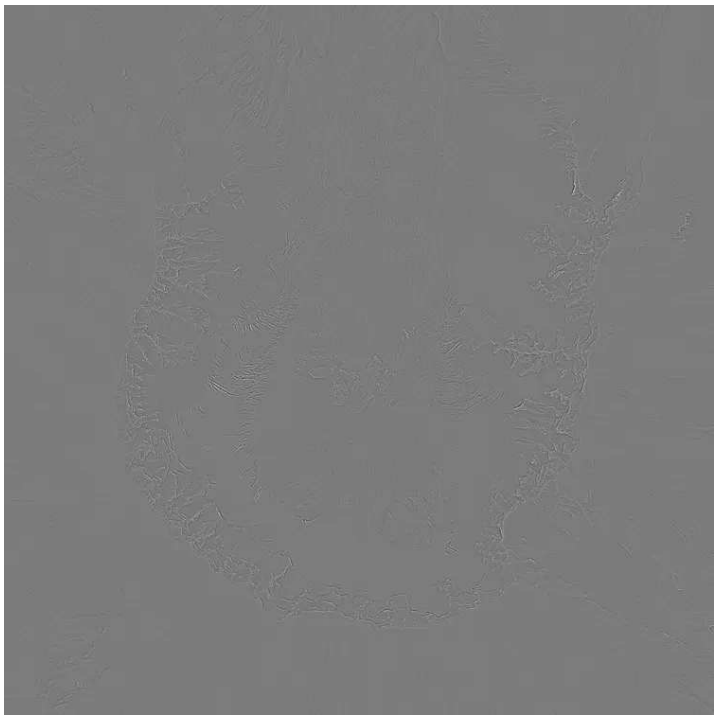
We're not all that great at distinguishing hues, especially desaturated (pale) hues, so the raw output (below, left) can be underwhelming. It looks better (and is much more interpretable) with hillshade (below, right).



Colorized aspect data isn't all that interesting on its own (left), but it really comes alive when blended with shaded relief (right). Images derived from the USGS National Map.

GDAL's final three modes are on the technical side, but they can add some subtle details to shaded relief maps so I think they're worth going over. `TRI` (Terrain Ruggedness Index), `TPI` (Topographic Position Index), and `roughness` are all measures of the variability in elevation in the area immediately surrounding each pixel. Terrain Ruggedness Index and roughness are so similar as to be almost indistinguishable, even viewed side by side. So I'll just show roughness (below, left). Note that I've inverted the image, so the roughest areas are black and the smoothest white. Topographic Position Index is at least *different,* but mostly gray with a few lighter & darker details where the topography is very complex.





Images derived from the USGS National Map.

The implementations of these algorithms in GDAL are all limited to a very small region (the 9 pixel square formed by a central pixel and the 8 pixels immediately surrounding it) so other software may be more useful if you need these parameters for analytic work. But the outputs share some similarities with the ambient occlusion and texture shading techniques that are becoming increasingly popular in cartography — so I think they're worth experimenting with.

The most elegant and informative maps don't typically rely on a singe technique. Rather, they are the end result of multiple elements blended together. Some obvious, some subtle. Here's my map of Mount St. Helens, incorporating six different layers: roughness, hypsometric tints, Igor hillshade, multidirectional hillshade, simple hillshade with illumination from 315˚, and aspect colored yellow and blue. Pretty intricate coming from a single dataset!

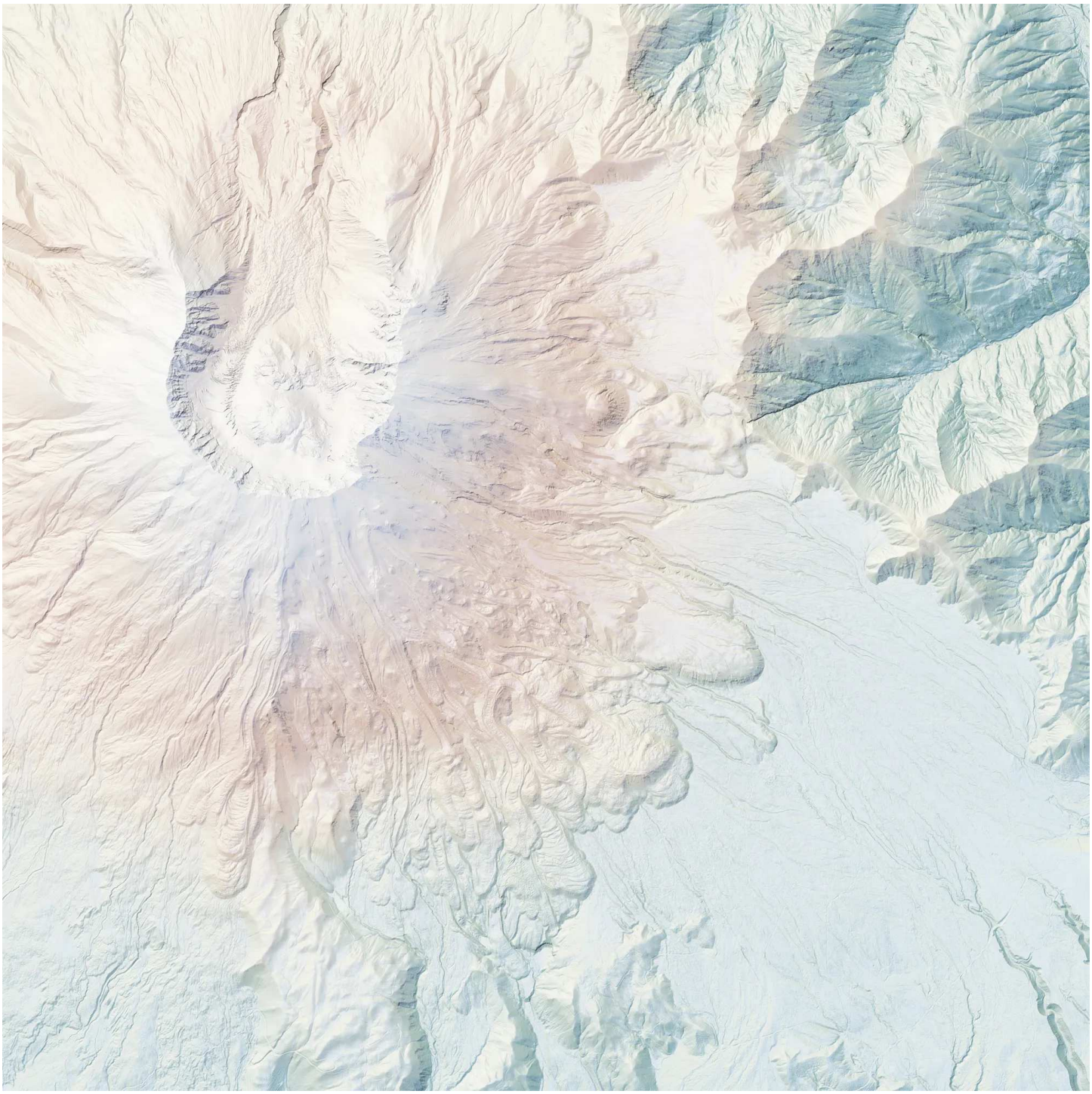


Image derived from the USGS National Map.

Now that I've gone through all this you may be wondering — why bother with GDAL? Can't all this be done in QGIS, or ArcGIS, or any number of other very capable applications? Well, yeah, it can (and QGIS even uses GDAL under the hood). But GDAL is *wonderfully* scriptable. I converted most of the web-friendly PNGs from

high-res TIFFs with gdal_translate:

```
for file in *.tif
do
  gdal_translate -outsize 1200 0 -r bilinear -of PNG $file ${file%.tif}.png -co
done
```

And you can even script the input parameters, like this:

```
for azimuth in {1..360}
do
  gdaldem hillshade -az $azimuth ../mount_st_helens_USGS_5m_dem.tif 00${azimuth
done
```

Which outputs 360 files with illumination azimuths coming from 1–360˚. Perhaps a bit of a gimmick here, but potentially useful. On the other hand, maybe you'd like to create custom multidirectional hillshading, automate the production of dozens of maps, or show a client how light would vary in a park over the course of a year — all possible with a bit of creativity.

Animated Shaded Relief: Mount St. Helens

360 different angles of hillshade on a 1-meter-per-pixel Digital elevation model of Mount St. Helens.

youtube.com

In my next post I'll show how to use `color-relief` with other datasets, that have nothing at all to do with elevation or hillshading. Plus some tips on doing calculations, reading scientific formats like HDF, and integrating GDAL with Python. With any luck it'll take fewer than 6 years!

1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp
3. Geodesy & Local Map Projections
4. Working with Satellite Data
5. Shaded Relief (you are here)
6. Visualizing Data
7. Transforming Data

Thanks go to Frank Warmerdam, who got me over my fear of GDAL, and Tom Patterson, for sharing his deep knowledge of mountain cartography, and Bernice Rogowitz, for opening my eyes to the possibilities of color.

GIS Cartography Data Visualization Maps Gdal

Follow

Written by Robert Simmon

2.4K Followers

Data Visualization, Ex-Planet Labs, Ex-NASA. Blue Marble, Earth at Night, color.

More from Robert Simmon

A Gentle Introduction to GDAL Part 6.1: Visualizing Data

Robert Simmon · Follow

16 min read · Oct 6, 2023

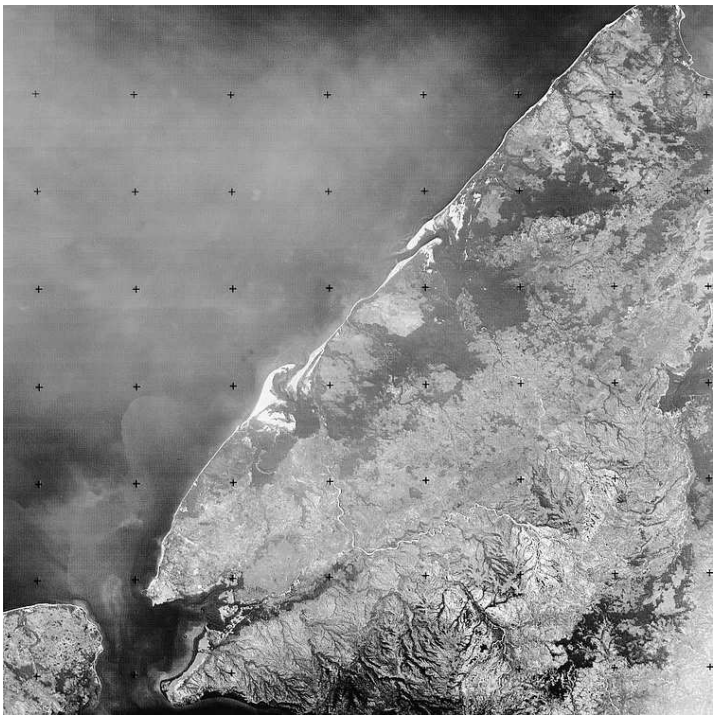
You may be wondering what I mean by "data visualization with GDAL"? After all, aren't satellites images and maps — prominently featured in previous installments — both *data?* Well, yes. But there are differences between types of data that I think it's worth expanding on, before diving into examples of using GDAL to create thematic maps (maps "used to emphasize the the spatial pattern of one or more geographic attributes" from *Thematic Cartography and Geographic Visualization*). If you want, you can just skip to the tutorials, but if you're curious, hang around for a bit.

Images vs. Data

The first Earth observations returned from space — everything from Corona spy satellites to the first pictures from a weather satellites to photographs of the Earth from Apollo astronauts — were more on the "image" side of the data/image divide than the "data" side. All of these examples were recorded with, broadcast by, and stored on analog equipment (seriously, the early TIROS data is stored in a series of oversized books), with the variations in brightness representing relative, not absolute, differences in intensity.

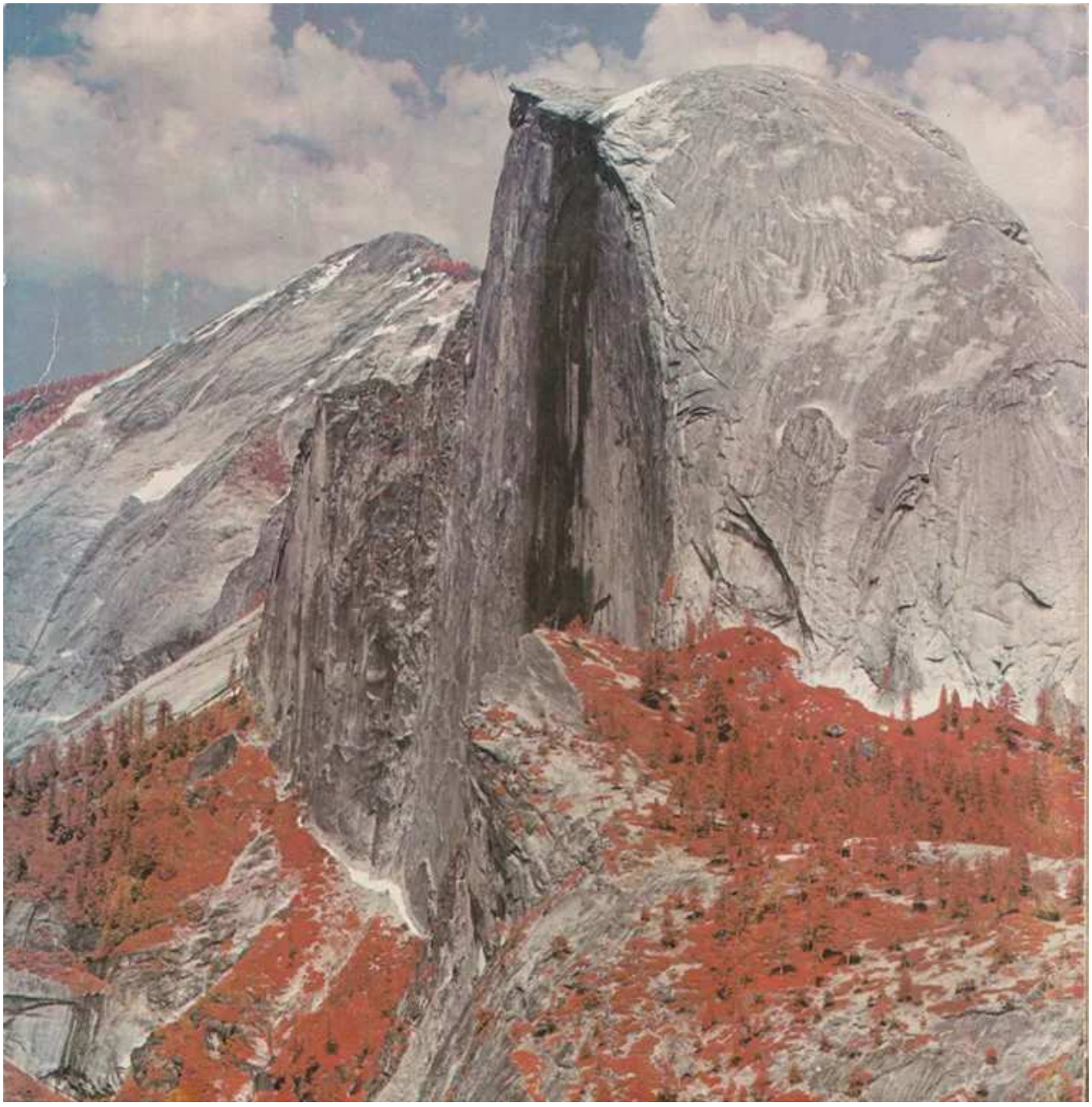
Half Dome and Glacier Point, Yosemite National Park, imaged by a Corona satellite on October 21, 1964. Image by Robert Simmon based on data from the USGS Earth Explorer. CC-by-SA-4.0.

