Workshop: Raster and vector processing with GDAL

Table of Contents

1 Introduction

-
- Links to documentation in this tutorial used the versionned documentation for GDAL 1.11 (http://gdal.org/1.11/). For the documentation of the latest version (generally in development), remove the « 1.11/ »
- Typographic conventions :
  - in the following line, the dollar sign symbolizes the console invite. This must not be typed.

```
$ somecommand options
```

  - command lines that cannot fit into a single line are broken up with \ character. They can be pasted directly in a Linux shell (from the .odt or .html versions of this document. Not from the .pdf for Python scripts due to indentation issues). In a Windows shell, the \ character must be removed and all the content put into a single line
- Prerequisites :
  - OSGeo Live 8.5 or GDAL 1.11.1 or later, with Python bindings and QGIS (for display). The workshop can be run directly from the OSGeo Live DVD/USB stick or in a VM (at least 3 GB of RAM dedicated for it, or 1 GB of RAM + 2 GB of disk for storing data) , but it is recommended to install the OSGeo Live in a VM hard disk for better performance.
  - Commands to type are generally multi-platform, but some of them might be Linux/Unix specific.
  - Workshop test data. Available at http://download.osgeo.org/gdal/workshop/foss4ge2015/workshop_data.zip
  - A console opened in the directory with the workshop test data
  - A text editor
- Workshop data used and associated rights:

- paris.tif : extract of OpenStreetMap. (C) OpenStreetMap contributors : http://www.openstreetmap.org/copyright
- world.tif : from OSGeo-Live sample data
- ne_10m_admin_0_countries.* and ne_10m_admin_1_states_provinces_shp.*: from OSGeo -Live sample data (and originally from http://www.naturalearthdata.com, public domain)
- wellington_west and wellington_east.png : derived from https://data.linz.govt.nz/layer/1870-wellington-03m-rural-aerial-photos-2012-2013 , Licensed by Wellington Regional Council for reuse un-der the Creative Commons Attribution 3.0 New Zealand licence (https://data.linz.govt.nz/license/attribution-3-0-new-zealand/). For the purpose of this workshop, they have been post-processed to decrease their resolution, reduce to 256 colors, add collars and convert to PNG.
- MK_30m.tif and ML_30m.tif : derived from https://data.linz.govt.nz/layer/1768-nz-8m-digital-elevation-model-2012/ . License Creative Commons Attribution 3.0 New Zealand (https://data.linz.govt.nz/license/attribution-3-0-new-zealand/). For the purpose of this workshop, they have been post-processed to decrease their resolution.
- geomatrix.tif: from GDAL autotest suite. X/MIT License
- m2frac10bit.l1b : from GDAL extended data test suite, X/MIT License (http://download.osgeo.org/gdal/data/l1b/m2frac10bit.l1b)

2 Raster operations

2.1 Getting metadata about a raster / gdalinfo

2.1.1 Introduction

gdalinfo is the utility you will use all the time to discover metadata about a raster. This will also enable us to get a practical knowledge of most of the concepts of the GDAL data model : http://www.gdal.org/gdal_datamodel.html

Documentation of the gdalinfo utility : http://gdal.org/1.11/gdalinfo.html

2.1.2 Common and basic information

Let's start by an example:

```
$ gdalinfo world.tif
```

Output:

```
Driver: GTiff/GeoTIFF
Files: world.tif
Size is 2048, 1024
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"]]
Origin = (-180.000000000000000,90.000000000000000)
Pixel Size = (0.175781250000000,-0.175781250000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (-180.0000000,  90.0000000) (180d 0' 0.00"W, 90d 0' 0.00"N)
Lower Left  (-180.0000000, -90.0000000) (180d 0' 0.00"W, 90d 0' 0.00"S)
Upper Right ( 180.0000000,  90.0000000) (180d 0' 0.00"E, 90d 0' 0.00"N)
Lower Right ( 180.0000000, -90.0000000) (180d 0' 0.00"E, 90d 0' 0.00"S)
Center      (   0.0000000,   0.0000000) (  0d 0' 0.01"E,  0d 0' 0.01"N)
Band 1 Block=256x256 Type=Byte, ColorInterp=Red
  Overviews: 1024x512, 512x256, 256x128, 128x64, 64x32, 32x16, 16x8
Band 2 Block=256x256 Type=Byte, ColorInterp=Green
  Overviews: 1024x512, 512x256, 256x128, 128x64, 64x32, 32x16, 16x8
Band 3 Block=256x256 Type=Byte, ColorInterp=Blue
  Overviews: 1024x512, 512x256, 256x128, 128x64, 64x32, 32x16, 16x8
```

Analysis of output :

- Driver : Formats in GDAL are managed by different « drivers ». Basically 1 driver is dedicated to 1 format. Lists of drivers available at http://gdal.org/1.11/formats_list.html
- Files : list of files. Main file + potential additional files (world files, etc...)
- Size is 2048, 1024. First figure is Width in pixels. Second one is Height in pixels.
- Coordinate System: Also called projection, SRS (Spatial Reference System), CRS (Coordinate Refrence System), … The string presented here is in WKT (Well Known Text) format. The one used here is one of the most simple one. Coordinates are expressed in longitude & latitude on the WGS84 (World Geodetic Survey 1984) datum (due to longitude & latitude being directly used, this is called a geographic coordinate system « GEOGCS »)
- Origin : This is the projected coordinate of the upper-left corner of the image (the upper-left corner of the upper-left pixel). Here -180 is the longitude and 90 the latitude.
- Pixel Size : The dimension of a pixel in the units of the coordinate system. The first value is the width of the pixel, the second one its height. Here 0.17578125 is in degrees (see UNIT["degree"...] in the Coordinate System string). At the equator, this means roughly 0. 17578125 * 40000 / 360 = 19.5 km (the circonference of the Earth is rougly 40 000 km, and covers 360 degrees). You can notice the negative value for the pixel height. This is to indicate that the geospatial coordinates are decreasing when you go from the top of the image to the bottom of the image. This is the case for most geospatial rasters, so they appear correctly in all viewers.
- Metadata : a list of KEY=VALUE pairs, depending on the format and data. Here AREA_OR_POINT=AREA is a GDAL specific metadata to indicate that the on-file convention for the geo-registration is to take the upper-left corner of pixels (to be opposed to AREA_OR_POINT=POINT where the center of pixel is considered). You generally don't have to care about this one. This is mostly informational. For more details (rather involved), see https://trac.osgeo.org/gdal/wiki/rfc33_gtiff_pixelispoint

- Image structure metadata : gives details about :
    - the arrangement of pixels. INTERLEAVE=BAND here means that in the file you have first all the pixels for the Red band, then for the Green band and finally for the Blue band. The other formulation is INTERLEAVE=PIXEL which means that for each pixel you have the red value followed by the green and blue values, and then for the next pixel another R,G,B tuple, etc... This can be interesting to know for the efficiency of algorithms when processing big images. You might want to proceed closely with the natural organization of the data for best performance.
    - Potentially, compression used (JPEG, LZW, DEFLATE, etc...). Here's none.
    - Potentially, number of bits used when the data width is smaller than the data type holding it. For example 12-bit wide data (values between 0 and 4095) will be stored in a unsigned 16-bit integer, and NBITS=12 will be advertized
- Corner coordinates : the geospatial coordinates of the 4 corner of the images (including any padding), as well as the center pixel, expressed in the coordinate system for the first tuple. The second tuple gives their equivalents as longitude, latitude. Here since it is a geographic coordinate system, both values are identical
- Band description :
    - Block=256x256 : A block corresponds to a rectangular subpart of the raster. The first value is the width of the block and the second value its height. Typical block shapes are lines or group of lines (in which case the block width is the raster width) or tiles (typically squares), such as here. Knowing the block size is important when efficient reading of a raster is needed. In case of tiles, this means reading rasters from the left-most tile of the raster to the right-most of the upper lines and progressing that way downward to the bottom of the image.
    - Type=Byte : This is the data type of a pixel. Byte (unsigned byte) can store integer values between 0 and 255 and is the most common one. Other data types are possibles such as Int16 ([-32768,32767], UInt16([0,65535],Int32,UInt32,Float32 (single-precision floating point),Float64 (double-precision floating point). Int16 or Float32/Float64 can be encountered for digital elevation models (DEMs). UInt16 for raw satellite imagery. There are also data types that store complex numbers (with real and imaginary part), but this is rather esoteric and only used in a few drivers, mainly in the field of SAR (Synthetic Aperture Radar).
    - ColorInterp=Red : The color interpretation of the band. Common values are Red, Green, Blue, Alpha (for opacity channel. 0=fully transparent, 255=fully opaque) or Unknown. Other values are possible for other color spaces such as Cyan, Magenta, Yellow, blacK, but not often encountered. Note that there's no color interpretation fo Near InfraRed for example. This is something that must be deduced from other metada or knowledge of the product characteristics
    - Overviews : this gives the list of overviews available for the band (this may be empty). Overviews are also called pyramids in GIS. They are versions of reduced size of the full resolution raster to enable fast zoom-out operations. The first overview level is typically half the size (in both dimensions) as the full resolution one, the second overview level half the size of the first one, and so on... So the extra « cost » in term of storage size to add overviews is : 1 / (2*2) + 1 / (4*4) + 1 / (8*8) + etc.... which equals to 1 / 3. So overviews are generally and worth building, especially for use that involves interactive display of the raster. Here, looking at the file list and seing that only one file is mentionned, you can deduce that the overviews are stored within that file. Only a few formats, like TIFF/GeoTIFF, allow that. Otherwise files may be stored in external .ovr

files

2.1.3 Getting those information with Python

Open the Python shell and paste following statements :

```
from osgeo import gdal
ds = gdal.Open('world.tif')
print('File list:', ds.GetFileList())
print('Width:', ds.RasterXSize)
print('Height:', ds.RasterYSize)
print('Coordinate system:', ds.GetProjection())
gt = ds.GetGeoTransform() # captures origin and pixel size
print('Origin:', (gt[0], gt[3]))
print('Pixel size:', (gt[1], gt[5]))
print('Upper Left Corner:', gdal.ApplyGeoTransform(gt,0,0))
print('Upper Right Corner:', gdal.ApplyGeoTransform(gt,ds.RasterXSize,0))
print('Lower Left Corner:', gdal.ApplyGeoTransform(gt,0,ds.RasterYSize))
print('Lower Right Corner:',
gdal.ApplyGeoTransform(gt,ds.RasterXSize,ds.RasterYSize))
print('Center:', gdal.ApplyGeoTransform(gt,ds.RasterXSize/2,ds.RasterYSize/2))
print('Metadata:', ds.GetMetadata())
print('Image Structure Metadata:', ds.GetMetadata('IMAGE_STRUCTURE'))
print('Number of bands:', ds.RasterCount)
for i in range(1, ds.RasterCount+1):
      band = ds.GetRasterBand(i) # in GDAL, band are indexed starting at 1!
      interp = band.GetColorInterpretation()
      interp_name = gdal.GetColorInterpretationName(interp)
      (w,h)=band.GetBlockSize()
      print('Band %d, block size %dx%d, color interp %s' % (i,w,h,interp_name))
      ovr_count = band.GetOverviewCount()
      for j in range(ovr_count):
            ovr_band = band.GetOverview(j) # but overview bands starting at 0
            print('  Overview %d: %dx%d'%(j, ovr_band.XSize, ovr_band.YSize))
```

Compare the results with the output of gdalinfo.