The information these pictures contained was more *qualitative* than *quantitative*, with the types of analysis that could be performed more suited to words than equations. These distinctions aren't absolute. It's certainly possible to make qualitative imagery into quantitative data with operations like counting aircraft or tracing the outline of a glacier. But for the most part, early satellite data is best thought of as a photograph (and some data *were* photographs, with the film returned to Earth).



Landsat Return Beam Vidicon image of Madagascar collected on June 8, 1981 (left), and Earth from Apollo 16, taken on April 16, 1972 (right). Early satellite data was, in general, more qualitative than quantitative. Images from the USGS (left) and NASA (right).

This changed in the 1970s as satellite remote sensing moved into the digital age, spearheaded by Landsat's Multispectral Scanner (MSS). (Keep in mind there's nothing *inherently* superior about digital vs. analog — an analog HDTV signal is higher quality than a digital DVD. "Digital" in this usage just means measurements broken up into discrete values. Digital data can be more easily stored and replicated than analog data, and computers — which require digital inputs — enable revolutionary types of analysis.) The MSS was originally intended to be complementary to Landsat 1's principal instrument — the TV-like Return Beam Vidicon (RBV). However, users soon realized the MSS was superior to the RBV. Scenes from the MSS were more uniform, more precisely aligned with features on the surface of the Earth, were more consistent from image to image, and contained two near infrared bands (which measured light in wavelengths slightly longer than what human eyes can see) that proved critical for mapping vegetation.

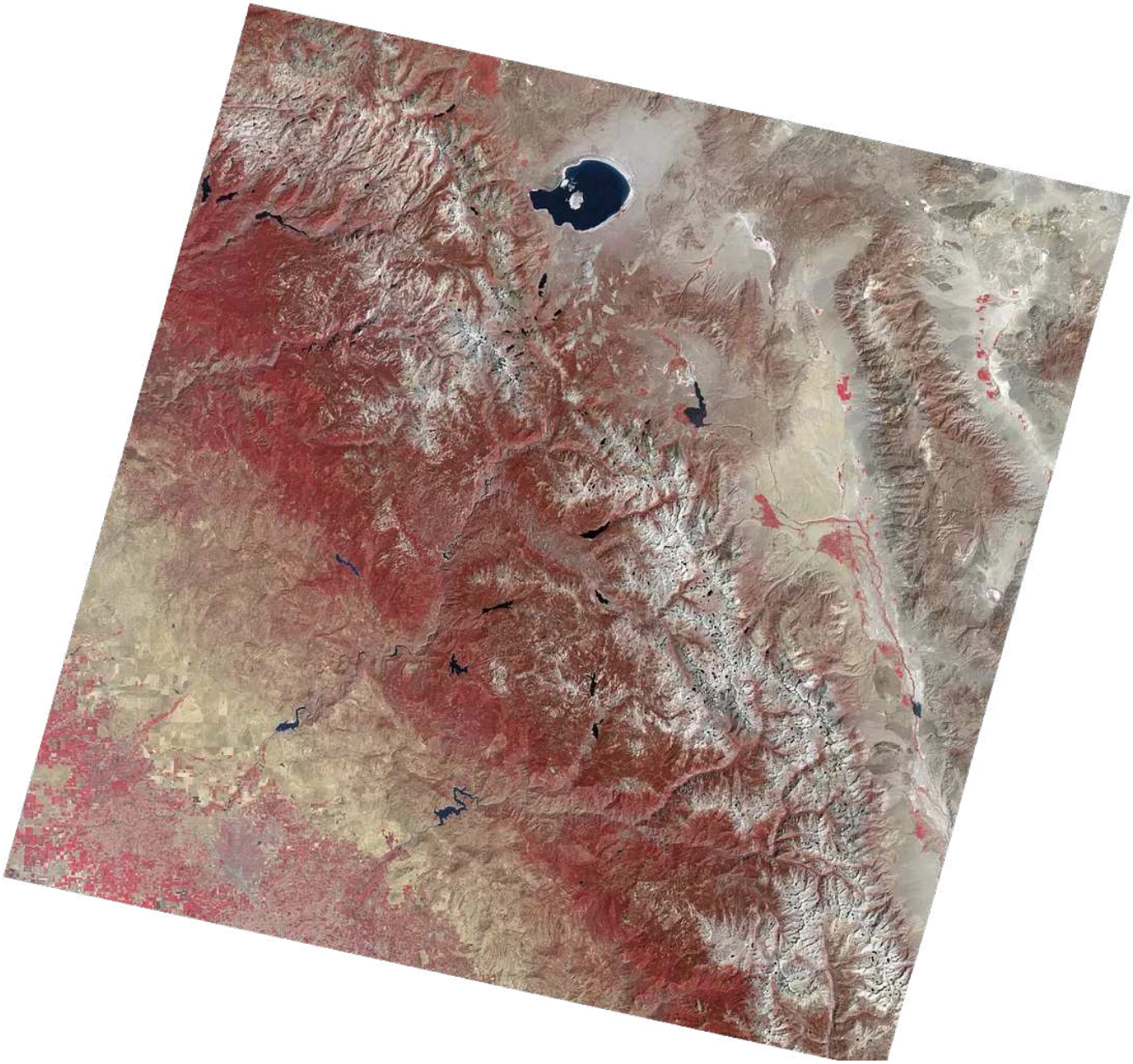


False-color image of Half Dome, Yosemite National Park, California taken by the Multispectral Scanner engineering model, which ended up being flown on Landsat-1. Photo courtesy NASA/USGS Landsat Program.

Crucially, data from the MSS were *calibrated.* Rather than being a relative measure, each pixel represented the amount of energy detected by the sensor in four separate wavelengths of light. In other words the instrument collected photons over an 80 by 80 meter point on the Earth's surface, converted those photons into an electric charge which was then measured and stoered as a digital number (sometimes

referred to as a “DN”). A process very similar to a modern digital camera. The resulting digital number can then be re-converted into real world units of radiance — *Watts per meter squared per steradian.*

That sounds scary, and maybe it is a little bit, but it accounts for the complexity of precisely measuring the Earth’s features from space. Radiance is usually converted into *reflectance* which is conceptually a bit easier to understand. Also known as *albedo* (a term I first heard listening to Vangelis, believe it or not), reflectance is the proportion (or percentage) of sunlight incident on an area being measured that is received by the sensor. For examples asphalt reflects less light than fresh snow. Unlike radiance, which varies based on ambient conditions, reflectance is a property of a surface. So observations collected at different times of day, different places, or under different conditions, can all be compared with each other. There is a lot of complexity here, and this is a digression within a digression, so if you want to know more I found this explanation very helpful.

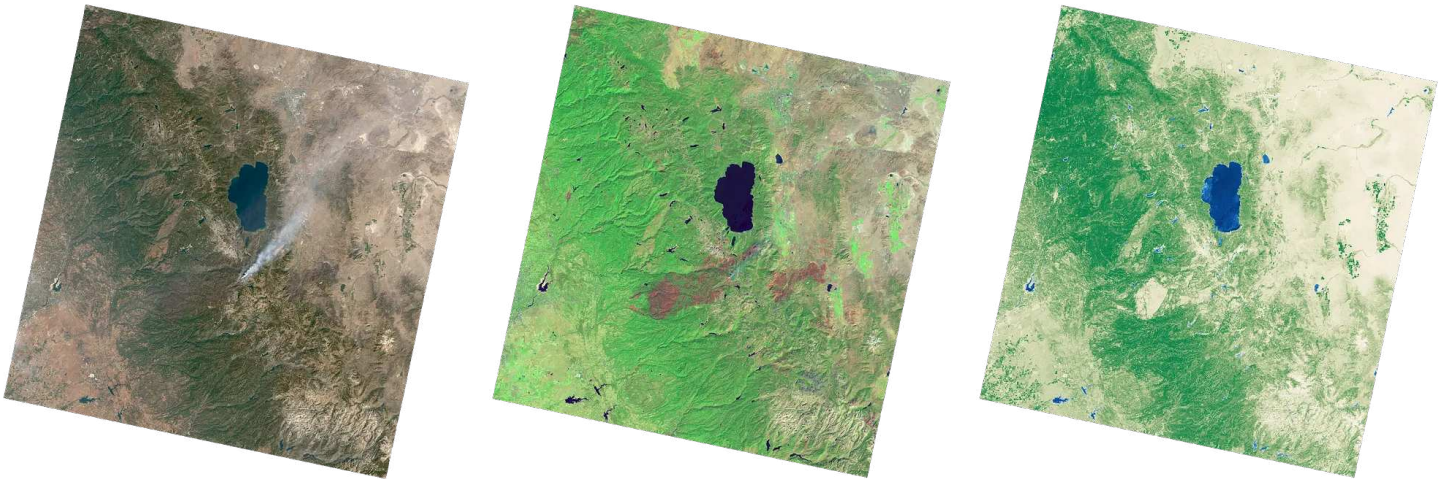


Landsat 1 Multispectral Scanner (MSS) false-color image of the Sierra Nevada included Yosemite National Park and Lake Tahoe, collected on September 6, 1974. Precisely calibrated data from the MSS allowed scientists to quantitatively map the surface of the Earth. Image by Robert Simmon based on data from the USGS Earth Explorer. CC-by-SA-4.0.

In my mind, calibration is what distinguished images from data, although there's no sharp boundary between the two. It's possible to make data from images, and images from data. Sometimes the conversion of data to image is reversible (i.e. one can derive the original calibrated data from the image), but more often the process is *lossy*, so the image can't be used to perfectly re-create the data it was derived from.

There's a few reasons for this. Human perception is — to put it mildly — imprecise, and the changes needed to make data interpretable to our eyes and brain inherently distort the underlying data. Plus, most image formats contain less information than scientific data formats. Sharpening, color-correction, bit-depth reduction, compression, limitations on dynamic range, and the small number of bands in some types of files can all result in the irretrievable loss of information. So imagery is usually a subset of the data it's based on.

Why is the collection of calibrated data from space so important? With calibrated measurements, it's possible to relate the amount of light (more accurately "electromagnetic radiation", although there are a handful of space borne remote sensing technologies that don't rely on light at all!) detected by a sensor to real-world "geophysical parameters".



These three visualizations of the same scene from Landsat 8 — true-color (red, green, blue), false-color (shortwave infrared, near infrared, green), and a map of vegetation and water quality — demonstrate the spectrum from "image" to "data". The true color scene was rendered from radiances to approximate what the human eye would see. The false-color image was derived from surface reflectance data, and uses two bands invisible to human eyes. The map combines normalized differential vegetation index (NDVI) and water quality index (WQI) both of which are numeric quantities that relate to properties of the Earth's surface. Images & map by Robert Simmon, based on data acquired on September 22, 2021, downloaded from the USGS Earth Explorer. CC-by-SA-4.0.

In other words, it's possible to go from the detection of varying amounts of light at different wavelengths (many of them invisible to humans!), to mapping things that help us better understand the world around us. Some of these parameters are

straightforward: vegetation health, cloud cover, air pollution, or surface temperature. Others are more exotic: energy balance, total ozone, turbidity, latent heat, evapotranspiration … there are literally thousands of different quantities being measured by NASA, other national and international organizations, and private companies all day every day.

Visualizing GeoTIFFs with GDAL

Which (finally!) brings me back to the topic at hand. How can GDAL — a code library mostly known for converting map projections and translating between file formats — help turn these massive archives of geospatial data into useful (and maybe beautiful) maps? If you read my previous installment in this series A Gentle Introduction to GDAL: Shaded Relief you hopefully noticed that GDAL's `gdaldem` program is not only capable of creating shaded relief from elevation data, but also mapping elevation to color. Not just elevation, but just about *any* type of georeferenced numeric data. In addition, GDAL has robust capabilities to read metadata and interpret a wide variety of scientific data formats. This unlocks some data sources that are, shall we say, *tricky* to access. (And yes, I am talking about HDF. You've been warned.)

I'll also confess that I stole this entire technique from Josh Stevens. (See Commanding Cartography: Take Control of Faster, More Elegant Workflows from the Command Line.) He was one of several people who dragged me (kicking and screaming) into modern cartography and data visualization workflows.

The basic pattern to use `gdaldem` to render data is:

```
gdaldem color-relief <input_dem> <color_text_file> <output_color_relief_map>
```

`<input_dem>` is the full name of the input data file (reminder — this doesn't need to be a DEM!)

`<color_text_file>` is a text file relating data values to colors. The format is data value, red value (from 0–255), green value (from 0–255), blue value (from 0–255),

with an optional value for alpha (also from 0–255, with 0 indicating fully transparent and 255 indicating fully opaque — for this to work you need to add the `-alpha` flag.)

`<output_dem>` is the full name of the output image file.

I'll work through an example using Daymet data — 2022 annual maximum and minimum temperatures for North America — downloaded from Oak Ridge National Laboratory. It's a nice dataset because it's derived "using inputs from multiple instrumented sites and weights for each site that reflect the spatial and temporal relationships of the estimation location to the instrumental observations". Which means minimal artifacts and no missing data — perfect for visualization.

The two file names are `daymet_v4_tmax_annavg_na_2022.tif` and `daymet_v4_tmin_annavg_na_2022.tif`. It's usually a good idea to get a sense of what's in the file before trying to render it with `gdalinfo`:

```
gdalinfo daymet_v4_tmax_annavg_na_2022.tif
```

There's a ton of output text, mostly focused on projection information, but I want to call attention to some of the information about the data itself (if you've heard the term "metadata" before, but didn't know what it means, it's data about data!):

```
Band 1 Block=256x256 Type=Float32, ColorInterp=Gray
  Min=-25.118 Max=37.871
  Minimum=-25.118, Maximum=37.871, Mean=9.832, StdDev=12.086
  NoData Value=-9999
```

Breaking it down:

`Band 1` indicates this information is about the first dataset in the file (in this case there's only one, but a true-color image would have at least three (red, green, and blue), a multispectral dataset could have four to a few dozen, and a hyperspectral

dataset would have hundreds). And you may have noticed that in this case Band 1 is *temperature* — it's not even a band at all! Another reminder that images and data are related, and often can be treated similarly, but aren't always the same thing.