A Python port of gdalinfo is also available at http://svn.osgeo.org/gdal/trunk/gdal/swig/python/samples/gdalinfo.py

2.1.4 General form of the geo-transformation matrix

Try:

```
$ gdalinfo geomatrix.tif
```

Output:

```
Driver: GTiff/GeoTIFF
Files: geomatrix.tif
Size is 20, 20
Coordinate System is:
PROJCS["WGS 84 / UTM zone 11N",
    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["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",-117],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",0],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AXIS["Easting",EAST],
    AXIS["Northing",NORTH],
    AUTHORITY["EPSG","32611"]]
GeoTransform =
  1841001.75, 1.5, -5
  1144003.25, -5, -1.5
Metadata:
  AREA_OR_POINT=Point
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  ( 1841001.750, 1144003.250) (104d50'47.45"W, 10d 7'13.55"N)
Lower Left  ( 1840901.750, 1143973.250) (104d50'50.69"W, 10d 7'12.72"N)
Upper Right ( 1841031.750, 1143903.250) (104d50'46.60"W, 10d 7'10.33"N)
Lower Right ( 1840931.750, 1143873.250) (104d50'49.85"W, 10d 7' 9.50"N)
Center      ( 1840966.750, 1143938.250) (104d50'48.65"W, 10d 7'11.53"N)
Band 1 Block=20x20 Type=Byte, ColorInterp=Gray
```

You can notice that contrary to the first example we no longer have a Origin and Pixel Size reported, but instead a GeoTransform.

The GeoTransform is the geo-transformation matrix, which is in mathematics described as an affine transformation from the coordinates in the pixel space (col,row) to the coordinates of the projected space (X,Y), with col and row starting from 0 for the upper-left pixel (in pixel space, not necessarily in projected space !). This is a series of 6 values gt[0], gt[1], … gt[5] such as :

```
                      X = gt[0] + col * gt[1] + row * gt[2]
                      Y = gt[3] + col * gt[4] + row * gt[5]
```

This is exactly the computation done by gdal.ApplyGeoTransform :
[X,Y]=gdal.ApplyGeoTransform(gt,row,col)

In the above example,

```
gt[0] = 1841001.75
gt[1] = 1.5
gt[2] = -5
gt[3] = 1144003.25
gt[4] = -5
gt[5] = -1.5
```

When the gt[2] and gt[4] terms are non zero, the image is no longer « north-up » with respect to the projected space, and you may have rotation and/or shearing.

To generate a more familiar image with those rotation/shearing being applied, you can use gdalwarp.

The related mathematics are detailed at http://en.wikipedia.org/wiki/Transformation_matrix

2.1.5 GCPs, subdatasets, geolocation arrays

Try:

```
$ gdalinfo m2frac10bit.l1b | less
```

Output:

```
Size is 2048, 222
Coordinate System is `'
GCP Projection =
GEOGCS["WGS 72",
    DATUM["WGS_1972",
        SPHEROID["WGS 72",6378135,298.26,
            AUTHORITY["EPSG","7043"]],
        TOWGS84[0,0,4.5,0,0,0.554,0.2263],
        AUTHORITY["EPSG","6322"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9108"]],
    AUTHORITY["EPSG","4322"]]
GCP[  0]: Id=, Info=
          (2023.5,221.5) -> (34.5374,59.0067,0)
GCP[  1]: Id=, Info=
          (1983.5,221.5) -> (32.1151,59.0086,0)
GCP[  2]: Id=, Info=
          (1943.5,221.5) -> (30.1062,58.9758,0)
GCP[  3]: Id=, Info=
          (1903.5,221.5) -> (28.3943,58.9232,0)
[…]
GCP[2800]: Id=, Info=
          (183.5,0.5) -> (-3.2412,55.8261,0)
GCP[2801]: Id=, Info=
          (143.5,0.5) -> (-4.3956,55.4315,0)
GCP[2802]: Id=, Info=
          (103.5,0.5) -> (-5.6875,54.9671,0)
GCP[2803]: Id=, Info=
          (63.5,0.5) -> (-7.158,54.408,0)
GCP[2804]: Id=, Info=
          (23.5,0.5) -> (-8.8701,53.714,0)
Metadata:
  DATA_TYPE=AVHRR FRAC
  DATASET_NAME=NSS.FRAC.M2.D08128.S1813.E1953.B0804243.SV
  LOCATION=Ascending
  PROCESSING_CENTER=NOAA/NESDIS - Suitland, Maryland, USA
  REVOLUTION=08042
  SATELLITE=METOP-A(2)
  SOURCE=Unknown receiving station
  START=year: 2008, day: 128, millisecond: 71248670
  STOP=year: 2008, day: 128, millisecond: 71285504
Subdatasets:
  SUBDATASET_1_NAME=L1B_ANGLES:"m2frac10bit.l1b"
  SUBDATASET_1_DESC=Solar zenith angles, satellite zenith angles and relative
azimuth angles
  SUBDATASET_2_NAME=L1B_CLOUDS:"m2frac10bit.l1b"
  SUBDATASET_2_DESC=Clouds from AVHRR (CLAVR)
Geolocation:
  LINE_OFFSET=0
  LINE_STEP=1
  PIXEL_OFFSET=0
  PIXEL_STEP=1
  SRS=GEOGCS["WGS
72",DATUM["WGS_1972",SPHEROID["WGS72",6378135,298.26,AUTHORITY["EPSG",7043]],TOW
GS84[0,0,4.5,0,0,0.554,0.2263],AUTHORITY["EPSG",6322]],PRIMEM["Greenwich",0,AUTH
ORITY["EPSG",8901]],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG",9108]],AUT
HORITY["EPSG",4322]]
  X_BAND=1
  X_DATASET=L1BGCPS_INTERPOL:"m2frac10bit.l1b"
  Y_BAND=2
  Y_DATASET=L1BGCPS_INTERPOL:"m2frac10bit.l1b"
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,  222.0)
Upper Right ( 2048.0,    0.0)
Lower Right ( 2048.0,  222.0)
Center      ( 1024.0,  111.0)
Band 1 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 1:  0.58  micrometers -- 0.68 micrometers
Band 2 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 2:  0.725 micrometers -- 1.10 micrometers
Band 3 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 3A: 1.58  micrometers -- 1.64 micrometers
Band 4 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 4:  10.3  micrometers -- 11.3 micrometers
Band 5 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 5:  11.5  micrometers -- 12.5 micrometers
```

GCPs

Lots of information here! The main novelty is the report of GCPs, which stand for Ground Control Points, also called Tiepoints or Control Points. Those are points of the raster for which the geolocation is known.

Let's look at the first one :

```
GCP[  0]: Id=, Info=
          (2023.5,221.5) -> (34.5374,59.0067,0)
```

Id, Info are text fields identifying the ground control points. They are rarely used in practice and often let to empty.

2023.5 and 221.5 are respectively the column and row in the raster for this GCP. As you can see, decimal values can be used. Here it means that the GCP is located at the center of the pixel (2023,221).

34.5374, 59.0067 and 0 are respectively the longitude/easting, latitude/northing and altitude of the GCP in the coordinate space. Here the reported GCP coordinate system is a geographic coordinate system, so they are longitude and latitude. The altitude is in most of the case 0.

You can also notice the lack of a geo-transform : the raster isn't projected yet. This can be done with gdalwarp

If the GCPs annoy you in the gdalinfo output, you can suppress them with the -nogcp option :

```
gdalinfo -nogcp  m2frac10bit.l1b
```

The http://svn.osgeo.org/gdal/trunk/gdal/swig/python/samples/gcps2ogr.py script can be used to plot the GCPs as vector points.

Download the script and run

```
$ python gcps2ogr.py m2frac10bit.l1b gcps.shp
```

This will generate gcps.shp that you can display gcps.shp with QGIS.

Subdatasets

This dataset also report subdatasets. Subdatasets are in fact full featured GDAL datasets

In :

```
SUBDATASET_1_NAME=L1B_ANGLES:"m2frac10bit.l1b"
SUBDATASET_1_DESC=Solar zenith angles, satellite zenith angles and relative
azimuth angles
```

The SUBDATASET_X_NAME is a string that can be passed to gdalinfo, and the following item describes its nature.

Subdatasets are often found in « scientific container » formats like netCDF, HDF4 or HDF5 that may contain multidimensionnal images (GDAL rasters are 2 dimensionnal), or images of different dimensions (all the bands of a same GDAL dataset must have the same width and height). In some cases, opening the main dataset gives a GDAL dataset without any raster band but with only subdatasets. In other cases, like the above case, the main dataset still contains raster bands.

To get the list of subdatasets in Python :

```
from osgeo import gdal
ds = gdal.Open('m2frac10bit.l1b')
subdatasets = ds.GetSubDatasets()
print(subdatasets)
sub_ds = gdal.Open(subdatasets[0][0]) # open first subdataset
print(sub_ds.GetDescription())
```

Geolocation array

You can see in the gdalinfo report a section about a geolocation array. This is an alternate way of providing geo-registration of a raster. In its basic form a geolocation array contains the longitude/easting and latitude/northing for each pixel of the main raster. A geolocation array is technically a GDAL datasets made of 2 bands (one for longitude/easting and latitude/northing), or 2 GDAL datasets of a single band. In some cases, there might be lesser sampling points in the geolocation arrays than in the main raster, hence the LINE_OFFSET, LINE_STEP, PIXEL_OFFSET and PIXEL_STEP items.

The gdalwarp utility can make use of the geolocation array.

2.1.6 Statistics, histogram, checksum

Try:

```
$ gdalinfo -stats -nogcp -nomd m2frac10bit.l1b
```

Output:

```
Driver: L1B/NOAA Polar Orbiter Level 1b Data Set
Files: m2frac10bit.l1b
Size is 2048, 222
Coordinate System is `'
Subdatasets:
 SUBDATASET_1_NAME=L1B_ANGLES:"m2frac10bit.l1b"
 SUBDATASET_1_DESC=Solar zenith angles, satellite zenith angles and relative
azimuth angles
 SUBDATASET_2_NAME=L1B_CLOUDS:"m2frac10bit.l1b"
 SUBDATASET_2_DESC=Clouds from AVHRR (CLAVR)
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,  222.0)
Upper Right ( 2048.0,    0.0)
Lower Right ( 2048.0,  222.0)
Center      ( 1024.0,  111.0)
Band 1 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 1:  0.58  micrometers -- 0.68 micrometers
  Min=39.000 Max=165.000
  Minimum=39.000, Maximum=165.000, Mean=43.898, StdDev=7.642
Band 2 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 2:  0.725 micrometers -- 1.10 micrometers
  Min=39.000 Max=226.000
  Minimum=39.000, Maximum=226.000, Mean=43.882, StdDev=8.992
Band 3 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 3A: 1.58  micrometers -- 1.64 micrometers
  Min=533.000 Max=983.000
  Minimum=533.000, Maximum=983.000, Mean=906.890, StdDev=27.716
Band 4 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 4:  10.3  micrometers -- 11.3 micrometers
  Min=454.000 Max=821.000
  Minimum=454.000, Maximum=821.000, Mean=563.780, StdDev=61.628
Band 5 Block=2048x1 Type=UInt16, ColorInterp=Undefined
  Description = AVHRR Channel 5:  11.5  micrometers -- 12.5 micrometers
  Min=446.000 Max=803.000
  Minimum=446.000, Maximum=803.000, Mean=550.119, StdDev=61.217
```

The -stats option asks gdalinfo to compute the minimum, maximum, mean and standard deviation for each band. The -nogcp and -nomd options are just to suppress GCP and metadata from the output.

Now just run:

```
$ gdalinfo -nogcp -nomd m2frac10bit.l1b
```

and compare the output. You may notice that a m2frac10bit.l1b.aux.xml file is now mentionned, but the statistics are still reported. This file has been generated when computing the statistics. Let's display it

```
$ cat  m2frac10bit.l1b.aux.xml
```

Output:

```
<PAMDataset>
 <Metadata>
 <MDI key="DATA_TYPE">AVHRR FRAC</MDI>
 <MDI key="DATASET_NAME">NSS.FRAC.M2.D08128.S1813.E1953.B0804243.SV</MDI>
 <MDI key="LOCATION">Ascending</MDI>
 <MDI key="PROCESSING_CENTER">NOAA/NESDIS - Suitland, Maryland, USA</MDI>
 <MDI key="REVOLUTION">08042</MDI>
 <MDI key="SATELLITE">METOP-A(2)</MDI>
 <MDI key="SOURCE">Unknown receiving station</MDI>
 <MDI key="START">year: 2008, day: 128, millisecond: 71248670</MDI>
 <MDI key="STOP">year: 2008, day: 128, millisecond: 71285504</MDI>
 </Metadata>
 <Metadata domain="GEOLOCATION">
 <MDI key="LINE_OFFSET">0</MDI>
 <MDI key="LINE_STEP">1</MDI>
 <MDI key="PIXEL_OFFSET">0</MDI>
 <MDI key="PIXEL_STEP">1</MDI>
 <MDI key="SRS">GEOGCS["WGS 72",DATUM["WGS_1972",SPHEROID["WGS 
72",6378135,298.26,AUTHORITY["EPSG",7043]],TOWGS84[0,0,4.5,0,0,0.554,0.2263],AUT
HORITY["EPSG",6322]],PRIMEM["Greenwich",0,AUTHORITY["EPSG",8901]],UNIT["degree",
0.0174532925199433,AUTHORITY["EPSG",9108]],AUTHORITY["EPSG",4322]]</MDI>
 <MDI key="X_BAND">1</MDI>
 <MDI key="X_DATASET">L1BGCPS_INTERPOL:"m2frac10bit.l1b"</MDI>
 <MDI key="Y_BAND">2</MDI>
 <MDI key="Y_DATASET">L1BGCPS_INTERPOL:"m2frac10bit.l1b"</MDI>
 </Metadata>
 <Metadata domain="SUBDATASETS">
 <MDI key="SUBDATASET_1_NAME">L1B_ANGLES:"m2frac10bit.l1b"</MDI>
 <MDI key="SUBDATASET_1_DESC">Solar zenith angles, satellite zenith angles 
and relative azimuth angles</MDI>
 <MDI key="SUBDATASET_2_NAME">L1B_CLOUDS:"m2frac10bit.l1b"</MDI>
 <MDI key="SUBDATASET_2_DESC">Clouds from AVHRR (CLAVR)</MDI>
 </Metadata>
 <PAMRasterBand band="1">
 <Description>AVHRR Channel 1:  0.58  micrometers — 0.68 
micrometers</Description>
 <Metadata>
 <MDI key="STATISTICS_MAXIMUM">165</MDI>
 <MDI key="STATISTICS_MEAN">43.898287936374</MDI>
 <MDI key="STATISTICS_MINIMUM">39</MDI>
 <MDI key="STATISTICS_STDDEV">7.6418424876281</MDI>
 </Metadata>
 </PAMRasterBand>
 <PAMRasterBand band="2">
 <Description>AVHRR Channel 2:  0.725 micrometers — 1.10 
micrometers</Description>
 <Metadata>
 <MDI key="STATISTICS_MAXIMUM">226</MDI>
 <MDI key="STATISTICS_MEAN">43.882075679195</MDI>
 <MDI key="STATISTICS_MINIMUM">39</MDI>
 <MDI key="STATISTICS_STDDEV">8.9918754785884</MDI>
 </Metadata>
 </PAMRasterBand>
 <PAMRasterBand band="3">
 <Description>AVHRR Channel 3A: 1.58  micrometers — 1.64
micrometers</Description>
 <Metadata>
 <MDI key="STATISTICS_MAXIMUM">983</MDI>
 <MDI key="STATISTICS_MEAN">906.890341269</MDI>
 <MDI key="STATISTICS_MINIMUM">533</MDI>
 <MDI key="STATISTICS_STDDEV">27.715692093586</MDI>
 </Metadata>
  </PAMRasterBand>
  <PAMRasterBand band="4">
    <Description>AVHRR Channel 4:  10.3  micrometers — 11.3
micrometers</Description>
    <Metadata>
      <MDI key="STATISTICS_MAXIMUM">821</MDI>
      <MDI key="STATISTICS_MEAN">563.78016566371</MDI>
      <MDI key="STATISTICS_MINIMUM">454</MDI>
      <MDI key="STATISTICS_STDDEV">61.627940356416</MDI>
    </Metadata>
  </PAMRasterBand>
  <PAMRasterBand band="5">
    <Description>AVHRR Channel 5:  11.5  micrometers — 12.5
micrometers</Description>
    <Metadata>
      <MDI key="STATISTICS_MAXIMUM">803</MDI>
      <MDI key="STATISTICS_MEAN">550.11860615498</MDI>
      <MDI key="STATISTICS_MINIMUM">446</MDI>
      <MDI key="STATISTICS_STDDEV">61.216671085773</MDI>
    </Metadata>
  </PAMRasterBand>
</PAMDataset>
```

You can see this file collects the statistics (as well as the other metadata). In case you are wondering what PAM in the above stands for, it is Persistant Auxiliary Metadata. The PAM .aux.xml format is something GDAL specific (and of course indirectly recognized by all software based on GDAL).

Caution : when a .aux.xml file exists, even if you specify -stats, gdalinfo will not recompute the statistics from the raster file but will use the ones stored in the .aux.xml. So in case the raster file would have been updated with new values, you may need to manually delete the .aux.xml file.

In Python :

```
from osgeo import gdal
ds = gdal.Open('m2frac10bit.l1b')
for i in range(1,ds.RasterCount+1):
      band = ds.GetRasterBand(i)
      (min, max, mean, stddev) = band.GetStatistics(False, True)
      print('Band%d, min=%.3f, max=%.3f,mean=%.3f,stddev=%.3f'% \
             (i, min, max, mean, stddev))
```

In GetStatistics(False,True), the first boolean value means whether to use approximate statistics or not (here False=exact statitstics), and the second value means whether to force or not the computation of statistics (here True means force compute of statistics if the .aux.xml file does not exist yet)

2.2 Raster conversion / gdal_translate

2.2.1 Introduction

This is the utility to use to do format conversions, subsetting, format optimization, adding georeferencing, ...

Documentation of the gdal_translate utility : http://gdal.org/1.11/gdal_translate.html

2.2.2 Format conversion

Try :

```
$ gdal_translate wellington_west.png wellington_west.gif
```

You get a warning here mentionning that the conversion has been done to TIFF and not to GIF as it was perhaps intended. GDAL is generally not sensitive to extensions, so you have to be explicit about the format, otherwise the default output format, which is GeoTiff will be used.

To get the list of supported formats, do :

```
$ gdal_translate --formats
```

The letters between parenthesits after the format short name indicate the capabilities :

- ro means read-only
- rw means read and one-time creation only
- rw+ means read, create and update
- The v signs means that it support virtual files, we will see that later.

So the GIF driver support read and one-time creation, which is enough for gdal_translate. So let's correct our last attempt to specify the output format with -of :

```
$ gdal_translate -of GIF wellington_west.png wellington_west.gif
```

2.2.3 Adding georeferencing

Let's look at wellington_west.png with gdalinfo. A long list of quadruplets is displayed. It is a color table. This image has a single band with Byte values, that point to 256 coded colors. Use -noct to disable the display of this color table.

```
$ gdalinfo -noct wellington_west.png
```

Output :

```
Driver: PNG/Portable Network Graphics
Files: wellington_west.png
       wellington_west.wld
Size is 1831, 1835
Coordinate System is `'
```

```
Origin = (1731543.836827248800546,5461586.738620690070093)
Pixel Size = (28.001501693600002,-28.001034482800002)
Corner Coordinates:
Upper Left  ( 1731543.837, 5461586.739)
Lower Left  ( 1731543.837, 5410204.840)
Upper Right ( 1782814.586, 5461586.739)
Lower Right ( 1782814.586, 5410204.840)
Center      ( 1757179.212, 5435895.789)
Band 1 Block=1831x1 Type=Byte, ColorInterp=Palette
  Color Table (RGB with 256 entries)
```

We can see the presence of georeferenced coordinates (Origin and Pixel Size), that comes from wellington_west.wld, a World File. This is a 6 line text file that gives the information needed to build the 6-value geo-transformation matrix. Note that the order of values and their semantics are not exactly the same (World File use center of pixel convention), but GDAL will make the needed conversions.

But here we lack the Coordinate System. We must use external knowedge here to fill the gap. This dataset being from New Zealand and the coordinates being obviously not longitudes/latitudes, we could assume that this dataset is in the New Zealand Transverse Mercator (NTZM) map projection (for example by searching New Zealand in http://epsg.io/?q=new+zealand), and this is indeed the case. This coordinate system is codified by the EPSG (European Petroleum Survey Group) as being « EPSG:2193 »

We can see its definition with :

```
$ gdalsrsinfo EPSG:2193
```

Output :

```
PROJ.4 : '+proj=tmerc +lat_0=0 +lon_0=173 +k=0.9996 +x_0=1600000 +y_0=10000000
+ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs '

OGC WKT :
PROJCS["NZGD2000 / New Zealand Transverse Mercator 2000",
    GEOGCS["NZGD2000",
        DATUM["New_Zealand_Geodetic_Datum_2000",
            SPHEROID["GRS 1980",6378137,298.257222101,
                AUTHORITY["EPSG","7019"]],
            TOWGS84[0,0,0,0,0,0,0],
            AUTHORITY["EPSG","6167"]],
        PRIMEM["Greenwich",0,
            AUTHORITY["EPSG","8901"]],
        UNIT["degree",0.0174532925199433,
            AUTHORITY["EPSG","9122"]],
        AUTHORITY["EPSG","4167"]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",173],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",1600000],
    PARAMETER["false_northing",10000000],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","2193"]]
```

Now let us create a GeoTIFF from that :

```
$ gdal_translate -a_srs EPSG:2193 wellington_west.png wellington_west.tif
```

Let's look at the result :

```
$ gdalinfo -noct wellington_west.tif
```

Output :

```
Driver: GTiff/GeoTIFF
Files: wellington_west.tif
Size is 1831, 1835
Coordinate System is:
PROJCS["NZGD2000 / New Zealand Transverse Mercator 2000",
    GEOGCS["NZGD2000",
        DATUM["New_Zealand_Geodetic_Datum_2000",
            SPHEROID["GRS 1980",6378137,298.2572221010002,
                AUTHORITY["EPSG","7019"]],
            TOWGS84[0,0,0,0,0,0,0],
            AUTHORITY["EPSG","6167"]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433],
        AUTHORITY["EPSG","4167"]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",173],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",1600000],
    PARAMETER["false_northing",10000000],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","2193"]]
Origin = (1731543.836827248800546,5461586.738620690070093)
Pixel Size = (28.001501693600002,-28.001034482800002)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  ( 1731543.837, 5461586.739) (174d33'49.52"E, 40d59'10.66"S)
Lower Left  ( 1731543.837, 5410204.840) (174d34'29.34"E, 41d26'56.27"S)
Upper Right ( 1782814.586, 5461586.739) (175d10'22.40"E, 40d58'35.11"S)
Lower Right ( 1782814.586, 5410204.840) (175d11'17.70"E, 41d26'20.14"S)
Center      ( 1757179.212, 5435895.789) (174d52'29.74"E, 41d12'47.03"S)
Band 1 Block=1831x4 Type=Byte, ColorInterp=Palette
  Color Table (RGB with 256 entries)
```

And let's check that the coordinate system we select is consistant by using the OGR geocoding API:

```
$ ogrinfo :memory: -q -sql "SELECT ogr_geocode('Wellington, New Zealand')"
```

Output :

```
Layer name: SELECT
OGRFeature(SELECT):0
```

```
  POINT (174.7772239 -41.2887639)
```

Yes, that seems to be consistant !

2.2.4 Subsetting (extracting a sub-window)

Let's suppose we are only interested in a 4 km x 4 km area centered around Wellington city center. We will convert first the above longitude, latitude in NZGD2000 with :

```
$ gdaltransform -s_srs WGS84 -t_srs EPSG:2193
174.7772239 -41.2887639
```

The command expects tuple of longitude/easting latitude/northing to be entered on the console and validated by Enter. The application may be terminated with Ctrl+Z.