`Type=Float32` denotes that the numbers that make up the data are formatted as 32-bit floating point value. This type of number can be positive or negative, integers or decimals, with about 7 significant figures. I'll cover some other common options later. Also keep in mind that some of these data types (floating point and signed 16 bit integers, in particular) can't be interpreted correctly by image processing programs (like Photoshop) and need to be translated to be displayed on a screen.

`ColorInterp=Gray` means that the data aren't assigned to a color channel or alpha channel, (which makes sense, since there's only one band). The data *could* be assigned to a color palette, like in a GIF or 8-bit PNG, but that's a bit of a special case.

`Min=-25.118 Max=37.871` are the maximum and minimum values in the dataset. These are important, since they'll help you pick end points when you apply a color ramp. If they're not in the displayed metadata, add the `-mm` flag, like this: `gdalinfo -mm daymet_v4_tmax_annavg_na_2022.tif` That will compute the values, but it can take a long time, since it needs to read every value in the entire dataset!

`Minimum=-25.118, Maximum=37.871, Mean=9.832, StdDev=12.086` more statistics. I'm honestly not sure why `Minimum` and `Maximum` are repeated. `Mean` (average) and `StdDev` (standard deviation) are both useful, but aren't needed to make the data into a picture.

`NoData Value=-9999` specifies the value for missing data. NoData represents ocean water in this particular dataset, but it could be bad data, missing data, cloudy data, or even land in an ocean dataset. It's important to remember that `NoData` is not the same thing as zero! So it usually gets its own color, or is even made transparent.

Phew, even stripping out all the geographic info and basics about the file, that was a lot! But it's necessary to define the relationships between color and value, which is what the `<color_text_file>` contains. Here's what I used for this particular dataset, contained in a file named `daymet_v4_temperature_palette_-20_40.txt`:

```
-9999,0,0,0,0
-100,69,117,180,255
-20,69,117,180,255
-10,145,191,219,255
0,224,243,248,255
10,255,255,191,255
20,254,224,144,255
30,252,141,89,255
40,215,48,39,255
```

It follows the same format as I described in A Gentle Introduction to GDAL Part 5: Shaded Relief. Except in this case instead of `elevation` or `aspect` (which is itself derived form elevation) the `value` is average max temperature. And the same pattern works regardless of dataset, as long as the data is a range of numbers:

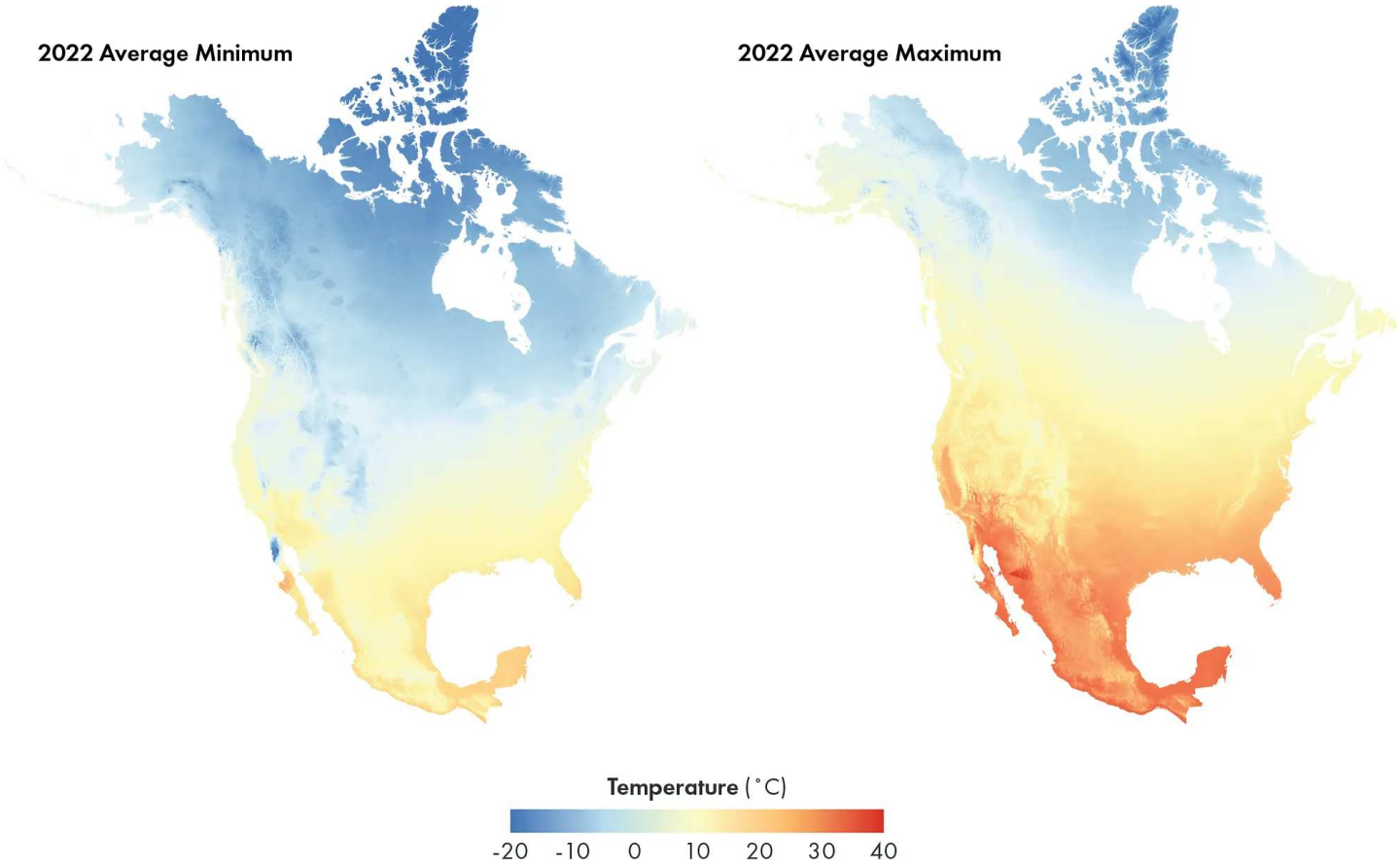
`value,red,green,blue,alpha` (alpha is optional)

In the case of the elevation data in my previous post, the file was a nice perfect square with no missing data, so figuring out how to represent `NoData` or assigning a value for alpha wasn't important. This example, however, has *plenty* of missing data points. I could assign those missing data to a color (preferentially a neutral color that's distinct from any of the colors on the color ramp), but it's more elegant to make them transparent. That's indicated by the `0` in the fifth column of the first row. All the other values have an alpha of `255` — fully opaque (like the red, green, and blue channels, the alpha channel is represented by an 8-bit number, which has a possible range from 0 to 255).

A few other subtleties I should explain. There's a line for temperatures well below the minimum value in the file: `-100` which has the same color assigned to it as `-20`. (Both are red = 69, green = 117, blue = 180, alpha = 255; a dark blue that's opaque.) If that additional, lower value *wasn't* there, GDAL would try to interpolate from that dark blue to the black assigned to `-9999`, since the algorithm doesn't take into account that `-9999` really means "nothing to see here". There's a small tail of temperature data below the -20˚ C that I want as the minimum of the range, and that

extra line makes sure it appears correctly.

Here's what that color ramp looks like applied to two DAYMET datasets: average minimum daily temperature (left) & average maximum daily temperature (right) for 2022 over North America.



2022 average minimum (left) and maximum (right) temperatures. Maps by Robert Simmon based on data from DAYMET, Oak Ridge National Laboratory. CC-by-SA-4.0.

Yet another aside: these maps use a palette (slightly modified from the ColorBrewer `RdYlBu` scheme) usually reserved for *divergent* datasets — datasets that are positive and negative (electric charge), or otherwise have a departure from a central value (the difference of one year's rainfall from average). Temperature has a continuous range from low to high, so could be suitably displayed by a sequential palette, which goes from light/pale to dark/saturated (or vice-versa) instead of having a light/neutral color in the middle with dark/saturated colors on either extreme. The trouble with using a sequential palette here is that the lighter values are usually de-emphasized, and viewers are typically going to be interested in **both** cold and hot

extremes. So the divergent palette works. Plus people often have a strong association with "cool" (green and blue) and "warm" (yellow, orange, and red) colors, so palettes that break those expectations can be confusing for viewers.

And y'know what? Rules are meant to be broken. Just do so with a clear idea of *why* you're breaking them.

The code to generate both images is almost identical, with the only difference being the input and output filenames. (And make sure you specify different output file names! Because `gdal_dem` will *happily* overwrite your files with no warning. Batch processing a large stack of humungous files and realizing they were all named `you_forgot_to_change_the_output_file_name_dummy.tif` is … unpleasant.)

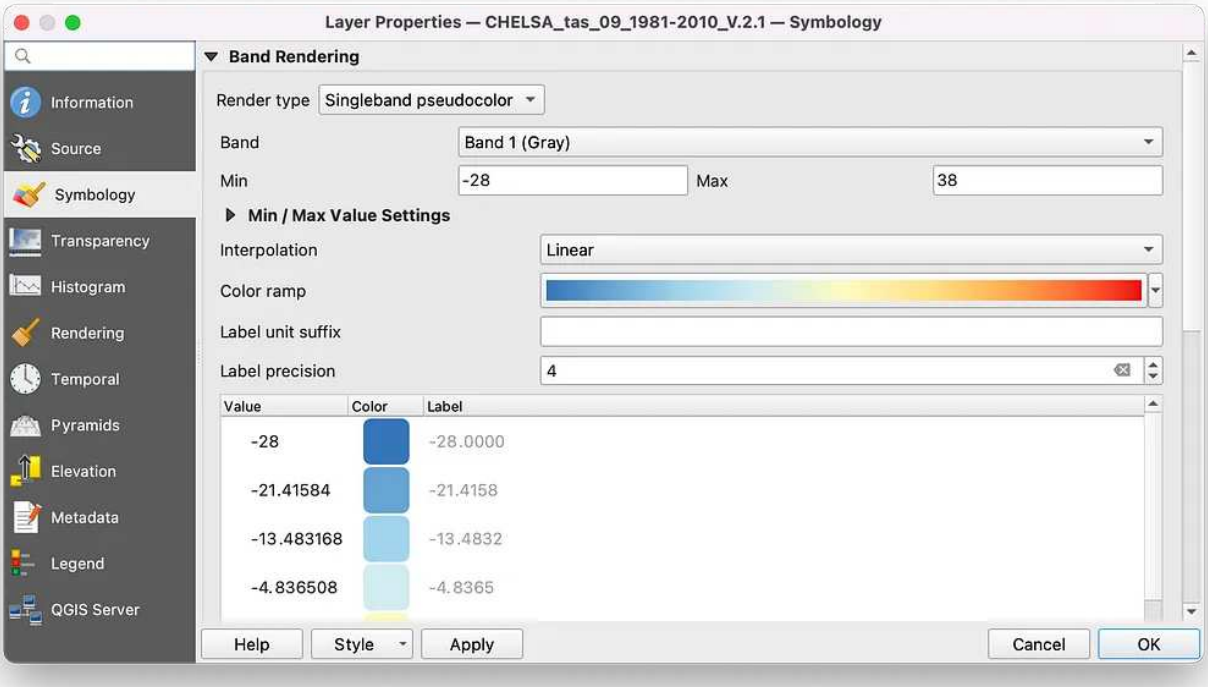
```
gdaldem color-relief -alpha daymet_v4_tmax_annavg_na_2022.tif  daymet_v4_temper

gdaldem color-relief -alpha daymet_v4_tmin_annavg_na_2022.tif daymet_v4_tempera
```

(**A few notes:** code blocks in Medium don't wrap, so copy/paste into a text editor or terminal or scroll right to see the full command. I also include these potentially cryptic "creation options" in almost all of my GDAL examples: `-co COMPRESS=DEFLATE -co PREDICTOR=2`. Creation options are specific to writing particular file types in GDAL, in this case TIFFs. These are telling GDAL to write the TIFFs with an efficient lossless compression algorithm to save drive space, at the expense of processing time. (Koko Alberti wrote a very thorough explanation of GeoTIFF compression options, if you're curious.))

Another confession — I didn't figure out the colors and ranges for these maps by looking at the minimum and maximum values spit out by `gdalinfo` and editing a text file. I did that part of the work in QGIS, which is more interactive than the command line (there's plenty of other options, too, like Photoshop with Avenza Geographic Imager, ENVI, or ArcGIS). However, once I decided on the color ramp and range, I used GDAL. If you know what you want, it's faster, less error prone, and more repeatable to execute a bit of code than it is to manually punch in numbers

and edit file names.



Screen shot of QGIS showing assignment of colors to data values in the symbology sub-menu of the layer properties dialog box (there's *a lot* of options available in QGIS...). It's often more efficient to prototype color palettes in a GUI than to switch between writing code and previewing reasults. GDAL (or another programmatic approach), on the other hand, is better for repeatability.

That's the basic workflow for visualizing data with GDAL:

- Read the metadata with `gdalinfo` to determine maximum and minimum values
- Write a text file that maps values to colors: `value, red, green, blue, alpha`
- Run `gdaldem` with `color-relief`

When Things Get Weird: Working with Scale and Offset

With GeoTIFFs, that's usually straightforward, but there can be curveballs.

Here's an example from another dataset: global September air temperature for 1981–2010 from Climatologies at High Resolution for the Earth's Land Surface Areas (CHELSEA). (Download here.) First, the metadata from gdalinfo:

```
Band 1 Block=43200x1 Type=UInt16, ColorInterp=Gray
Min=2100.000 Max=3103.000 Computed Min/Max=2099.000,3104.000
Minimum=2100.000, Maximum=3103.000, Mean=2801.645, StdDev=212.587
NoData Value=-2147483647
Offset: -273.15, Scale:0.1
```

Which seems … fine, but the values don't look quite right for temperature (thousands of degrees?) and there's a new line: `Offset: -273.15, Scale: 0.1` What's that all about? There's another clue, too: `Type=UInt16`.

Any ideas?

There's several reasons for the odd scaling of the data. First is that the data are in `UINT16`, an integer format (no decimal points), so to get a precision of a tenth of a degree the data are stored as 10 times the original value. That means they need to be multiplied by 0.1 when the data is read back in. The second reason is that `UINT16` is an *unsigned* integer, which means no negative values. Since there's lots of places in the world where's it below 0˚C the data need to be offset so that negative data can be stored correctly in the chosen format.

Which dovetails nicely with the fact that satellite temperature data is generally measured in kelvins (degrees above Absolute Zero). (It's a long story, but satellite temperature estimates are made indirectly, by measuring the *brightness temperature* of a surface. Which itself relates to the blackbody radiation emitted by the surface, (mostly) determined by its temperature above Absolute Zero.) That's what the `Offset` parameter is for: 0˚ in Celsius = 273.15 Kelvin. It's necessary to subtract 273.15 from the data to convert from Kelvin to Celsius (after dividing by 10, of course).