Output :

```
1748816.1501341 5427663.38112408 0
```

So if we substract/add 2000 from this coordinate, we can do :

```
$ gdal_translate -projwin 1746816 5429663 1750816 5425663 \
                 wellington_west.tif wellington_city.tif
```

Conventions are « -projwin ulx uly lrx lry » where :

- ulx : X of upper-left corner
- uly : Y of upper-left corner
- lrx : X of lower-right corner
- lry : Y of lower-right corner

You can check with QGIS that this indeed the city center area.

2.2.5 Data type conversion

Let's look at a digital elevation model file :

```
$ gdalinfo -mm MK_30m.tif
```

Output :

```
Driver: GTiff/GeoTIFF
Files: MK_30m.tif
Size is 2185, 2185
Coordinate System is:
PROJCS["NZGD2000 / New Zealand Transverse Mercator 2000",
    GEOGCS["NZGD2000",
        DATUM["New_Zealand_Geodetic_Datum_2000",
            SPHEROID["GRS 1980",6378137,298.2572221010002,
                AUTHORITY["EPSG","7019"]],
            AUTHORITY["EPSG","6167"]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433],
        AUTHORITY["EPSG","4167"]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",173],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",1600000],
    PARAMETER["false_northing",10000000],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","2193"]]
Origin = (1703936.000000000000000,5439488.000000000000000)
Pixel Size = (30.000000000000000,-30.000000000000000)
Metadata:
  AREA_OR_POINT=Point
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  ( 1703936.000, 5439488.000) (174d14'21.77"E, 41d11'21.48"S)
Lower Left  ( 1703936.000, 5373938.000) (174d15' 2.45"E, 41d46'46.57"S)
Upper Right ( 1769486.000, 5439488.000) (175d 1'14.35"E, 41d10'41.67"S)
Lower Right ( 1769486.000, 5373938.000) (175d 2'20.64"E, 41d46' 5.92"S)
Center      ( 1736711.000, 5406713.000) (174d38'14.80"E, 41d28'46.35"S)
Band 1 Block=2185x1 Type=Float32, ColorInterp=Gray
    Computed Min/Max=0.000,928.151
  NoData Value=-32767
```

The -mm option computes only the min/max values. We can also see a NoData value being advertized. The NoData value is a special value which means that pixel that have that value have no valid information. The NoData value is ignored in min/max, statistics or histogram computations. We can however easily find if there are actually pixels in that tile that are set to the NoData value with the following Python script :

```
from osgeo import gdal
ds = gdal.Open('MK_30m.tif')
values = ds.ReadAsArray()
print(-32768 in values)
```

The answer is False. All values in this DEM tile are valid.

We might want to generate a more convenient visual dataset from the DEM with :

```
$ gdal_translate -outsize 50% 50% -ot Byte -of PNG MK_30m.tif MK_vis.png
```

- -outsize 50% 50% is just to get a half-size reduction. Note that gdal_translate uses nearest neighbour sampling, which is not always appropriate. gdalwarp can be used for other resampling methods.
- -ot Byte : asks for data type conversion

Display MK_vis.png. You can notice large zones saturated to white. Why so ? Because -ot does only data type conversion and clip values that would become out-of-range. So any elevation above 255 got clipped to 255. We also want to rescale the range [0,928.151] to [0,255]. We can do this by adding -scale

```
$ gdal_translate -outsize 50% 50% -ot Byte -scale -of PNG MK_30m.tif MK_vis.png
```

You can control more precisely the rescaling with :

```
-scale src_min src_max
```

or

```
-scale src_min src_max dst_min dst_max
```

When not specifying dst_min and dst_max, 0 and 255 are selected. When not specifying src_min and src_max, the minimum/maximum values of the dataset are used.

You can also experiment with non-linear scaling with the -exponent value. For example try with -exponent 0.9. This will give an image slightly brighter. For reference, the destination value (dst_val) is computed from the (src_val) according to the following formula :

dst_val = dst_min + (dst_max – dst_min) * ((src_val-src_min)/(src_max – src_min)) ^ exponent

2.3 Georeferencing and reprojection / gdalwarp

2.3.1 Introduction

This is the utility to use to do georefercing from ground control points, reprojection, resampling, mosaicing, ...

Full documentation of the gdalwarp utility : http://gdal.org/1.11/gdalwarp.html

2.3.2 Manual georeferencing from ground control points

The raster wellington_east.png comes without its sidecar .wld file, so we have to do it manually. Fortunately the right part of wellington_west.png and wellington_east.png slightly overlaps. We can then try to identify particular points for which we will have the georeferenced coordinates from wellington_west.png and the pixel coordinate in wellington_east.png. Open 2 QGIS instances with each of the image in one instance. Note that for wellington_east.png QGIS will display negative values for the line, but we must ignore the negative sign here (QGIS has used a pseudo geomatrix with a negative gt[5] coefficient so that it displays « north-up »). We need at the very least 3 GCPs to be later be able to rectify the image. The more the better.

We have already pre-identified 3 ground control points for you :

- (col,row)=(251.5,520.5) : (x,y)=(1778689,5430015)
- (col,row)=(157.5,1001.5) : (x,y)=(1776053,5416548)
- (col,row)=(274.5,1036.5) : (x,y)=(1779333,5415561)

We can add them with gdal_translate :

```
$ gdal_translate -a_srs EPSG:2193 -gcp 251.5 520.5 1778689 5430015 \
                                  -gcp 157.5 1001.5 1776053 5416548 \
                                  -gcp 274.5 1036.5 1779333 5415561 \
               wellington_east.png wellington_east_with_3gcp.tif
```

Check with gdalinfo that they are correctly set.

Datasets with GCPs are not easy to work with, so we want to build a dataset with only a geotransformation matrix.

Let's do the warp :

```
$ gdalwarp wellington_east_with_3gcp.tif wellington_east_warped.tif
```

Now open wellington_west.tif and wellington_east_warped.tif in the same QGIS project : they perfectly overlap.

Important note : gdalwarp will not erase the target dataset if it exists and will reuse its existing coordinate system, geospatial extent and existing pixel values. So unless you do mosaicing or compositing, you need to delete the potentially already existing target dataset, or append -overwrite as an option

There are several transformation methods possible in GDAL :

- Thin Plate Spline (TPS) : the ground control points after transformation are guaranteed to be at their specified gelocation, which is great when they are exact, not so great otherwise. Points between them are interpolated in a smooth way. Computation time can be long when they are thousands or more GCPs
- Polynomial order 1 / Affine transform : A least-squared regression is done to compute a polynom of order 1 from the GCPs. So they will generally not be at their specified geolocation after reprojection
- Polynomial order 2 : Same with a polynom of order 2. Needs at least 6 GCPs to be computed
- Polynomial order 3 : Not recommended because of known numerical instability in current implementation.

By default, polynomial order 1 will be selected if there are less than 6 points, otherwise polynomial order 2. TPS can be selected with the -tps option of gdalwarp and polynomial with -order n where n=1 or 2

Since no important warping has been done during the process (the source image being already orthorectified : only the georegistration was missing), the target image has almost the same dimensions has the source image : 2339x2050 for wellington_east_warped.tif versus 2340x2048 for wellington_east_with_3gcp.tif. We can check the pixel locations corresponding to the geospatial locations of our 3 GCPs :

```
$ gdallocationinfo -geoloc wellington_east_warped.tif 1778689 5430015
```

Output :

```
Report:
  Location: (251P,520L)
  Band 1:
    Value: 18
```

```
$ gdallocationinfo -geoloc wellington_east_warped.tif 1776053 5416548
```

Output :

```
Report:
  Location: (157P,1000L)
  Band 1:
    Value: 9
```

```
$ gdallocationinfo -geoloc wellington_east_warped.tif 1779333 5415561
```

Output :

```
Report:
  Location: (274P,1035L)
  Band 1:
    Value: 42
```

So this almost matches our input GCP.

2.3.3 Reprojection

Basic reprojection

A simple example to begin with, with the WebMercator / GoogleMaps projection used in allmost all slippy maps web applications.

```
$ gdalwarp world.tif world_webmercator.tif -t_srs EPSG:3857
```

Look at the result : Greenland seems to be as big as Africa, whereas it is only 1/15th of its area in reality. Yes, this is the expected result for this projection !

Dealing with discontinuities

Now let's try to have a point of view centered around the north pole. A projection suited for north pole is the Universal Polar Stereographic projection, whose EPSG code is 32661. Should be easy :

```
$ gdalwarp world.tif north_pole.tif -t_srs EPSG:32661
```

Output :

```
Creating output file that is 1619P x 1619L.
Processing input file world.tif.
ERROR 1: Too many points (441 out of 441) failed to transform,
unable to compute output bounds.
Warning 1: Unable to compute source region for output window 0,0,1619,1619,
skipping.
0...10...20...30...40...50...60...70...80...90...100 – done.
```

Hum, lots of errors. Checking the result shows a fully black image... But after all, we started from a whole world raster, including the southern hemisphere which cannot be represented in the target projection.

Let's create a subset with just the northern hemisphere :

```
$ gdal_translate world.tif world_north_hemisphere.tif -projwin -180 90 180 0
```

And then :

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
           -overwrite
```

No errors! Let's check visually. Ouch : only a thin circle is non black. We must admit that polar projections are hard, and thus we must give some hint to the warper algorithm. Instead of walking only on the edges of the target extent to determine the zone of the input dataset that will fill it, we will instruct it to sample more points inside this target extent with the warping option SAMPLE_GRID=YES (note: there will be no warning in case of typo)

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif \
                            -t_srs EPSG:32661 -wo SAMPLE_GRID=YES -overwrite
```

Check the visual result. Almost perfect, but there is a strange black disk section crossing far-east Russia and Pacific. This actually corresponds to the anti-meridian (meridian of longitude +/- 180), which is another annoying discontinuity that often cause troubles.

To understand what happens, let's re-run the above command by adding debug traces:

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif \
                     -t_srs EPSG:32661 -wo SAMPLE_GRID=YES -overwrite --debug on
```

The follwing traces are emitted :

```
GDAL: GDALOpen(north_pole.tif, this=0x24791e0) succeeds as GTiff.

GDAL: GDALOpen(world_north_hemisphere.tif, this=0x247baf0) succeeds as GTiff.
Processing input file world_north_hemisphere.tif.
WARP: Copying metadata from first source to destination dataset
OGRCT: PROJ >= 4.8.0 features enabled
OGRCT: Wrap source at 0.
OGRCT: Source: +proj=longlat +datum=WGS84 +no_defs
OGRCT: Target: +proj=stere +lat_0=90 +lat_ts=90 +lon_0=0 +k=0.994 +x_0=2000000
+y_0=2000000 +datum=WGS84 +units=m +no_defs
OGRCT: Wrap target at 0.
OGRCT: Source: +proj=stere +lat_0=90 +lat_ts=90 +lon_0=0 +k=0.994 +x_0=2000000
+y_0=2000000 +datum=WGS84 +units=m +no_defs
```

```
OGRCT: Target: +proj=longlat +datum=WGS84 +no_defs
GDAL: GDALWarpKernel()::GWKNearestNoMasksByte()
Src=0,0,2016x512 Dst=0,0,4222x4222
0...10...20...30...40...50...60...70...80...90...100 - done.
GDAL: GDALClose(world_north_hemisphere.tif, this=0x247baf0)
GDAL: GDALClose(north_pole.tif, this=0x24791e0)
```

Let's look at the line « Src=0,0,2048x512 Dst=0,0,4222x4222 ». This means that to fill the rectangular area starting at pixel (0,0) and with a (width,height)=(4222,4222) in the target dataset, a rectangular area starting at pixel (0,0) and with a (width,height)=(2016,512) in the source dataset has been read. But the source dataset is 2048x512 big. So there is a band of 32 pixels at the right of the dataset that wasn't used. This is due to the sampling grid used that is a 21x21 points grid. We can to make it more precise with the SAMPLE_STEPS=N warping option.

Let's try with N=100

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
       -wo SAMPLE_GRID=YES -wo SAMPLE_STEPS=100 -overwrite --debug on
```

Still a circular portion mission, thinner than previous time, and a pin hole appearing at north pole ! This is due to the sampling grid missing to hit the north pole and thus the line of latitude 90° is missed... With SAMPLE_STEPS=101 the pinhole artifact disappears. Experimentally with SAMPLE_STEPS=500, the result would be the one expected, but this begins to be slow and hard to guess. A more reliable way in that situation would be to add a hint to consider extra pixels beyond the source area compute. This can be done with the SOURCE_EXTRA warping option. Here the source dataset width is 2048 and the computed source width was 2016, so we need 2048-2016=32 extra pixels.

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -overwrite --debug on
```

The result is now the one we expected.

Transparency

In the above results, we have black corners that are not desirable. We can ask gdalwarp to generate an alpha channel so they can be transparent with the -dstalpha option:

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -dstalpha -overwrite
```

What is we want to make the oceans transparent ? First, let's identify the RGB triplet for the blue. We could do that with the identify tool of QGIS, but as the North Pole is in the middle of the ocean, we can just query the values at latitude=90 (and longitude=0 since we must specify one), with :

```
$ gdallocationinfo  -geoloc world_north_hemisphere.tif 0 89.9
```

Output:

```
Report:
  Location: (1024P,0L)

  Band 1:
    Value: 12
  Band 2:
    Value: 11
  Band 3:
    Value: 68
```

Now, we are going to specify the 12 11 68 triplet as the source nodata combination for gdalwarp :

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -dstalpha \
            -srcnodata '12 11 68' -ts 1024 1024 -overwrite
```

Check the result. We have some edge effects near the cost due to non uniform blue.

We are going to try improving this result by generating a transparency mask that accepts RGB triplets to be not exactly (12,11,68). Let's run the following Python script :

```
from osgeo import gdal
import numpy
ds = gdal.Open('world_north_hemisphere.tif')
ar_data = ds.ReadAsArray()
ar_blue = numpy.ndarray(ar_data.shape)
ar_blue[0].fill(12)
ar_blue[1].fill(11)
ar_blue[2].fill(68)
ar_0 = numpy.zeros(ar_data[0].shape)
ar_255 = numpy.ones(ar_data[0].shape) * 255
# If the sum of the absolute difference between the actual value and the
# reference blue is less than 40, then select 0 (transparent), otherwise select
# 255 (opaque)
ar_res = numpy.choose(sum(abs(ar_data - ar_blue)) < 40, (ar_255,ar_0))
gtiff_drv = gdal.GetDriverByName('Gtiff')
out_filename = 'world_north_hemisphere.tif.msk'
out_ds = gtiff_drv.Create(out_filename,ds.RasterXSize,ds.RasterYSize)
# Those flags indicate that the mask if common to all bands and is an alpha
# mask (so we could potentially use values between 0 and 255 if needed
mask_flag = str(gdal.GMF_PER_DATASET + gdal.GMF_ALPHA)
out_ds.SetMetadataItem('INTERNAL_MASK_FLAGS_1', mask_flag)
out_ds.SetMetadataItem('INTERNAL_MASK_FLAGS_2', mask_flag)
out_ds.SetMetadataItem('INTERNAL_MASK_FLAGS_3', mask_flag)
out_ds.GetRasterBand(1).WriteArray(ar_res)
out_ds = None
```

Check:

```
$ gdalinfo world_north_hemisphere.tif
```

Output:

```
Driver: GTiff/GeoTIFF
Files: world_north_hemisphere.tif
       world_north_hemisphere.tif.msk
Size is 2048, 512
```

```
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"]]
Origin = (-180.000000000000000,90.000000000000000)
Pixel Size = (0.175781250000000,-0.175781250000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  INTERLEAVE=PIXEL
Corner Coordinates:
Upper Left  (-180.0000000,  90.0000000) (180d 0' 0.00"W, 90d 0' 0.00"N)
Lower Left  (-180.0000000,   0.0000000) (180d 0' 0.00"W,  0d 0' 0.01"N)
Upper Right ( 180.0000000,  90.0000000) (180d 0' 0.00"E, 90d 0' 0.00"N)
Lower Right ( 180.0000000,   0.0000000) (180d 0' 0.00"E,  0d 0' 0.01"N)
Center      (   0.0000000,  45.0000000) (  0d 0' 0.01"E, 45d 0' 0.00"N)
Band 1 Block=2048x1 Type=Byte, ColorInterp=Red
  Mask Flags: PER_DATASET ALPHA
Band 2 Block=2048x1 Type=Byte, ColorInterp=Green
  Mask Flags: PER_DATASET ALPHA
Band 3 Block=2048x1 Type=Byte, ColorInterp=Blue
  Mask Flags: PER_DATASET ALPHA
```

We see that the generate mask is now recognized by GDAL. We can now run (without -srcnodata flag) :

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -dstalpha \
            -ts 1024 0 -overwrite
```

Note that we have used « -ts 1024 0 ». The 0 here means that gdalwarp must determine the appropriate height from the specified target width to conserve the aspect ratio.

The result is now nearly perfect!

Use of cutline

The cutline is a polygon or a set of polygons that define the areas that must be updated by gdalwarp from the areas that must be let untouched. We are going to use the Natural Earth administrative polygons of countries to do the masking.

First delete or rename the previously generated mask (not strictly needed, but to demonstrate that the cutline works)

```
$ mv world_north_hemisphere.tif.msk world_north_hemisphere.tif.msk.dis
```

And then :

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -dstalpha \
            -ts 1024 1024 -overwrite -cutline ne_10m_admin_0_countries.shp
```

I declare this result perfect (taking into account the relatively low resolution of the source raster) :-)

Resampling

By default, when computing the result of the warp operation, gdalwarp will select only one source pixel for each target pixel. This results in the fastest computation, but sometimes in poor result. Let's do a few comparisons :

```
$ gdalwarp paris.tif paris_wgs84.tif -t_srs WGS84
$ gdalwarp paris.tif paris_wgs84_cubic.tif -t_srs WGS84 -r cubic
$ gdalwarp paris.tif paris_wgs84_lanczos.tif -t_srs WGS84 -r lanczos
```

And display the results in QGIS. Make sure to zoom to 100% to have a fair comparison. Generally Lanczos is considered to give the best results when they are text labels (and is also the slowest method)

Note that in GDAL 1.11, when downsizing an image by a factor of 2 or more, not all resampling methods are appropriately implemented to give the expected result. Only cubicspline, lanczos and average are (nearest can of course be used but leads to poor results by design). This has been corrected in GDAL 2.0 !

Speeding-up reprojection

- Computing a smaller output dataset with -ts target_width target_height. Try :

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
            -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -ts 1000 1000 -overwrite
```

- Using several CPU cores for the reprojection (this assumes that the computer has several cores (or if running in a Virtual Machine, that it has been assigned at least 2 virtual cores). The effect will be more important if heavy computations are involved (for example a resampling kernel like Lanczos, or for reprojections that are computation intensive, like when involving a UTM coordinate system)

```
$ gdalwarp world_north_hemisphere.tif north_pole.tif -t_srs EPSG:32661 \
     -wo SAMPLE_GRID=YES -wo SOURCE_EXTRA=32 -wo NUM_THREADS=ALL_CPUS -overwrite
```

- Parallelize computations and read/write operations in the datasets with the -multi option. This will give generally only a modest improvement, and mostly if the input dataset is compressed.
- By default, gdalwarp uses an approximate coordinate transformer with a target error threshold of at most 0.125 pixels, ie each reprojected point will be at most at 0.125 pixel of its exact position. If you increase that value with the -et error_threshold_value option, less reprojections will be done but at the expense of prevision. On the contrary, setting -et 0 will use the exact transformer. This is rarely needed, except if you use RPC warping with a digital elevation model, but we will not have time to discover this.

2.4 Mosaicing

2.4.1 Introduction

Depending on the context, several utilities can be used to do mosaicing :

- gdal_merge can be used when merging together datasets that each can fit into RAM and whose resolutions are relatively close. gdal_merge (as of GDAL 1.11) does not take into alpha channel to blend images that might overlap.
- gdalwarp is more powerful in that it can deal with arbitrary larget input datasets and with transparency. It is also slower at execution
- gdalbuildvrt will not produce a GeoTIFF but a virtual raster describing how the mosaic is assembled. This can be usefull to save storage space.

Documentation of the gdalwarp utility : http://gdal.org/1.11/gdalwarp.html

Documentation of the gdal_merge utility : http://gdal.org/1.11/gdal_merge.html

Documentation of the gdalbuildvrt utility : http://gdal.org/1.11/gdalbuildvrt.html

2.4.2 Using gdal_merge

Our 2 DEM tiles respect the optimal use condition of gdal_merge, so let's merge them :

```
$ gdal_merge.py  -o merged_dem.tif MK_30m.tif ML_30m.tif
```

2.4.3 Using gdalwarp

Let's try to merge wellington_west.tif and wellington_east_warped.tif

```
$ gdalwarp wellington_west.tif wellington_east_warped.tif wellington_merged.tif
```

Check at the result with QGIS. Hum, the texts of the east part are corrupted, and by comparing the colours of the middle zone, we can see that the land area has issues too (if you overlay the original wellington_east_warped.tif, this will be obvious)

What happened ? The issue is that we have merged 2 datasets that have different color tables. gdalwarp is not particularly aware of color tables, so it has picked up the first one and used it for both.

We will solve this by expanding the color table into 3 R,G,B bands

```
$ pct2rgb.py wellington_west.tif wellington_west_rgb.tif
$ gdal_translate wellington_east_warped.tif wellington_east_rgb.tif -expand rgb
```

Just for the sake of demonstrating the 2 possibilities. No particular advantage or drawback in using of of the 2, but gdal_translate -expand rgb can be combined with all other gdal_translate options.

And proceed again to the merge :

```
$ gdalwarp wellington_west_rgb.tif wellington_east_rgb.tif \
        wellington_merged_rgb.tif
```

Colors are now OK but we have this annoying black areas in the collar, the middle area stil being black and the text labels. We have to remove all that.

First let notice that the collar is a 100 pixel large zone, that can easily crop with gdal_translate :

```
$ gdal_translate wellington_west_rgb.tif wellington_west_rgb_cropped.vrt \
                 -srcwin 100 100 1631 1635 -of VRT
$ gdal_translate wellington_east_rgb.tif wellington_east_rgb_cropped.vrt \
                 -srcwin 100 100 2139 1850 -of VRT
```

Here we generate a virtual raster instead of a plain GeoTIFF since this is just an intermediate result.

Now let's merge them together :

```
$ gdalwarp wellington_east_rgb_cropped.vrt wellington_west_rgb_cropped.vrt \
           wellington_merged_rgb.tif -overwrite
```

The middle area is now OK.

An idea would be to use the contour of New Zealand with -cutline

```
$ gdalwarp -cutline ne_10m_admin_0_countries.shp -dstalpha \
           wellington_merged_rgb.tif wellington_merged_rgba.tif -overwrite
```

Ouch ! the resulting image is almost empty now. This is due to using all the world countries and reprojecting them to the New Zealand projection of wellington_merged_rgb.tif, causing invalid polygons to be generated. We can fix that by just selecting the New Zealand outline :

```
$ gdalwarp -cutline ne_10m_admin_0_countries.shp -cwhere "admin='New Zealand'" \
         -srcnodata "0 0 0" -dstalpha \
          wellington_merged_rgb.tif wellington_merged_rgba.tif
```

Not bad, but some land areas have been cut off, because this Natural Earth dataset is not precise enough when comparing with the resolution of our raster dataset. Using a more precise dataset would fix that.

Another way of removing the colars would be to use the nearblack utility. Here we want black and white areas of the collar to be transparent

```
$ nearblack -of gtiff -setalpha -color 0,0,0 -color 255,255,255 \
             wellington_merged_rgb.tif -o wellington_merged_rgba.tif
```

Documentation : http://gdal.org/1.11/nearblack.html

Note that nearblack is the only utility of the GDAL suite to not output to GeoTIFF by default, so we have to specify it explicitley. We also request it to create an alpha channel with -setalpha otherwise it would just set white as black.