I hope that all made sense, because if you want to display the data for an audience that's more comfortable in ˚C (I'm not even going to get into ˚F) than K, either the palette or the data needs to be scaled first. Further complicating things, some software (like QGIS) automatically applies scale and offset when displaying datasets

with that information in the metadata. Which is great if your workflow ends in QGIS, but not so great if you've used QGIS as a preview tool and are going back to GDAL to visualize the data.

Doing math on every pixel of a large dataset (43,200 by 21,600 pixels in this case) can take a while, so it's generally more efficient to translate the units in the palette file rather than modifying the underlying data. For a small number of value-color pairs it's not too hard to do it by hand or in a spreadsheet, but Python's a good tool for reading a text file, converting that to numbers, doing some calculations, converting back to text, and writing a new file. I'm not going to explain this in detail (although I might in the future ...) but here's what my program looks like:

```
# open original palette, in ˚C
palette_c = open("ast_palette_RdYlBl.txt", "r")

# create an empty list to store the data
palette_k = []

# read the original palette into the list as floats & convert from ˚C to K
for line in palette_c:
    line_clean = line.strip()
    line_list = line_clean.split(",")
    line_list[0] = str((float(line_list[0]) + 273.15) * 10)
    palette_k.append(line_list)

# close the data object pointing to the original file
palette_c.close()

# create a destination file
outfile = open("palette_c_gdal.txt", "w")

# write each line of the modified palette into the file plaette_k.txt with newl
for palette_line in palette_k:
    row_string = '{},{},{},{}'.format(palette_line[0], palette_line[1], palette
    outfile.write(row_string)
    outfile.write("\n")

# close the outfile data object
outfile.close()
```

Hopefully the comments are sufficient to explain what's going on (familiarity with Python will help).

That prgram will convert this:

```
-28,69,117,180
-22,111,159,205
-16,153,196,225
-10,189,222,236
-4,219,237,237
2,243,246,212
8,255,243,182
14,254,227,149
20,253,195,116
26,250,153,87
32,241,104,65
38,215,48,39
```

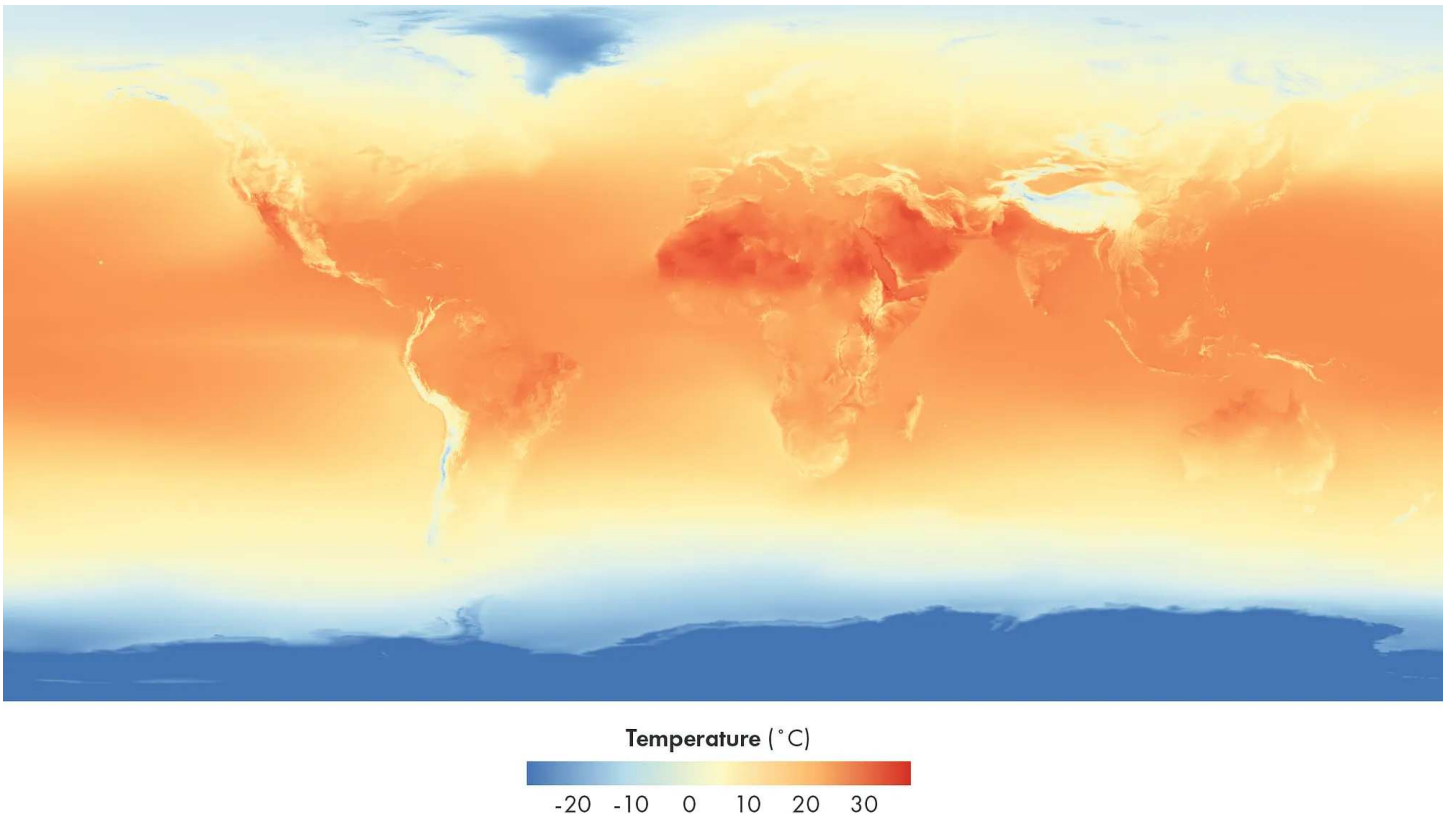
To this (the only difference is the first column, converting from degrees Celsius to kelvins):

```
2451.5,69,117,180
2511.5,111,159,205
2571.5,153,196,225
2631.5,189,222,236
2691.5,219,237,237
2751.5,243,246,212
2811.5,255,243,182
2871.5,254,227,149
2931.5,253,195,116
2991.5,250,153,87
3051.5,241,104,65
3111.5,215,48,39
```

All that's left is to run the GDAL command (remember to copy/paste to see the whole line):

```
gdaldem color-relief CHELSA_tas_09_1981-2010_V.2.1.tiff palette_k_-22_38_gdal.t
```

Here's the result (with the color key added later):

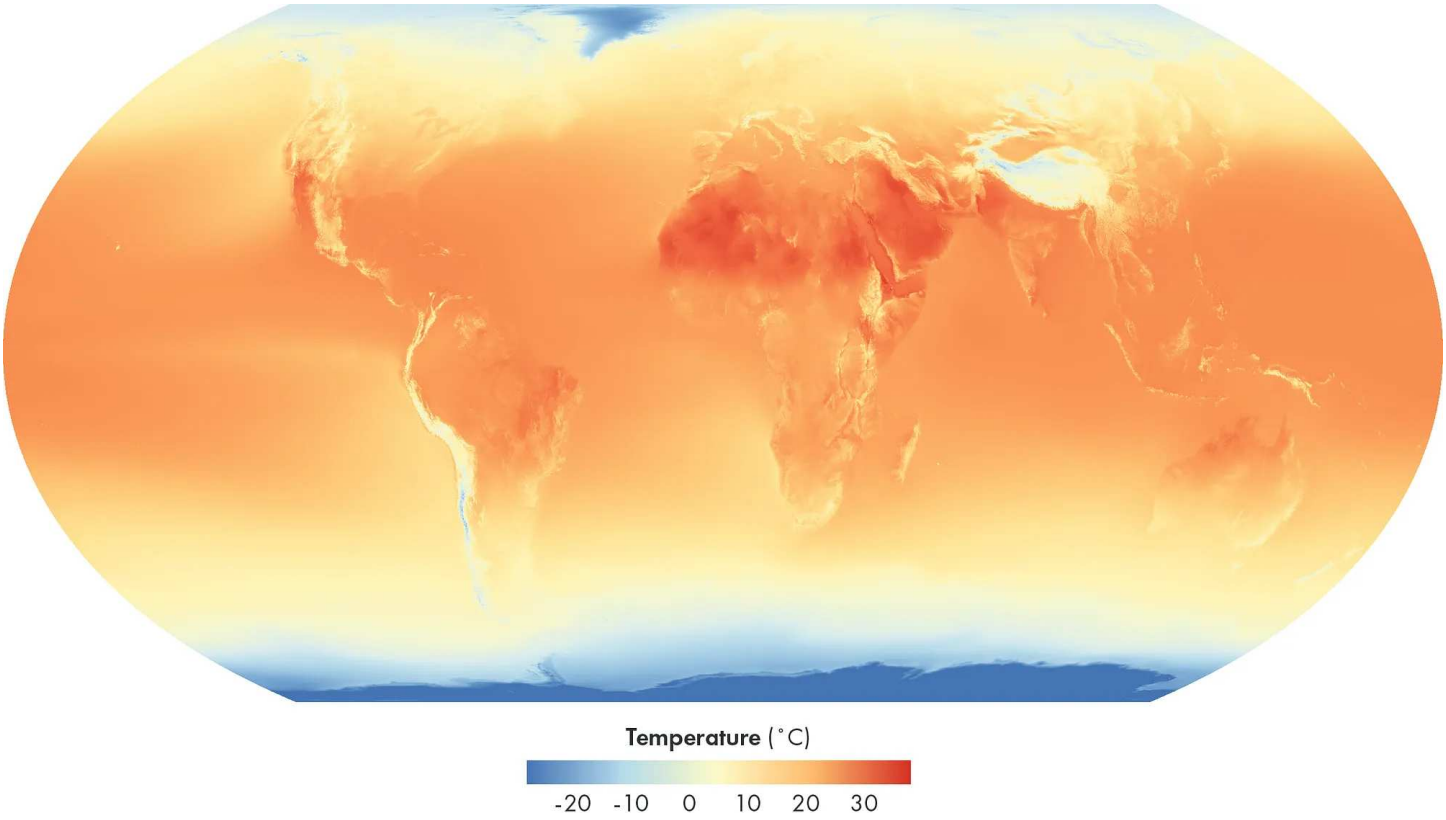


CHELSEA global average temperature, September 1981–2010 in equirectangular projection. Lots of global datasets are stored in this projection (it creeates a nice rectangular array of evenly spaced latitudes and longitudes), so consider reprojecting the data if appropriate for your needs.

Oh, and one more thing — the data are stored in a equirectangular projection (a regular grid that spans ±90˚ latitude and ±180˚ longitude). Great for importing into a 3D program like Blender or Maya, not so great for displaying as a map. High latitudes (near the North and South Poles) are stretched horizontally, making them appear to cover more area than mid and equatorial latitudes. For global thematic data an equal area projection like Equal Earth is a better choice. You can convert a map into Equal Earth by using `-t_srs '+proj=eqearth'` with the `gdalwarp` program. (If you're not familiar with `gdalwarp` I explain the basics in part 2 of this series: Map

Projections & gdalwarp.)

```
gdalwarp -t_srs '+proj=eqearth' -r bilinear -dstalpha CHELSA_tas_09_1981-2010_V
```



Equal Earth variation of the CHELSEA September global average temperature dataset. A side effect of switching to an equal-area projection is to reduce the visual impact of Antarctica, almost all of which has average temperatures lower than the minimum shown with this color scaling.

To reiterate, my workflow with this data was:

1. Read the metadata with `gdalinfo`
2. Preview and decide on the scaling of minimum and maximum values in QGIS (which automagically converted from Kelvins to degrees Celsius based on the scale and offset values in the metadata)
3. Create a text file mapping Celsius values to colors
4. Convert temperature in Celsius to temperature in kelvins with Python

5. Finally, generate the image with `gdaldem color-relief`

Somehow this got long, and I'm only about halfway through the examples I wanted to show — so I'm going to call this "A Gentle Introduction to GDAL Part 6.1" and continue with Part 6.2 after the 2023 NACIS meeting. That will cover some of the ways to convert imagery into data with `gdal_calc.py`, dealing with HDF and NetCDF, a very brief introduction to using GDAL with Python, and maybe a few surprises.

1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp
3. Geodesy & Local Map Projections
4. Working with Satellite Data
5. Shaded Relief
6. Visualizing Data Part 1 (you are here)
7. Transforming Data

Thanks (again, still) to Frank Warmerdam, who got me over my fear of GDAL, Joshua Stevens for describing how to use gdaldem on non-elevation data, and Joe Kington for being a sounding board.

Cartography | Data Visualization | Maps | Satellite Imagery | Programming

Follow

A Gentle Introduction to GDAL Part 7: Transforming Data

Robert Simmon · Follow

18 min read · Oct 30, 2023

The ability to color-code geographic datasets is a key reason to include GDAL in a visualization workflow . It's great for batch processing a time series or automating production of graphics in near-real-time. But GDAL isn't limited to making pictures of data. Tools like `gdal_calc.py` help transform data from a raw material into usable information. Information that forms the basis for analysis (and mapping).