Let's look at the result. Close to our hopes but there are still artifacts around the text labels. This is due to antialiasing. After quite a few iterations to determine the appropriate error threshold (-near 10) and list the intermediate colors, you can try :

```
$ nearblack -near 10 -setalpha -color 0,0,0 -color 25,25,25 -color 45,45,45 \
            -color 70,70,70 -color 90,90,90 -color 95,95,95 -color 115,115,115 \
            -color 130,130,130 -color 170,170,170 -color 180,180,180 \
            -color 195,195,195 -color 210,210,210 -color 225,225,225 \
            -color 240,240,240 -color 255,255,255 \
            wellington_merged_rgb.tif -o wellington_merged_rgba.tif -of gtiff
```

2.4.4 Using gdalbuildvrt

```
$ gdalbuildvrt merged_dem.vrt MK_30m.tif ML_30m.tif
```

Look at the generated merged_dem.vrt file. This is a XML file describing all the charactheristics of the dataset :

- dimensions
- coordinate system
- geo-transformation matrix
- bands

Several sources can be compositing to define the band pixel values.

Full documentation of the format can be found at http://www.gdal.org/gdal_vrttut.html

2.5 Image optimization

2.5.1 Tiling

For fast image exploration and serving, you generally want to tile the raster so that requesting a small window in it only needs accessing modest amount of data :

```
$ gdal_translate wellington_merged_rgba.tif tiled.tif -co TILED=YES
```

The default tile size of the GeoTIFF driver is 256x256. This can be changed with the -co BLOCKXSIZE=xxx and -co BLOCKYSIZE=yyy creation option, but this is rarely needed.

2.5.2 Overviews

For fast zoom out operations, we want to provide overviews. Generally overview factors are specified as successive power of two values. This is not an obligation, but a convenient practical rule. How to know to up to which level to limit ? One can consider that we can generate overviews until the smallest overview is less than a tile size (256x256 by default). Our merged TIFF is 3575x2460 large. And 3575/256=13.xxxx, so up to factor 16 should be fine.

```
$ gdaladdo -r average tiled.tif 2 4 8 16
```

Documentation : http://gdal.org/1.11/gdaladdo.html

On a GeoTIFF, gdaladdo will be default create the overviews as internal parts of the main image. Sometimes this is not desirable. You can build external overviews by asking the image to be opened in read-only mode with -ro :

```
$ gdal_translate wellington_merged_rgba.tif tiled.tif -co TILED=YES
$ gdaladdo -ro -r average tiled.tif 2 4 8 16
```

2.5.3 Compression

Lossless compression

On fast computers, reading from storage/disk might be the bottleneck, so compression can (sometimes) bring performance gain.

A lossless compression scheme is the DEFLATE algorithm (the one used in .zip file) :

```
$ gdal_translate wellington_merged_rgba.tif tiled_deflate.tif \
           -co TILED=YES -co COMPRESS=DEFLATE
```

Compare the file sizes : 4.1 MB for deflate vs 49.2 MB for uncompressed. Of course here, this is a dramatic improvement due to about half the image surface being transparent, so easily compressable. In more typical cases, you can expect ½ or 1/3 reduction.

Note : in case maximum interoperability is wished, one must be aware that DEFLATE compression is not universally supported by all software packages (should be OK with all using GDAL). LZW might be a safer choice, even if slighly less efficient regarding compression rate.

There are 2 potential ways of improving DEFLATE compression :

- specifying -co ZLEVEL=9 which is the maximum compression level possible. This can significantly slow down compression. Generally the compression improvement will be very modest (a few percent)
- using horizontal differencing with -co PREDICTOR=2 (for integer data types) or 3 (for floating-point data). Let's assume that the suite of pixel in a row is P1 P2 P3 P4 etc. Horizontal differencing consists in passing P1 P2-P1 P3-P2 P4-P3 etc.. to the compressor. This often (but not always) lead to further compression gain for continuous data with slow variations (so not worth trying on a dataset with a color table since consecutive values in a row, which are indices to the color table entries, are generally not at all related). Let try it on our mosaic :

```
$ gdal_translate wellington_merged_rgba.tif tiled_deflate_pred2.tif \
  -co TILED=YES -co COMPRESS=DEFLATE -co PREDICTOR=2
```

And now compare the file sizes of tiled_deflate.tif and tiled_deflate_pred2.tif. Ouch, the later is 50% larger than the former.

Let try on our merged DEM :

```
$ gdal_translate merged_dem.vrt merged_dem_tiled_deflate.tif -co TILED=YES \
                 -co COMPRESS=DEFLATE
$ gdal_translate merged_dem.vrt merged_dem_tiled_deflate_pred2.tif \
                 -co TILED=YES -co COMPRESS=DEFLATE  -co PREDICTOR=2
```

This time, horizontal differencing leads to a smaller file size (modest gain here, can sometimes be more significant like 20 or 30%). Conclusion is that PREDICTOR=2 should not be blindly used. If minimizing storage size is the objective and computation time is available, trying with or without it is therefore recommended.

Lossy compression

The TIFF format offers JPEG compression as the only relatively standard lossy compression scheme.

```
$ gdal_translate wellington_merged_rgba.tif tiled_jpeg.tif \
                -co TILED=YES -co COMPRESS=JPEG
```

The resulting tiled_jpeg.tif is 36 % smaller than tiled_deflate.tif

You can play with the -co QUALITY=xx creation option where xx is between 1 and 100. The default value used is 75, which is a generally satisfactory value. Lesser values will improve the compression rate at the expense of increasingly the visible « blocking » artifacts typical of the JPEG compression (visible as squares of 8x8 pixels)

When dealing with RGB data a useful improvement is to use the YCbCr colorspace (Y=luminance, Cb=blue chrominance, Cr=red chrominance) with a subsampling of factor 2 for the Cb and Cr channels since the human eye is more sensitive to changes in luminance rather than in chrominance. Let's try that :

```
$ gdal_translate wellington_merged_rgba.tif tiled_jpeg_ycbcr.tif \
                -co TILED=YES -co COMPRESS=JPEG -co PHOTOMETRIC=YCbCr
```

Output :

```
ERROR 6: PHOTOMETRIC=YCBCR requires a source raster with only 3 bands (RGB)
```

Indeed our dataset is RGBA not RGB, and due to the specification of the format, this is not possible.

A potential workaround is to only compress the RGB channels with JPEG and create a separate mask channel with the alpha channel.

```
$ gdal_translate wellington_merged_rgba.tif tiled_jpeg_ycbcr.tif \
           -co TILED=YES -co COMPRESS=JPEG -co PHOTOMETRIC=YCbCr \
           -b 1 -b 2 -b 3 -mask 4
$ gdalinfo tiled_jpeg_ycbcr.tif
```

Output:

```
Driver: GTiff/GeoTIFF
Files: tiled_jpeg_ycbcr.tif
       tiled_jpeg_ycbcr.tif.msk
[…]
Image Structure Metadata:
  COMPRESSION=YCbCr JPEG
  INTERLEAVE=PIXEL
  SOURCE_COLOR_SPACE=YCbCr
[…]
Band 1 Block=256x256 Type=Byte, ColorInterp=Red
  Mask Flags: PER_DATASET
```

```
Band 2 Block=256x256 Type=Byte, ColorInterp=Green
  Mask Flags: PER_DATASET
Band 3 Block=256x256 Type=Byte, ColorInterp=Blue
  Mask Flags: PER_DATASET
```

Seems to have worked. And when comparing the sizes of tiled_jpeg_ycbcr.tif + tiled_jpeg_ycbcr.tif.msk with tiled_jpeg.tif, we can see that they are 63% smaller. Let's display tiled_jpeg_ycbcr.tif in QGIS. You can see that transparency is not handled. This is because QGIS (and most software, with the exception of MapServer) only handles a regular alpha band, but not a mask band like the one we generated.

VRT power to the rescue ! : let's generate a virtual dataset that will pretent to have an alpha band

```
$ gdal_translate -of VRT tiled_jpeg_ycbcr.tif  tiled_jpeg_ycbcr_alpha.vrt \
                -b 1 -b 2 -b 3 -b mask
```

Open the VRT with QGIS. This time, this works as expected. If you open the .vrt file, you'll see <SourceBand>mask,1</SourceBand> in the source definition of the alpha band. This means « use the mask band of the first band » (which is the mask band of all bands here)

Since a few years, JPEG decompression on Intel platforms is much faster than before with the widespread use of libjpeg-turbo (at least on Linux platforms). Combined with the reduced file size, this results generally in faster or similar decompression time than DEFLATE, so if lossy compression is acceptable for the intended use of the data, JPEG is often a good choice.

2.5.4 Compression and overviews

Let's generate external overviews on a compressed raster

```
$ gdaladdo -ro merged_dem_tiled_deflate.tif 2 4 8
```

Ouch ! merged_dem_tiled_deflate.tif.ovr is larger than merged_dem_tiled_deflate.tif. This is due to external overviews not being compressed by default. In order to compress them, you have to use the COMPRESS_OVERVIEW configuration option

But priorly delete the existing overview merged_dem_tiled_deflate.tif.ovr

```
$ rm -f merged_dem_tiled_deflate.tif.ovr
$ gdaladdo -ro merged_dem_tiled_deflate.tif 2 4 8 \
          --config COMPRESS_OVERVIEW DEFLATE
```

Much better : the resulting overview is roughly 1/3 of the full resolution dataset, as expected.

For internal overviews, the compression of the main dataset will be used, so no special action required.

2.6 DEM processing

2.6.1 Introduction

gdaldem is a utility that enables the user to apply different processings on a digital elevation model :

- hypsometric rendering
- hillshading
- slope and aspect computations
- terrain ruggedness index (TRI), topographic position index(TPI), roughness

The gdal_contour utility can be used to compute contour lines (iso-altitude lines) from a DEM.

The gdal_rasterize utility creates a raster from a vector.

The gdal_fillnodata utility fills voids in a raster from neighbouring values.

The gdal_grid utility creates regular grid from scattered data.

Documentation of the gdaldem utility : http://gdal.org/1.11/gdaldem.html

Documentation of the gdal_contour utility : http://gdal.org/1.11/gdal_contour.html

Documentation of the gdal_rasterize utility : http://gdal.org/1.11/gdal_rasterize.html

Documentation of the gdal_fillnodata script : http://gdal.org/1.11/gdal_fillnodata.html

Documentation of the gdal_grid utility : http://gdal.org/1.11/gdal_grid.html

2.6.2 Hypsometric rendering

Hypsometric rendering produces a colored map from the elevation and a color ramp that defines which colors to apply to a range of altitude.

The color ramp is a simple text file with one line for a particular elevation value and the corresponding color. The first value on the line is the elevation (in the units of the DEM, typically meters) and the 3 or 4 following values are the RGB/RGBA values.

It is also possible to use percentage that are relative to the [min, max] range of the whole DEM.

For example create a color_relief.txt file with the following content :

```
0     50 100 255
0.01 110 220 110
25%  240 250 160
50%  230 220 170
75%  220 220 220
100% 250 250 250
```

Linear interpolation is done for intermediate elevations.

Now let's apply this color ramp to our merge DEM :

```
$ gdaldem color-relief merged_dem.vrt color_relief.txt colored.tif
```

Display it in QGIS.

Instead of RGB triplets, you can also use a few predefined color names : white, black, red, green, blue, yellow, magenta, cyan, aqua, grey/gray, orange, brown, purple/violet and indigo

The nv keyword can also be used to mean the nodata value

Note : the VRT format can be used as the output format for this algorithm of gdaldem.