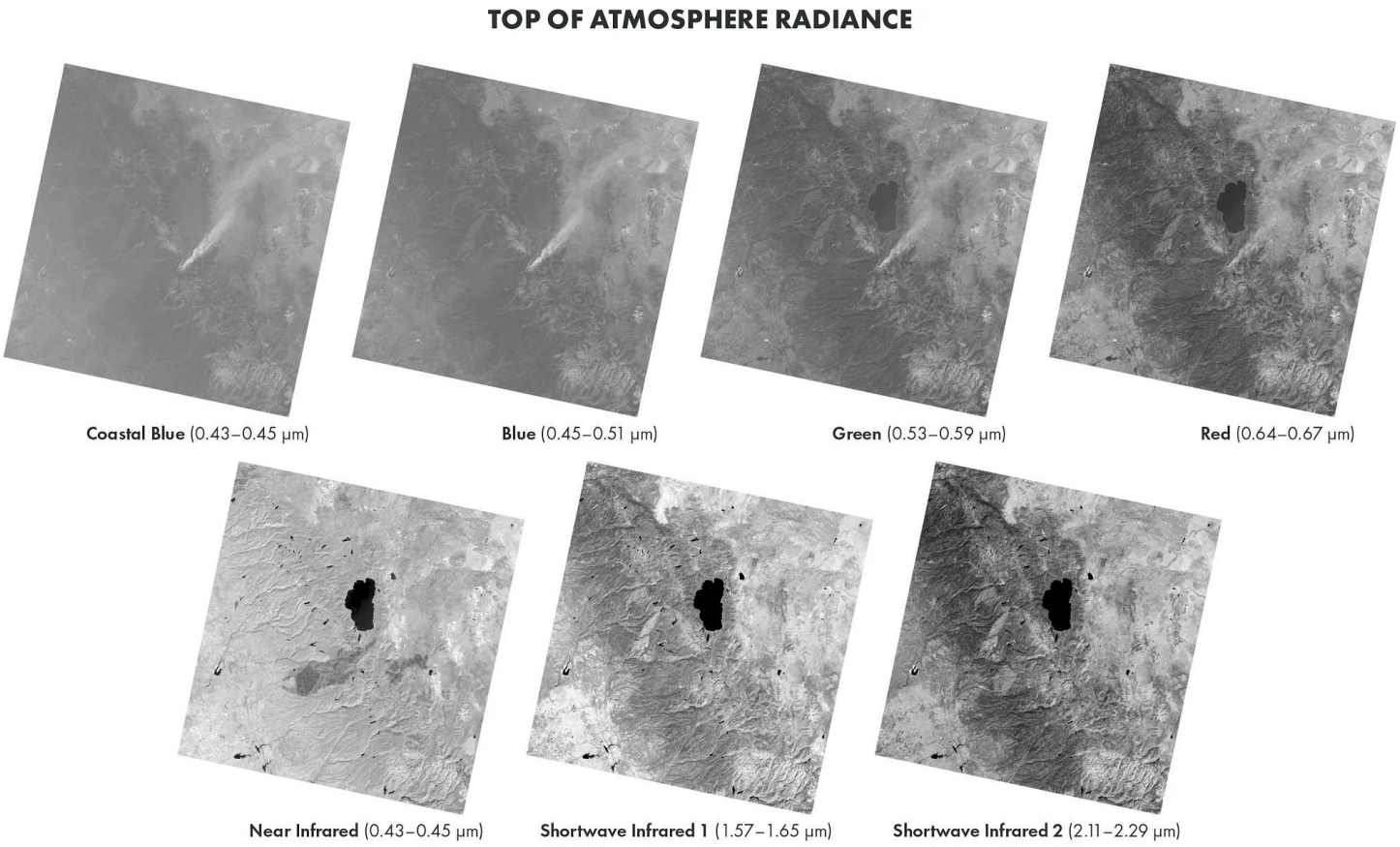
End users of remote sensing data rarely want imagery — they want answers from the information contained within the imagery (and other more exotic remote sensing techniques, but this post is mainly about deriving data from imagery). Known variously as geophysical parameters, planetary variables, analytics, or insights; derived datasets form the heart of why remote sensing is so valuable for Earth science. They help us understand the past, react in the present, and plan for the future.

In this article I'll demonstrate a few methods for using GDAL to transform data, helping make the information it contains more usable. I'll show how to calculate spectral indices from individual bands, how to mask out invalid data (like water in a vegetation index), and how to combine datasets into multivariate maps — along with some background on remote sensing.

An Introduction to Spectral Indices

One key technique to go from numbers to useful information is a *spectral index*. The

technique uses the relationship between the brightness of pixels in different spectral bands to derive a quantity that relates to the physical world.



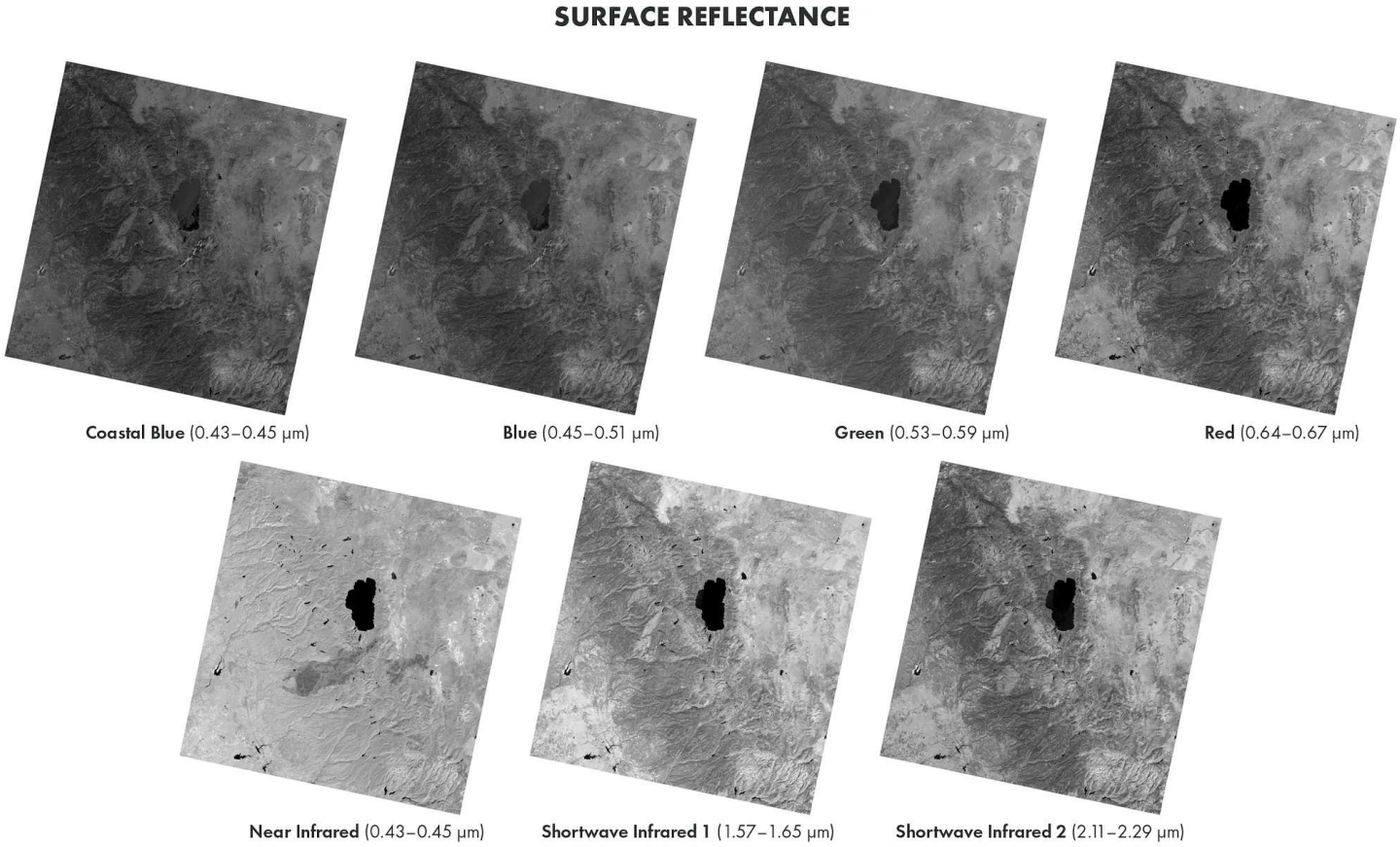
These images show each of Landsat 8's seven multispectral bands. Radiance data — while calibrated — varies based on the brightness of the Sun at different wavelengths and the state of the atmosphere. The shorter the wavelength the greater the effect of atmospheric scattering, from both nitrogen and oxygen molecules, trace gases (in particular water vapor and ozone) and particulates like smoke. Longer wavelengths pass more directly through the atmosphere. Notice how the smoke from the Caldor Fire (just south of Lake Tahoe) nearly disappears in the two shortwave infrared bands. (Data collected on September 22, 2021. Available on the USGS Earth Explorer.)

The images above show the Operational Land Imager's (the primary instrument on Landsat 8 and 9) seven multispectral bands (excluding the panchromatic band — which is a blend of visible wavelengths, and the thermal infrared bands which are collected by a different instrument and measure energy emitted from the Earth — not reflected sunlight). Each image represents the perspective from the top of the atmosphere (often abbreviated as TOA) in a different color (including three infrared "colors" that are invisible to us puny humans). I've scaled them to be roughly equivalent, rather than stretched for maximum contrast, which is a more typical display technique. (I cheated a bit and brightened the two shortwave infrared bands,

because the Sun is much brighter in visible wavelengths.)

Looking at the bands next to each other, you might get some clues as to what features each band is sensitive to, and how comparing bands in various ways might reveal some insights about the Earth's surface. Water is dark in near and shortwave infrared light, but bright in coastal blue. Vegetation is bright in near infrared but dark in all the other bands. Smoke is transparent in shortwave infrared. The images taken with blue light are washed out because the short wavelengths are scattered by the atmosphere.

That last fact is important, since observers are often (but not always) interested in properties of the land and water, and the atmosphere interferes in a way that's dependent on wavelength. Therefore most algorithms require surface reflectance — rather than digital numbers or radiance — as inputs. Surface reflectance is a measure of the proportion of light reflected from the Earth's surface, but it can be tricky to calculate. Fortunately most providers of satellite data make a surface reflectance product available so it's not something you'll usually need to worry about.



Grayscale images of surface reflectance data from each of the seven multispectral bands on Landsat 8. Each has been processed identically, so they show how various surfaces reflect light differently in each wavelength. Notice how the landscape varies based on wavelength. These differences form the basis of the algorithms that quantify surface properties. (Data collected on September 22, 2021. Available on the USGS Earth Explorer.)

The images above show the same Lake Tahoe scene as before, but the surface reflectance of each band rather than top of atmosphere radiance. Each band is scaled from 0 to 50% reflectivity, so they can be directly compared with one another. Equal brightness in the different images indicates the same proportion of light is being reflected in each different bands. Surface reflection measurements also help ensure observations taken in different locations or at different times are analogous — important for time series analysis or comparisons across regions.

With surface reflectance data, surface properties — vegetation health, burn severity, water quality, soil conditions — can be derived by comparing the relative reflectance of a pixel at different wavelengths. The resulting parameters are called spectral indices, and transform satellite data from an image to a quantity linked to real world physical properties.

Notice how I'm being slightly vague in this description — many spectral indices produce a number that is related to properties of the Earth's surface, but *unitless.* So the indices are not absolute measurements of a physical property, but a relative measurement. It's a bit of a strange concept. In practice, indices are a good way to tell if there's more or less of something, but not necessarily exactly how much more or how much less. (And yes, I'm still being precisely imprecise with my language.)

The results of indices can also vary based on small differences between sensors, specifically the wavelengths of light measured by each band, which can vary from mission to mission. So the exact same index derived from Sentinel-2, Landsat 9, or a commercial sensor may give different results. The challenge is significant enough that both NASA (Harmonized Landsat and Sentinel-2) and the ESA (Copernicus Sentinel-2 MSI Level-2H and Level-2F) have programs to make Sentinel-2 and Landsat data consistent with each other. Just something to be aware of if you're trying to compare indices across multiple instruments.

Yet Another NDVI Tutorial (Using gdal_calc.py to Calculate a Spectral Index)

OK, I know — you're probably already familiar with the equation to calculate Normalized Difference Vegetation Index (NDVI), and you can get pre-baked NDVI from just about any Earth observing mission under the sun, so why bother? First of all, because NDVI is among the simplest algorithms out there, so it's a good place to introduce `gdal_calc.py`, a program that enables you to use the mathematical capabilities of numpy through GDAL. (I'll explain why that's powerful soon.) NDVI is also ubiquitous, with a history going all the way back to the earliest Landsat missions, so it's a good place to start. I promise things will get more interesting.

Here's the general flow of the process to calculate a spectral index and visualize it with GDAL:

1. Get data
2. Determine the equation(s)
3. Convert into reflectance (if necessary)
4. Execute calculations with `gdal_calc.py`
5. Apply color table to convert data to an image with `gdaldem` (which I described in detail in Part 6: Visualizing Data)

I'm going to illustrate how to calculate NDVI with Landsat 8 surface reflectance data of Lake Tahoe and its surroundings. The same data I've already shown, which was collected on September 22, 2021. The data are available from the USGS Earth Explorer. If you're unsure how to download Landsat data from the USGS, I wrote a tutorial on Earth Explorer that's still more-or-less correct, but you'll want to choose `Landsat > Landsat Collection 2 Level 2 data > Landsat 8-9 OLI/TIRS C2/L2` under `Select Your Data Set(s)` instead of `Landsat Archive > L8 OLI/TIRS`. You'll only need the surface reflectance bands — I won't be using surface temperature data at all.

If you want to work with another area, feel free to find a location that's interesting to you. Just pick someplace with both plants and water, since I'll be demonstrating vegetation and water quality indices. Make sure the data are from Landsat 8 or 9 — a few things I discuss will be specific to the Operational Land Imager, so older

Landsats or Sentinel 2 would be more of an extra credit project. You'll also need both GDAL and Python installed. (Instructions in A Gentle Introduction to GDAL Part 5.)

Calculating NDVI

Here's the equation for NDVI:

```
NDVI = (Near Infrared - Red)/(Near Infrared + Red)
```

Conceptually simple, but there's a little bit of complexity introduced by working with real world satellite data.

Here's the GDAL command (code blocks don't wrap in Medium, so you'll need to copy/paste into a text editor to see the whole thing.):

```
gdal_calc.py -D LC08_L2SP_043033_20210922_20210930_02_T1_SR_B4.TIF -E LC08_L2SP
```

Unfortunately, it's not nearly as simple as the plain NDVI equation. I'll break down the parts, and explain why there are some extra bits.

The first part of the command assigns files to the variables used for the calculation:

```
-D LC08_L2SP_043033_20210922_20210930_02_T1_SR_B4.TIF -E LC08_L2SP_043033_20210
```

Each band of input data needs to be linked to a letter (either uppercase or lowercase) and that letter is used as shorthand in the part of the code that describes the calculation. I assigned the red band on Landsat 8, `B4` to `D` and the near infrared band `B5` to `E`. I chose "D" and "E" since they're the 4th and 5th letters in the alphabet, corresponding to the 4th and 5th Landsat 8 bands. Easier to remember than something arbitrary, at least to me.

```
--calc="((E * 0.0000275 - 0.2) - (D * 0.0000275 - 0.2))/((E * 0.0000275 - 0.2)
```

Is the calculation itself, called by `--calc` and enclosed in double quotes. The operations are performed on the bands defined previously, and can be any function available in NumPy. The NDVI calculation uses simple arithmetic (only addition `+` subtraction `-` and division `/`), but having access to trigonometric functions, logarithms, and other mathematical operation unlocks a whole bunch of interesting possibilities. I won't explore them much in this post, but maybe later.

The calculation is complicated by the fact that Landsat surface reflectance data are stored as unsigned integers (`Type=UInt16` if you check with `gdalinfo`) not floating point numbers. So there is a `scale` (0.0000275) and `offset` (0.2) applied to each value in the file. This mathematical trick allows fractional reflectance values that range between 0 and 1 to be squeezed into a file format that only accepts whole numbers from 0 to 65535. The USGS explains this in more detail in a FAQ.

(The additive offset may look a little weird, but it allows room for negative reflectances that don't exist in the real world, but can be generated if the surface reflectance correction overestimates the scattering effect of the atmosphere. Something that's relatively common over dark surfaces, like vegetation and water.)

The calculation is significantly simpler if the data don't need to be re-formatted, in which case the operation would look like this:

```
--calc="(E - D)/(E + D)"
```

Back to the straightforward NDVI equation (remember `D` is the red band and `E` is the near infrared band)! As with most computer languages and mathematical notation, the parentheses are needed to force addition and subtraction to be

performed before division.