2.6.3 Hillshading

Hillshading creates a shaded relief raster from a DEM. This works by simulating a light source that would be at an infinite distance and whose direction is by default the upper-left corner of the raster, with an altitude of 45°

```
$ gdaldem hillshade merged_dem.vrt hillshade.tif
```

There are many ways of tuning the result. The -combined option can be used and it will generate a result that is much brighter.

```
$ gdaldem hillshade -combined merged_dem.vrt hillshade_combined.tif
```

In case the DEM has very low slopes but you want to show them in a contrasted way, you can use the -z z_factor option to multiply the DEM values by z_factor in the hillshading computation.

```
$ gdaldem hillshade -z 2 merged_dem.vrt hillshade.tif
```

Display the results in QGIs and compare.

2.6.4 Colored hillshaded

Sometimes you want to have a visual representation of both the altitude range and the relief slope.

This is possible with the hsv_merge.py Python script (http://svn.osgeo.org/gdal/trunk/gdal/swig/python/samples/hsv_merge.py). This scritpt takes a RGB or RGBA raster as first parameter, a single-band raster as second parameter (with Byte datatype) and the output raster as last parameter. It transforms each pixel of the first raster from the RGB color space to the HSV (Hue Saturation Value) color space, and substitude the value (which represents the brightness/intensity) with the corresponding value from the second raster.

```
$ python hsv_merge.py colored.tif hillshade_combined.tif colored_hillshade.tif
```

And display it in QGIS.

This type of raster is often an excellent basemap to display vectors on top of it.

2.6.5 Contour lines

We can generate a set of contour lines every 50 m of altitude with the following :

```
$ gdal_contour -i 50 merged_dem.vrt dem_contours.shp
```

Display it on top of the colored hillshade in QGIS

Select a line and look at its attributes. Hum, only the id, but not the elevation. We have to request it explicitely

```
$ rm -f dem_contours.*
$ gdal_contour -a elevation -i 50 merged_dem.vrt dem_contours.shp
```

2.6.6 Building a DEM from a set of contour lines

Let's generate contour lines with a vertical step of 10 meters. This time we use the -3d option to ask for the altitude to be included as the z component of each vertex of the lines.

```
$ gdal_contour -i 10 -3d merged_dem.vrt dem_contours_10m.shp
```

And assume this was the only input we have. We are now going to build a DEM from that.

```
$ gdal_rasterize -3d dem_contours_10m.shp dem_from_contours.tif
```

This emits an error mentionning that dem_from_contours.tif does not exist. Indeed, we have to provide a few hints to gdal_rasterize to initialize the raster. The very minimum is the resolution or the width/height. We may also be able to specify the wished geospatial extent In order to be able to compare the result with our origin DEM, we are going to use exactly its extent and resolution :

```
$ gdal_rasterize -3d -tr 30 30 -te  1703936 5373938 1835036 5439488 \
                   dem_contours_10m.shp dem_from_contours.tif
```

Let's look at the result with QGIS. Not bad, but there are a number of zones with missing values (in flat areas where contour lines are very separate)

We are going to interpolate the missing values with gdal_fillnodata.py

```
$ gdal_fillnodata.py dem_from_contours.tif dem_from_contours_filled.tif
```

Let's display it and compare with dem_from_contours.tif. Hum, they are the same ! Indeed, gdal_fillnodata.py only touches pixels which are at the nodata value, but there's none for now. So let's set the nodata value on dem_from_contours.tif.

```
$ gdal_edit.py -a_nodata 0 dem_from_contours.tif
```

gdal_edit.py is a convenient script that can be used to modify in-place (for drivers that support in-place update, such as Gtiff) geotransform matrix, coordinate system, nodata value, metadata.

Documentation at http://gdal.org/1.11/gdal_edit.html

And re-run :

```
$ gdal_fillnodata.py dem_from_contours.tif dem_from_contours_filled.tif
```

Look at the result. Pretty close to the original DEM ! But can we measure more precisely the error :

We can use the gdal_calc.py script for that :

```
$ gdal_calc.py -A merged_dem.vrt -B dem_from_contours_filled.tif \
                      --calc="A-B" –outfile=diff.tif
```

And look at the statistics :

```
$ gdalinfo -stats diff.tif
```

So the result is relatively unbiased (mean of difference less than one meter), but with errors up to 25 m. How can we explain that ? Probably due to the fact that the there are some pixels of the original DEM that are crossed by more than one contour line (you can see it zooming in strongly), so when burning the vector, one of this line will be arbitrarily burnt. A technique to improve this would be rasterize to a much higher resolution, do the void filling and then resample with averaging. Doing that with a horizontal resolution of 10 m reduce the maximum error to 20 m, and with an improved mean error.

2.6.7 Gridding

Gridding is the operation that consists in creating a raster from a grid of sparse points and interpolating values in the gaps.

First let's being by creating a random set of 10000 points by running the following Python snippet:

```
from osgeo import gdal
import random
random.seed(0) # initialize so that everybody has the same random sampling;-)
ds = gdal.Open('merged_dem.vrt')
ar = ds.ReadAsArray()
gt = ds.GetGeoTransform()
f = open('points.csv', 'wt')
f.write('X,Y,Z\n')
for i in range(10000):
  while True:
    x = random.randrange(0,ds.RasterXSize)
    y = random.randrange(0,ds.RasterYSize)
    val = ar[y,x]
    if val != 0: # only collect non-zero elevations
      (X, Y) = gdal.ApplyGeoTransform(gt, x, y)
      f.write('%f,%f,%f\n' % (X, Y, val))
    break


f.close()
```

Convert this CSV file to a Spatialite database and build a Point geometry :

```
$ ogr2ogr points.db points.csv  -sql "SELECT MakePoint(CAST(x AS FLOAT), \
 CAST(y AS FLOAT)), CAST(z AS FLOAT) AS Z FROM points" -dialect sqlite \
 -dsco spatialite=yes -f SQLite -a_srs EPSG:2193 -nln points
```

Now proceed to the gridding operation (note this is a slow operation) :

```
$ gdal_grid -zfield z -txe 1703936 1835036 -tye 5439488 5373938 \
      -a_srs EPSG:2193 -outsize 4370 2185 -l points points.db gridded_pow1.tif \
      -a invdist:power=1:smoothing=10:radius1=1000:radius2=1000
```

As an exercise, compare the results with what gdal_rasterize + gdal_fillnodata would give.

2.7 Virtual System Interface – Access to http://, .zip resources

Up to know we have dealt with local rasters. GDAL has a subsystem called VSI (presumably for Virtual System Interface) that enable the user to access transparently data inside archives (/vsizip/

for .zip, /vsitar/ for .tar or .tar.gz archives), in-memory file (/vsimem) or web ressources accessible through HTTP, HTTPS or FTP (/vsicurl/)

2.7.1 /vsicurl/

The European Commission has released a DEM of Europe with a 1 arc-second accuracy : http://www.eea.europa.eu/data-and-maps/data/eu-dem#tab-original-data

The whole file http://published-files.eea.europa.eu/eudem/jrc_r_3035_25_m_gsgrda-eudem-dem-europe_2012_rev1/eudem_dem_3035_europe.tif is 22 GB large.

Thanks to /vsicurl/ and gdalinfo, we can quickly retrieves its metadata :

```
$ gdalinfo /vsicurl/http://published-files.eea.europa.eu/eudem/\
jrc_r_3035_25_m_gsgrda-eudem-dem-europe_2012_rev1/eudem_dem_3035_europe.tif
```

Output :

```
Driver: Gtiff/GeoTIFF
Files: /vsicurl/http://published-
files.eea.europa.eu/eudem/jrc_r_3035_25_m_gsgrda-eudem-dem-
europe_2012_rev1/eudem_dem_3035_europe.tif
 /vsicurl/http://published-files.eea.europa.eu/eudem/jrc_r_3035_25_m_gsgrda-
eudem-dem-europe_2012_rev1/eudem_dem_3035_europe.tif.ovr
Size is 240000, 200000
Coordinate System is:
PROJCS["ETRS89 / LAEA Europe",
 GEOGCS["ETRS89",
 DATUM["European_Terrestrial_Reference_System_1989",
 SPHEROID["GRS 1980",6378137,298.2572221010002,
 AUTHORITY["EPSG","7019"]],
 AUTHORITY["EPSG","6258"]],
 PRIMEM["Greenwich",0],
 UNIT["degree",0.0174532925199433],
 AUTHORITY["EPSG","4258"]],
 PROJECTION["Lambert_Azimuthal_Equal_Area"],
 PARAMETER["latitude_of_center",52],
 PARAMETER["longitude_of_center",10],
 PARAMETER["false_easting",4321000],
 PARAMETER["false_northing",3210000],
 UNIT["metre",1,
 AUTHORITY["EPSG","9001"]],
 AUTHORITY["EPSG","3035"]]
Origin = (2000000.000000000000000,6000000.000000000000000)
Pixel Size = (25.000000000000000,-25.000000000000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  ( 2000000.000, 6000000.000) ( 51d50'39.04"W, 66d46'12.03"N)
Lower Left  ( 2000000.000, 1000000.000) ( 13d41' 3.88"W, 28d46'45.49"N)
Upper Right ( 8000000.000, 6000000.000) ( 86d 0'31.83"E, 56d26'45.47"N)
Lower Right ( 8000000.000, 1000000.000) ( 46d30'12.69"E, 24d 4'54.00"N)
Center      ( 5000000.000, 3500000.000) ( 20d26' 6.96"E, 54d 9'53.47"N)
Band 1 Block=256x256 Type=Float32, ColorInterp=Gray
  Min=-226.887 Max=5111.089
  Minimum=-226.887, Maximum=5111.089, Mean=341.729, StdDev=493.987
  NoData Value=nan
  Overviews: 60000x50000, 30000x25000, 15000x12500, 7500x6250, 3750x3125, 
1875x1563
  Metadata:
    STATISTICS_MAXIMUM=5111.0888671875
    STATISTICS_MEAN=341.72920322335
    STATISTICS_MINIMUM=-226.8865814209
    STATISTICS_STDDEV=493.98749395974
```

We can do better that just getting metadata. We can actually get data. For example Como elevation. First, for Linux users, we can use some syntaxic sugar and create a symbolic link to our virtual file, with :

```
$ ln -s /vsicurl/http://published-files.eea.europa.eu/eudem/\
jrc_r_3035_25_m_gsgrda-eudem-dem-europe_2012_rev1/eudem_dem_3035_europe.tif
```

And now query the raster (Windows users will have to paste the whole /vsicurl/... string instead)

```
$ gdallocationinfo -wgs84 eudem_dem_3035_europe.tif 9.0863028 45.8106805
```

Output (within a few seconds) :

```
Report:
  Location: (89995P,139102L)
  Band 1:
    Value: 204.521697998047
```

Since the raster has overviews, we could also get a subset at lower resolution on France for example

```
$ gdal_translate eudem_dem_3035_europe.tif out.tif -projwin 3074458 3145886 \
                 4315029 2030485 -outsize 1% 1%
```

2.7.2 /vsizip/ (+ /vsicurl/)

The USGS and NGA have released a Global Multi-resolution Terrain Elevation Data 2010 (GMTED2010), with resolutions down to 7.5 arc-seconds. This is http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be75_grd.zip which is 2.8 GB large.