The rest of the command specifies the output file name `--outfile=tahoe_LC08_20210922_SR_NDVI.tif` and forces the file to be written as floating point numbers `--type=Float32`. (By default `gdal_calc.py` will write the output file in the same format as the input files. Which is a problem, because writing floating point numbers into an unsigned integer file results in unintelligible garbage. So you need to be explicit about the data type.) Finally `--co COMPRESS=DEFLATE --co PREDICTOR=2` executes lossless compression (note that there's a double dash because this is sending arguments to a python program, rather than running GDAL directly).

Making an Image from the Index

The output file will have NDVI values stored as floating point numbers in a single-channel grayscale GeoTIFF. If you want something for display in color you'll still need to visualize the data with `gdaldem color-relief`.

The general workflow for this method is to convert a single band data file into a full-color image by correlating the data values with red, green, and blue color values defined in a text file. Read A Gentle Introduction to GDAL Part 6: Visualizing Data for more info.

Here is the command using the NDVI data created by `gdal_calc.py`:

```
gdaldem color-relief tahoe_LC08_20210922_SR_NDVI.tif ndvi_landsat_palette_beige
```

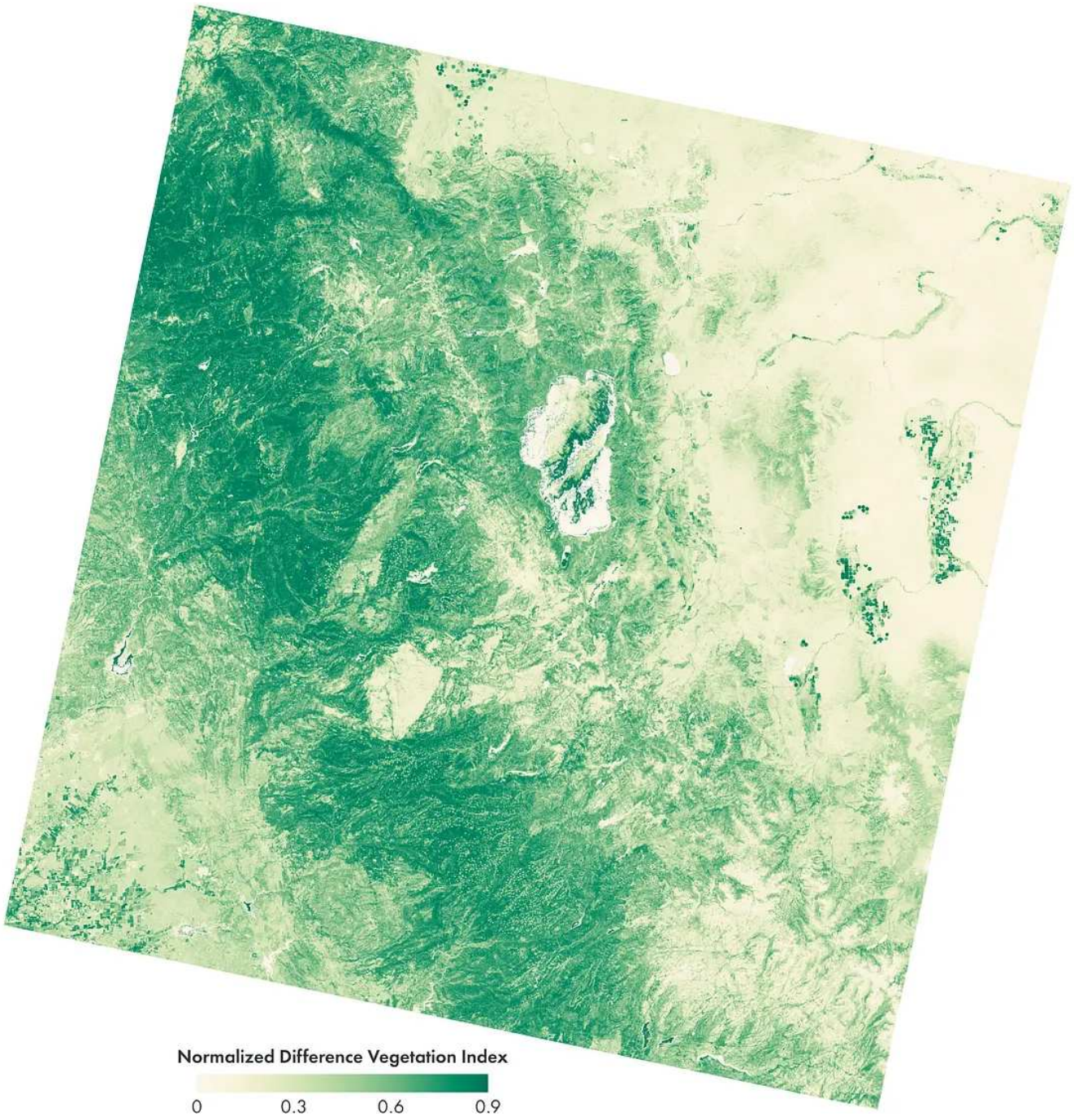
And here are the contents of the text file that correlate values to colors. (Note that this file will also create an alpha channel based on a "no data" value of `-9999` — this ensures the area outside of the boundaries of the Landsat scene will be transparent.)

```
-9999,0,0,0,0
-9998,251,246,237,255
0,251,246,237,255
0.1,249,244,220,255
0.2,236,238,203,255
```

```
0.3,216,229,185,255
0.4,189,218,166,255
0.5,156,203,149,255
0.6,116,185,132,255
0.7,62,164,117,255
0.8,0,141,102,255
0.9,0,116,89,255
1,0,116,89,255
```

I created this beige-green gradient with HCL Wizard, my current favorite perceptual palette generator. (Which is a good reminder that the "tools" section of Subtleties of Color needs an update ...)

And here's what it looks like (with an added color key):

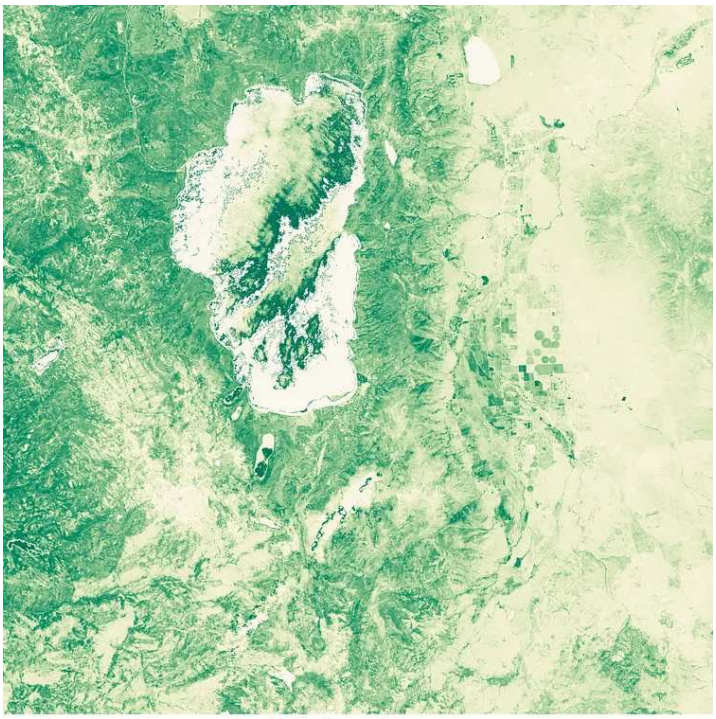


Normalized Difference Vegetation Index (NDVI) calculated from red and near infrared surface reflectance data with gdal_calc.py. Dark green represents thriving vegetation. Light green represents sparse vegetation, while beige is bare land or fresh burn scar. Water has not been masked. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

I think the beige to green color palette makes the landscape fairly easy to read. Bare granite of the High Sierra peaks is light beige. Sparsely vegetated desert lowlands in Nevada (upper right) and the Caldor Fire burn scar are both a little bit darker and greener. Dry grasslands in the Central Valley (lower left) are greener still. High

altitude forests along the Sierra Crest (upper left to lower right) are a medium green; while wetter, lower elevation forests on the Sierra's West Slope are dark green. Fields in Nevada and California are the darkest green, representing dense vegetation nourished by irrigation and fertilizer.

Look closely, and one discrepancy stands out — NDVI values for water range wildly from below zero to above 0.9 (outside the bounds of the color palette) seemingly at random. In fact, The variability is so high it's best to flag water as "no data" in an NDVI map, replacing it with either a fill value or more meaningful measurement. To understand why, and to find a replacement, it's worth delving a little more deeply into how NDVI works.



Unmasked NDVI (left) compared with a true color image (right) of Lake Tahoe. NDVI measurements of water bodies can vary in unpredictable ways. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

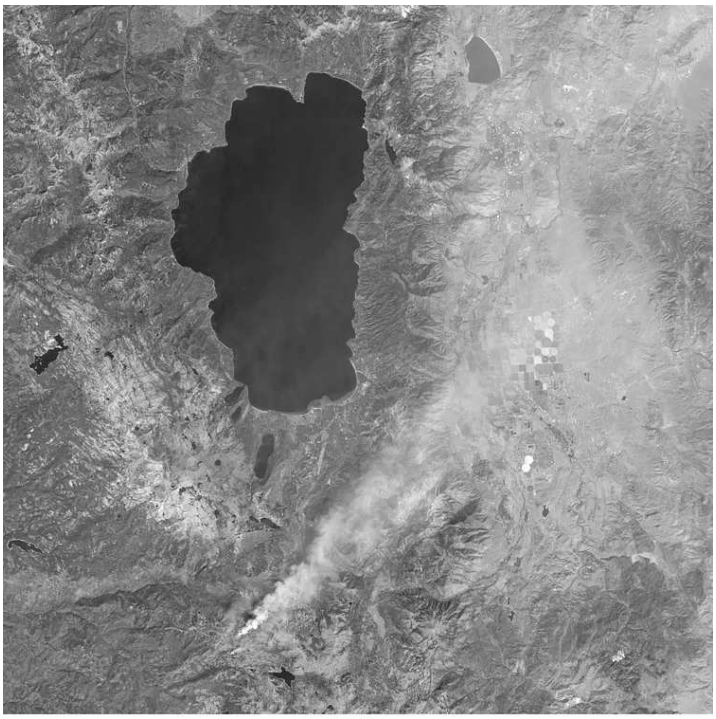
NDVI relies on two properties of vegetation — the strong absorption of visible light (particularly red light) by chlorophyll, and the strong reflection of near infrared light by the cellular structure of leaves. High NDVI values (over 0.5 or so) occur when healthy leaves cover a large fraction of any given pixel. Lower NDVI values can be due to leaves with relatively low amounts of chlorophyll (unhealthy or senescent

leaves) *or* sparse vegetation. The algorithm can't distinguish between lots of unhealthy leaves or a lack of leaves.

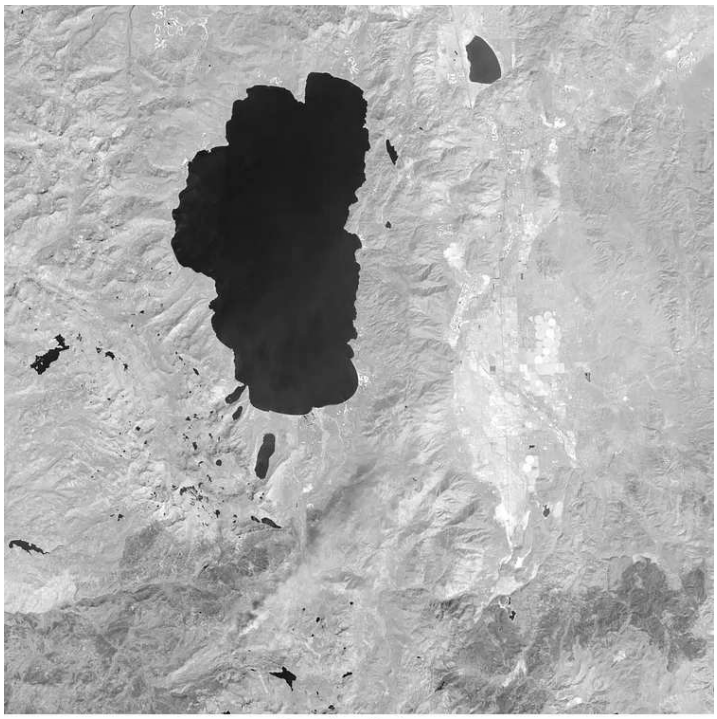
Although mathematically NDVI can range from -1 to +1, the realistic range for pixels with at least some vegetation varies from 0 to 0.8 or so (the precise minimum and maximum values depends on the attributes of the sensor).

If there's no vegetation at all NDVI *isn't really meaningful.* In general, natural vegetation-free surfaces (sand, bare rock, exposed soil) reflect similarly throughout the visible and near infrared spectrum. So NDVI will tend to be near zero, or even slightly negative — but the exact value of NDVI doesn't have physical significance in the absence of vegetation.

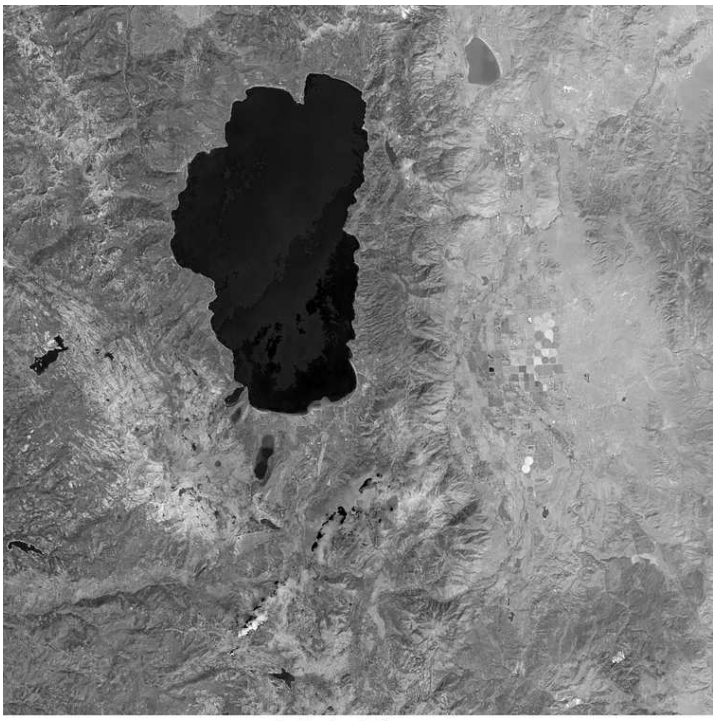
Unless there's water present. Clear water absorbs light more efficiently with longer wavelengths, so gets darker from blue to green to red to near infrared. As a result, under ideal conditions NDVI will be negative for water pixels. *If* the water is clear. Water with sediment in it (during a flood, or spring melt, for example) or with algae growing near the surface, can be bright in near infrared, and often exhibits positive NDVI. Worse yet, under certain conditions sunlight will reflect directly into the sensor taking measurements (a phenomenon called sunglint), effectively blinding it. All of this makes it difficult to confidently separate water from land pixels only using NDVI.



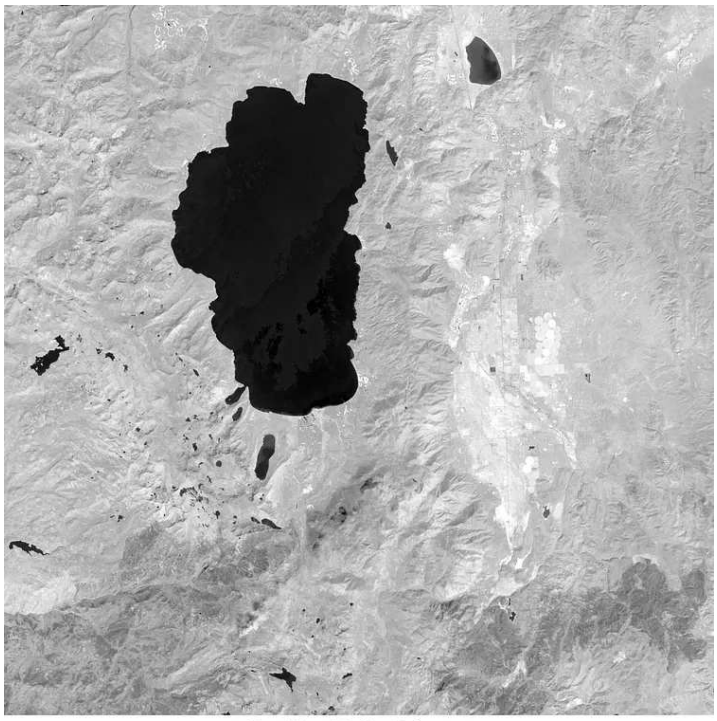
Red Radiance



Near Infrared Radiance



Red Surface Reflectance



Near Infrared Surface Reflectance

These four images compare radiance (top row) and surface reflectance (bottom row) in the red (left) and near infrared (right) bands of Landsat 8. Contrast between land and water is higher in near infrared than red, so NDVI (which uses red and near infrared light) yields some information about where there's water. But it's far from perfect, and particularly susceptible to mis-identifying water when there's a lot of sediment present or there is sunglint on the surface. In the radiance images the lower-right portion of Lake Tahoe is lighter than the upper-left, which suggests some sunlight is reflecting off the surface into the sensor. The surface reflectance algorithm isn't adequately compensating for the change in brightness, which results in the relative lightness flipping (areas in sunglint are darker) in the surface reflectance data. This is responsible for the extreme variablilty in NDVI apparent

in Tahoe. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

Put another way — NDVI is sensitive to the presence of water, but there's no threshold value that separates "water" from "not water". At least if you're limited to measurements in the visible and near infrared wavelengths. Water absorbs shortwave infrared (SWIR) *extremely* efficiently — so well that mud or wet sand can be nearly black in Landsat's SWIR bands. Algorithms that incorporate this extra information have a much better chance of accurately detecting water.

Detecting Water in Landsat Data

One of the advantages of calculating your own spectral indices is that you are not limited to pre-generated products — you can find an algorithm that works for the specific problem you're trying to solve. Poking around a bit for algorithms that use Landsat's shortwave infrared bands for water detection, I found the Automated Water Extraction Index (AWEI), described in this paper: *An Automated Method for Extracting Rivers and Lakes from Landsat Imagery*.

The formula is:

$$AWEIsh = BLUE + 2.5 \times GREEN - 1.5 \times (NIR + SWIR1) - 0.25 \times SWIR2$$

(The lowercase "sh" after AWEI denotes that this is the variant developed for use in mountainous areas, where shadows can be misidentified as water. NIR = near infrared, SWIR1 = shortwave infrared one, and SWIR2 = shortwave infrared two.)

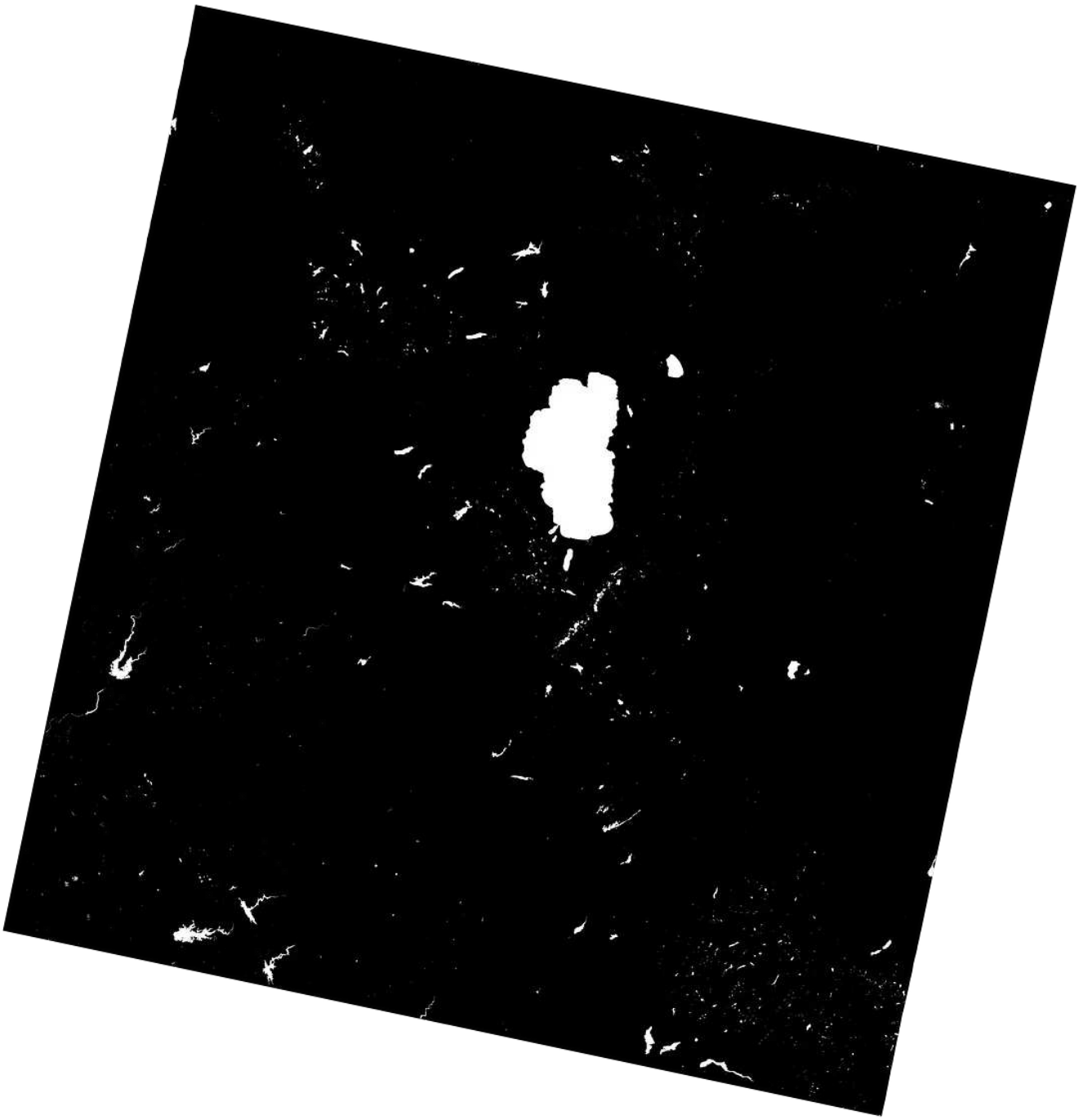
This algorithm was initially developed using data from the Thematic Mapper (TM) aboard Landsat 7, and fortunately the bands on Landsat 8 & 9's Operational Land Imager (OLI) are very close, so it works with both instruments. SWIR1 is OLI band 6, sensitive to wavelengths from 1.57–1.65 µm, and SWIR2 is OLI band 7, sensitive to wavelengths from 2.11–2.29 µm.

Here is the equivalent `gdal_calc.py` command:

```
gdal_calc.py -B tahoe_LC08_20210922_B2_SR_float.tif -C tahoe_LC08_20210922_B3_S
```

Notice that I didn't include the scale and offset values necessary to convert from 16-bit unsigned integers to 32-bit floating point data. I pre-calculated those values and saved the files with simpler names to make the code easier to read. As before, I linked the data file for each band to a capital letter and defined the calculation with `--calc`.

The output file is 32-bit floating point data, but the only important thing is that positive values are likely water, and negative values are likely land. Scaling so zero and above is white and less than zero is black turns out like this:



Water mask from the Automated Water Extraction Index optimized for use in areas likely to have shadows. Spectral indices that include shortwave infrared data are generally more accurate for detecting water than indices that only include visible and near infrared bands. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

Which looks pretty good. Great even, with the caveat that there's a smoke plume from the Caldor Fire just south of Tahoe that's being incorrectly identified as water. It's good to be aware that the real world can be a messy place, even if you've chosen your data carefully and done all your calculations correctly. The Landsat surface

reflectance product does include some data quality information which could help clean that up, but applying it properly is complicated enough to be worth its own tutorial.

So far I've calculated NDVI and AWEI (to find where there is (or is not) water), and I want to flag all the water pixels to be No Data (which will be transparent when the data is converted into an image or merged with another dataset). To do this, I have to combine the NDVI file and the AWEIsh files.

NDVI

0.7	0.5	0.2	0.1	0.1
0.2	0.5	0.4	0.2	0.1
0.8	0.2	-0.7	0.2	0.1
0.6	0.6	0.5	0.1	0.3
0.8	0.8	0.1	0.7	0.5

AWEIsh

-0.3	-0.4	-0.6	-0.3	-0.4
-0.2	-0.4	-0.6	0.1	-0.3
-0.1	-0.3	0.2	-0.4	-0.4
0.1	-0.4	-0.5	-0.2	-0.5
-0.3	-0.4	-0.2	-0.2	-0.3

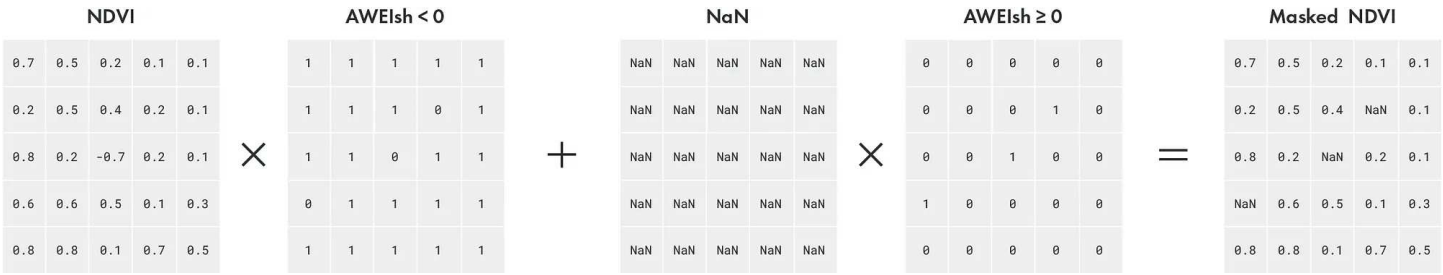
Idealized versions of the NDVI & AWEIsh arrays. Positive values in AWEIsh indicate water pixels.

The trick is to zero out all the NDVI pixels that are water, not land, then subtract a large value from those pixels and set it to No Data. You can't simply set zero as No Data because there are valid, non-water NDVI pixels with a value of zero or lower. Here's the command I used:

```
gdal_calc.py -V tahoe_LC08_20210922_SR_NDVI.tif -W tahoe_LC08_20210922_SR_AWEIs
```

I've assigned the NDVI data to `V` and water data to `W`, then used the `<` (less than) operator on AWEIsh to create an array with `1` where there is land (where AWEIsh is less than zero) and `0` where there is water (AWEIsh is greater than or equal to zero). Multiplying that array with NDVI sets all water pixels to zero and leaves NDVI as-is. Then I use the `>=` (greater than or equal to) operator on AWEIsh to create an array where `1` indicates water and `0` indicates land, and multiply that by `-9999`. Finally I add the water mask array to the NDVI array, which sets all water pixels to `-9999`, and then designate `-9999` as `No Data`, with the option `--NoDataValue=-9999`. This results in a nice clean dataset with land pixels having the calculated NDVI values

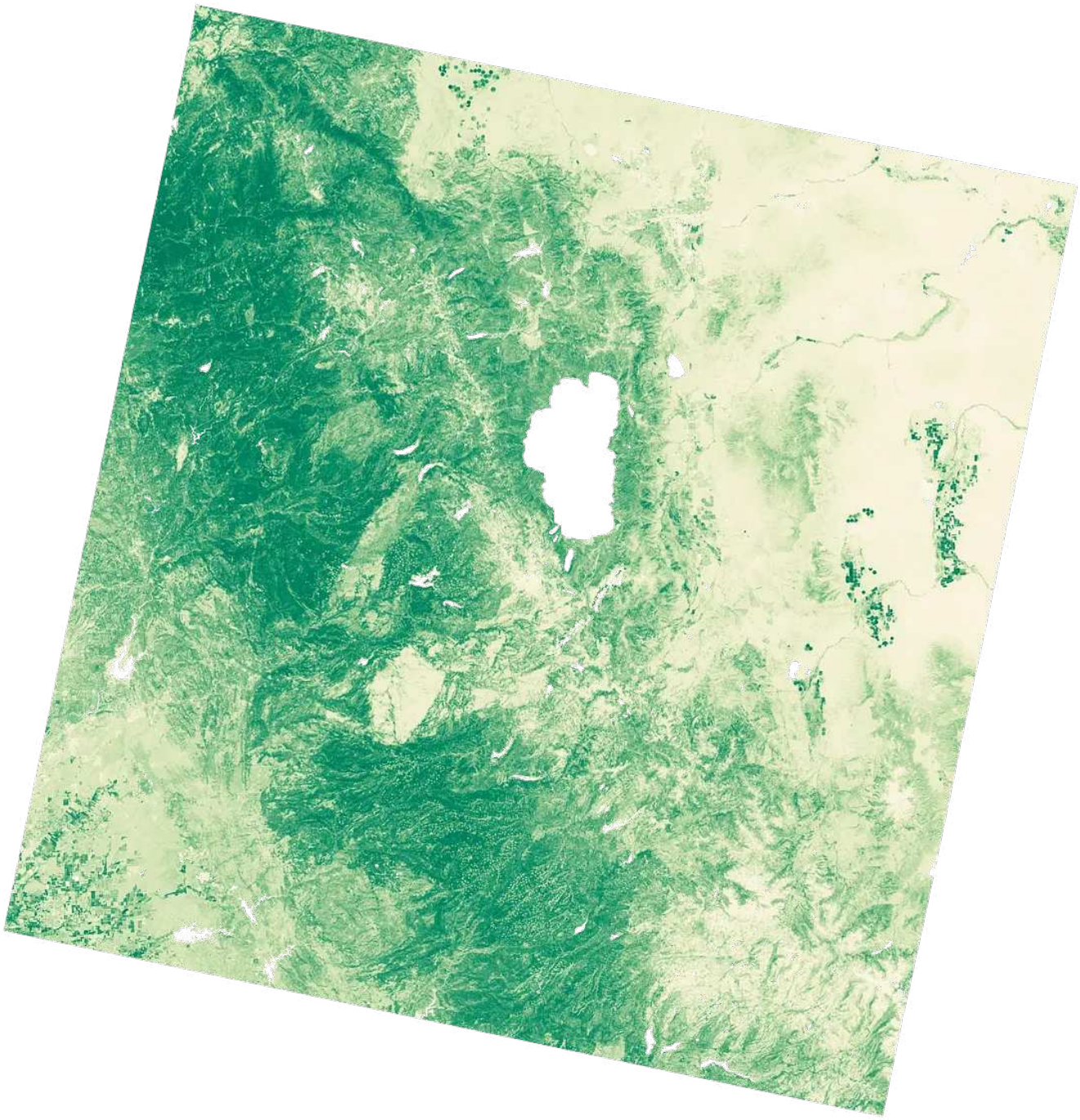
and water pixels assigned `No Data`.



Graphical version of the water mask calculation. Numpy's less than (<) and greater than (>) operators allow creation of a mask based on a threshold value. This allows invalid data to be removed from a map, or different types of data to be combined seamlessly.

I again used `gdaldem` to create a color image:

```
gdaldem color-relief tahoe_LC08_20210922_SR_NDVI_masked.tif ndvi_landsat_palett
```



A map of masked NDVI data in the Lake Tahoe region. In most cases vegetation indices aren't meaningful for water pixels, and it's best to remove them. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

In the resulting image all the land pixels are color-coded NDVI and all the water pixels are transparent. Which not only removes the distraction of all the oddly-colored water bodies, but allows for another dataset — perhaps one focused on water quality — to be featured. As with vegetation and binary water detection, there

are a number of ways to estimate water quality with Landsat data. I'm not an expert, but the Water Quality Index (WQI) defined in A reflectance-based water quality index and its application to examine degradation of river water quality in a rapidly urbanising megacity looks reasonable.

This is the Water Quality Index equation:

(GREEN + (BLUE − RED) − SWIR1)/(GREEN + (BLUE − RED) + SWIR1)

And the corresponding gdal_calc command:

```
gdal_calc.py -B tahoe_LC08_20210922_B2_SR_float.tif -C tahoe_LC08_20210922_B3_S
```

As with my calculation of the Automated Water Extraction Index, I used pre-scaled surface reflectance data converted to floating point data, renamed to be more readable. I also converted the land pixels to `No Data` with the AWEI mask at the same time I calculated WQI, rather than doing it in a separate step. Since the water pixels were valid in this case, I set all the locations where AWEI was **less** than zero to a no data value of `-9999`.

One last execution of `gdaldem color-relief`, this time using a classic Color Brewer palette:

```
gdaldem color-relief tahoe_LC08_20210922_SR_WQI_masked.tif color_brewer_Bl9.txt
```

The Color Brewer 9-class Blues palette formatted for gdaldem color-relief:

```
-9999,0,0,0,0
-9998,247,251,255,255
0,247,251,255,255
0.125,222,235,247,255
```

```
0.25,198,219,239,255
0.375,158,202,225,255
0.5,107,174,214,255
0.625,66,146,198,255
0.75,33,113,181,255
0.875,8,81,156,255
1,8,48,107,255
```

Which creates this map of blue lakes floating in empty space ...

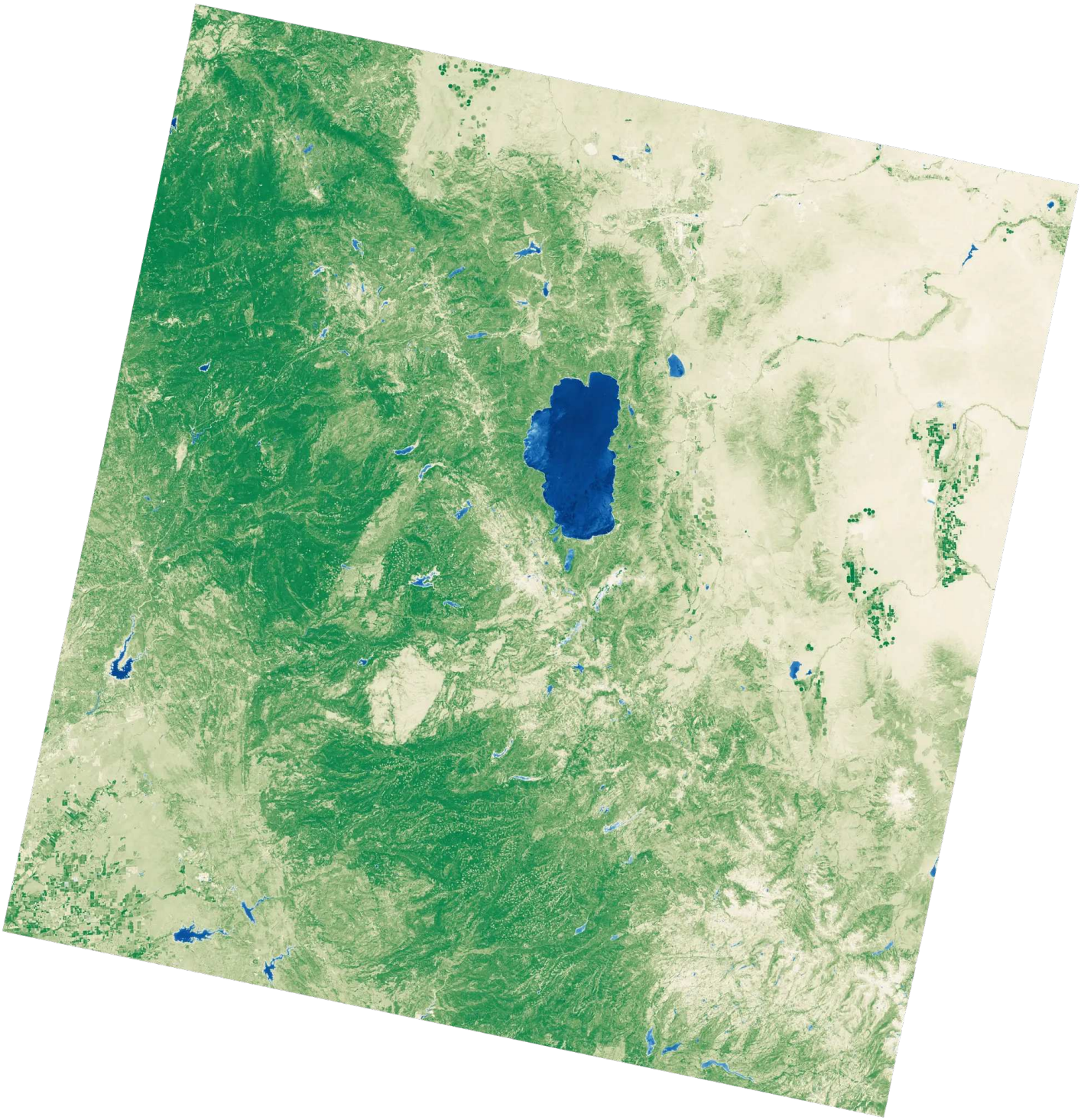
Map of Water Quality Index (WQI) for the Lake Tahoe region with land areas removed. Dark blue indicates high water quality, while lighter blue represents lower quality water. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

Nice, but not quite the result I want, which is to combine the NDVI data with the water quality data. Since areas of no data are already masked out in both the NDVI and WQI images, the data can be merged with a single line using gdalwarp:

```
gdalwarp tahoe_LC08_20210922_SR_WQI_color_masked.tif tahoe_LC08_20210922_SR_NDV
```

`gdalwarp` calls the command, `tahoe_LC08_20210922_SR_WQI_color_masked.tif` and `tahoe_LC08_20210922_SR_NDVI_masked.tif` are the input file names, `tahoe_LC08_20210922_WQI_NDVI.tif` is the output file name, and `-co COMPRESS=DEFLATE -co PREDICTOR=2` tells GDAL to losslessly compress the TIFFs to save drive space (single dash this time).

The result is a combined map of vegetation and water quality in Lake Tahoe and the surrounding Sierra Nevada, based on a single Landsat scene.



Combined NDVI and WQI map, generated entirely with GDAL. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

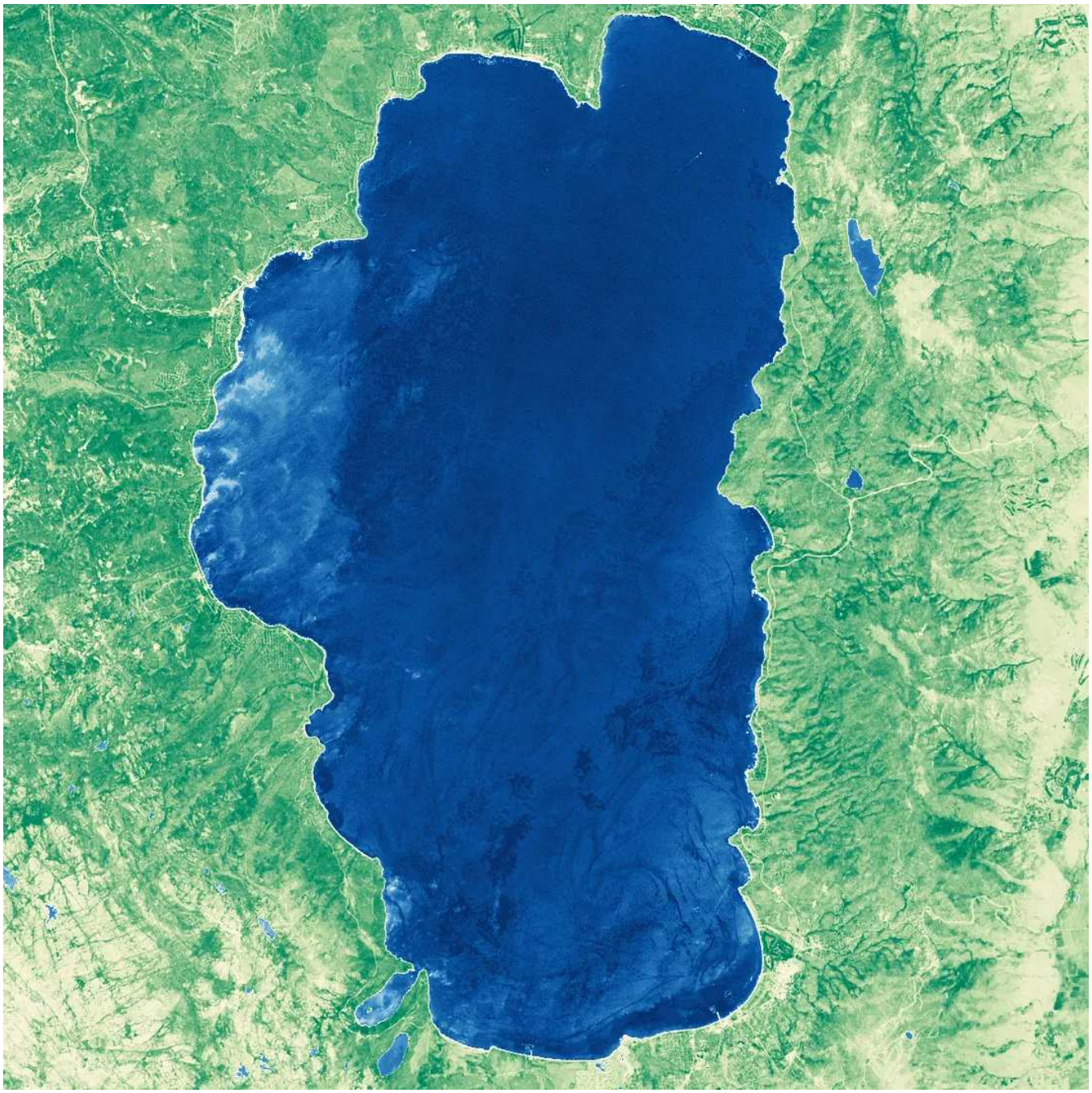
To recap: I used `gdal_calc.py` to calculate spectral indices and masks from multispectral data on the command line, colorized the data with `gdaldem color-relief`, and combined the files with `gdalwarp`.

I find `gdal_calc.py` convenient because it enables access to the powerful mathematical functions of numpy without the overhead of Python or other programming languages. As such it's great for prototyping visualizations or working with limited numbers of files. When you're working with large numbers of files or performing more complex tasks (like generating a similar map from a time series of Landsat scenes) it's probably better to look into full blown programming languages (which will likely use GDAL under the hood for parsing scientific data formats and reading georeferencing information). Which will be the topic of Part 8 — using the Python GDAL bindings to open and visualize HDF (and maybe NetCDF) files.

A Few More Thoughts on Spectral Indices

It's really easy to browse through a website, crack open a textbook, or read a blog post (including this one!) and find a spectral index that purports to quantify one physical parameter or another — there are literally hundreds of them. And often times you can calculate an index, throw a palette on it, and think "that looks about right". It's much harder to know which indices are appropriate to solve which problems, and understand the limitations inherent in different remote sensing systems and how that might affect your results. Remote sensing is hard!

For example, look at carefully at the water quality index data for Lake Tahoe. There are features that look like ripples and eddies that I'm convinced are due to surface glint and don't represent differences in water quality.



Water Quality Index map of Lake Tahoe. The features in the lake that look like ripples or eddies are much more likely to be due to sunglint than real differences in water quality. (Landsat 8 data collected on September 22, 2021. Available on the USGS Earth Explorer.)

There are a ton of variables and uncertainties that can impact the accuracy of even the simplest and most well understood indices, like NDVI. Seemingly small variations in the angle of the sun, the amount of ozone in the atmosphere, or sensor characteristics can have a large impact on the calculated measurement. A little effort put into understanding these limitations goes a long way towards helping to

realize the potential of satellite data to mitigate many social and environmental challenges.

Next up in A Gentle Introduction to GDAL Part 8: using GDAL to open sometimes-tricky scientific data formats like HDF & NetCDF, along with using GDAL to enable Python to read & write geospatial data.

1. A Gentle Introduction to GDAL
2. Map Projections & gdalwarp
3. Geodesy & Local Map Projections
4. Working with Satellite Data
5. Shaded Relief
6. Visualizing Data
7. Transforming Data (you are here)

Thanks to Joe Kington for being a sounding board and (as always) Frank Warmerdam for encouraging me to learn GDAL in the first place.

Cartography | Data Visualization | Remote Sensing | Maps | Gdal