Instead of downloading the whole archive, we are going to explore its content with the gdal_ls.py utility : http://svn.osgeo.org/gdal/trunk/gdal/swig/python/samples/gdal_ls.py

And now run it on the zip file :

```
$ python gdal_ls.py -R 
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/\
be75_grd.zip
```

Output (truncated) :

```
/
vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be75
_grd.zip/be75_grd/
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be7
```

```
5_grd.zip/be75_grd/dblbnd.adf
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be7
5_grd.zip/be75_grd/hdr.adf
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be7
5_grd.zip/be75_grd/prj.adf
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be7
5_grd.zip/be75_grd/sta.adf
[...]
```

We recognize a ArcInfo Binary grid, that we can now query with gdalinfo :

```
$ gdalinfo /vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/\
Grid_ZipFiles/be75_grd.zip/be75_grd -nofl -norat
```

Within a few seconds, we get the following output :

```
Driver: AIG/Arc/Info Binary Grid
Files:
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/be7
5_grd.zip/be75_grd
Size is 172800, 67200
Coordinate System is:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        TOWGS84[0,0,0,0,0,0,0],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9108"]],
    AUTHORITY["EPSG","4326"]]
Origin = (-180.000138888888927,83.999861111111059)
Pixel Size = (0.002083333333333,-0.002083333333333)
Corner Coordinates:
Upper Left  (-180.0001389,  83.9998611) (180d 0' 0.50"W, 83d59'59.50"N)
Lower Left  (-180.0001389, -56.0001389) (180d 0' 0.50"W, 56d 0' 0.50"S)
Upper Right ( 179.9998611,  83.9998611) (179d59'59.50"E, 83d59'59.50"N)
Lower Right ( 179.9998611, -56.0001389) (179d59'59.50"E, 56d 0' 0.50"S)
Center      (  -0.0001389,  13.9998611) (  0d 0' 0.50"W, 13d59'59.50"N)
Band 1 Block=256x16 Type=Int16, ColorInterp=Undefined
  Min=-606.000 Max=8746.000
  NoData Value=-32768
```

Let's get Wellington elevation:

```
$ gdallocationinfo
/vsizip/vsicurl/http://topotools.cr.usgs.gov/gmted_viewer/data/Grid_ZipFiles/\
be75_grd.zip/be75_grd -geoloc 174.7772239 -41.2887639
```

Output (roughly after one minute) :

```
Report:
  Location: (170293P,60138L)
  Band 1:
    Value: 12
```

Performance is less impressive as in previous case since we have to seek in arbitrary location in files up to 360 MB, and arbitrary seeking within a file in a ZIP is not possible.

2.8 Virtual memory mapping (Linux only)

This features enables to access the raster values of a GDAL dataset as a C/C++ array (or a NumPy array from Python) in a transparent way without loading the whole dataset into memory. It is consequently aimed at big datasets, potentially multi gigabytes. For 32-bit systems, only datasets of at most 2 GB can in practice be handled. On 64-bit systems, the limit goes up to several terabytes. This feature has only be implemented on Linux systems for now (could possibly be ported on Windows and MacOSX). Extensive details can be found at https://trac.osgeo.org/gdal/wiki/rfc45_virtualmem

Of course, you can always access to dataset regions with ds.ReadAsArray(xoff, yoff, xsize, ysize). The virtual memory array mechanism has the advantage of not having to bother about the needed window in advance, and saving the overhead of this call.

For the sake of this workshop, we will only demonstrate it on one of our relatively small datasets.

For example, run this Python snippet :

```
from osgeo import gdal
ds = gdal.Open('wellington_west_rgb.tif')
ar = ds.GetVirtualMemArray()
print(ar[0].mean())
ar  = None # drop the reference before closing the dataset
ds = None
```

ds.GetVirtualMemArray() just returns a standard NumPy array, except that it returns it instantaneously without reading any pixel values. They are only accessed on a per-request basis.

For best performance, it might be better to tweak options of GetVirtualMemArray() with the actual organization of the file. For example, this file has pixel interleaving so we might instead pass band_sequential = False as an option of GetVirtualMemArray() , so that an element is accessed with array[y][x][band]

```
from osgeo import gdal
ds = gdal.Open('wellington_west_rgb.tif')
ar = ds.GetVirtualMemArray(band_sequential = False)
print(ar[:,:,0].mean())
ar  = None # drop the reference before closing the dataset
ds = None
```

For a tiled dataset, we can also ask for a tiled array with GetTiledVirtualMemArray(). The actual shape of the array can be determined with the tilexsize, tileysize and tile_organization parameters. By default they default to tilexsize=256, tileysize=256, and tile_organization = gdal.GTO_BSQ. The various possible tile organizations are :

- gdal.GTO_ BSQ --> an element is accessed with array[band][tiley][tilex][y][x].
- gdal.GTO_TIP --> an element is accessed with array[tiley][tilex][y][x][band].
- gdal.GTO_BIT --> an element is accessed with array[tiley][tilex][band][y][x].

If there is only one band, all organizations are equivalent and an element is accessed with array[tiley][tilex][y][x]

For best performance, tile size should ideally be set to the actual block size, and gdal.GTO_BSQ used for band interleaved data and gdal.GTO_TIP/gdal.GTO_BIT for pixel-interleaved data

```
from osgeo import gdal
ds = gdal.Open('wellington_west_rgb.tif')
ar = ds.GetTiledVirtualMemArray()
print(ar[0].mean())
ar  = None # drop the reference before closing the dataset
ds = None
```

Note that you don't get the expected result here. This is because the raster size is not a multiple of the selected tile dimensions. Consequently right-most and bottom-tile mosts are zero filled. So this API must be used with that in mind.

For both GetVirtualMemArray() and GetTiledVirtualMemArray(), the areas fetched and their neighbouring pixels are kept into a RAM cache with a least-recently used strategy. The default cache size is 10 MB and can be tuned with the cache_size option (in bytes)

3 Vector operations

3.1 Getting metadata about a vector dataset and requesting it

ogrinfo is the utility you will use all the time to discover metadata about a vector dataset, as well as requesting it

Documentation of the ogrinfo utility : http://gdal.org/1.11/ogrinfo.html

Let's begin with the most basic use :

```
$ ogrinfo -ro ne_10m_admin_0_countries.shp
```

Output :

```
INFO: Open of `ne_10m_admin_0_countries.shp'
      using driver `ESRI Shapefile' successful.
1: ne_10m_admin_0_countries (Polygon)
```

This list the different vector layers available, giving their name and their geometry type. In that instance, there is a single layer with Polygon geometries. Shapefiles have a single layer, but the OGR shapefile driver can handle directory of shapefiles, in which case multiple layers can appear.

Note : the -ro flag is not strictly needed, but explicitely asks for the dataset to be opened in read-only mode. This can avoid a warning message if the dataset cannot be opened in update mode due to insufficient permissions.

To get more information, we can ask for the summary-only mode (with -so), and list explicitely the layer names which we are interested in, or specify -al for all.

```
$ ogrinfo -ro -so -al ne_10m_admin_0_countries.shp
```

Output :

```
INFO: Open of `ne_10m_admin_0_countries.shp'
 using driver `ESRI Shapefile' successful.

Layer name: ne_10m_admin_0_countries
Geometry: Polygon
Feature Count: 254
Extent: (-180.000000, -90.000000) - (180.000000, 83.634101)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
 DATUM["WGS_1984",
 SPHEROID["WGS_84",6378137.0,298.257223563]],
 PRIMEM["Greenwich",0.0],
 UNIT["Degree",0.0174532925199433]]
scalerank: Integer (4.0)
featurecla: String (30.0)
labelrank: Real (16.6)
sovereignt: String (254.0)
sov_a3: String (254.0)
adm0_dif: Real (16.6)
level: Real (16.6)
type: String (254.0)
admin: String (254.0)
adm0_a3: String (254.0)
geou_dif: Real (16.6)
geounit: String (254.0)
gu_a3: String (254.0)
su_dif: Real (16.6)
subunit: String (254.0)
su_a3: String (254.0)
brk_diff: Real (16.6)
name: String (254.0)
name_long: String (254.0)
brk_a3: String (254.0)
brk_name: String (254.0)
brk_group: String (254.0)
abbrev: String (254.0)
postal: String (254.0)
formal_en: String (254.0)
formal_fr: String (254.0)
note_adm0: String (254.0)
note_brk: String (254.0)
name_sort: String (254.0)
name_alt: String (254.0)
mapcolor7: Real (16.6)
mapcolor8: Real (16.6)
mapcolor9: Real (16.6)
mapcolor13: Real (16.6)
pop_est: Real (16.6)
gdp_md_est: Real (16.6)
pop_year: Real (16.6)
lastcensus: Real (16.6)
gdp_year: Real (16.6)
economy: String (254.0)
income_grp: String (254.0)
wikipedia: Real (16.6)
fips_10: String (254.0)
iso_a2: String (254.0)
iso_a3: String (254.0)
iso_n3: String (254.0)
```

```
un_a3: String (254.0)
wb_a2: String (254.0)
wb_a3: String (254.0)
woe_id: Real (16.6)
adm0_a3_is: String (254.0)
adm0_a3_us: String (254.0)
adm0_a3_un: Real (16.6)
adm0_a3_wb: Real (16.6)
continent: String (254.0)
region_un: String (254.0)
subregion: String (254.0)
region_wb: String (254.0)
name_len: Real (16.6)
long_len: Real (16.6)
abbrev_len: Real (16.6)
tiny: Real (16.6)
homepart: Real (16.6)
```

We get extra information :

- number of features
- spatial extent in (xmin, ymin) – (xmax, ymax) format
- coordinate system
- list of attributes with their name, type and width.precision

To get similar information in Python, run the following snippet:

```
from osgeo import ogr
ds = ogr.Open('ne_10m_admin_0_countries.shp')
for lyr in ds:
    print("Layer : %s" % lyr.GetName())
    print("Feature count : %d" % lyr.GetFeatureCount())
    (xmin, xmax, ymin, ymax) = lyr.GetExtent() # not in same order as ogrinfo
    print("Extent : (%f, %f) - (%f %f)" % (xmin, ymin, xmax, ymax))
    print("Geometry type: %s" % ogr.GeometryTypeToName(lyr.GetGeomType()))
    srs = lyr.GetSpatialRef()
    if srs is not None:
        print("SRS: %s"% srs.ExportToWkt())
    lyr_defn = lyr.GetLayerDefn()
    for i in range(lyr_defn.GetFieldCount()):
        field_defn = lyr_defn.GetFieldDefn(i)
        name = field_defn.GetName()
        type = ogr.GetFieldTypeName(field_defn.GetType())
        width = field_defn.GetWidth()
        prec = field_defn.GetPrecision()
        print('Field %s, type %s (%d.%d)' % (name, type, width, prec))
```

You can also get the full list of features by removing the -so option :

```
$ ogrinfo -ro -al ne_10m_admin_0_countries.shp
```

Very long output not included here (you can kill the process with Ctrl+C). You can make it shorter by asking for geometries not to be displayed with the -geom=no option

```
$ ogrinfo -ro -al -geom=no ne_10m_admin_0_countries.shp
```

For similar output in Python:

```
from osgeo import ogr
ds = ogr.Open('ne_10m_admin_0_countries.shp')
lyr = ds.GetLayerByName('ne_10m_admin_0_countries')
lyr_defn = lyr.GetLayerDefn()
for feat in lyr:
    print('Feature %d' % feat.GetFID())
    for i in range(feat.GetFieldCount()):
        if feat.IsFieldSet(i):
            print('Field %s: %s' % (feat.GetFieldDefnRef(i).GetName(), \
                                    str(feat.GetField(i))))
    geom = feat.GetGeometryRef()
    if geom is not None:
        print('Geometry: %s...' % geom.ExportToWkt()[0:60]) # truncated geometry
    print('')
```

You can also filter with a SQL expression on the fields.

```
$ ogrinfo -ro -al -geom=no ne_10m_admin_0_countries.shp \
        -where "admin = 'France'"
```

You can also filter on a bounding-box with -spat xmin ymin xmax ymax. Only features that intersect this bounding-box will be returned.

```
$ ogrinfo -ro -al -geom=no ne_10m_admin_0_countries.shp -spat 2 49 3 50
```

The speed of attribute or spatial filters depend on the size of the dataset, and if the dataset has indexes for the fields being requested.

In that instance, there is a ne_10m_admin_0_countries.qix file, which is a GDAL specific format for ESRI Shapefile spatial index (also recognized by MapServer).

You can delete it and recreate it with :

```
$ rm -f ne_10m_admin_0_countries.qix
$ ogrinfo  ne_10m_admin_0_countries.shp -sql \
          "CREATE SPATIAL INDEX ON ne_10m_admin_0_countries"
```

This special syntax only works on shapefiles and is documented in http://gdal.org/1.11/ogr/drv_shapefile.html

Similarly, we can create an index on the admin attribute with :

```
$ ogrinfo ne_10m_admin_0_countries.shp \
          -sql "CREATE INDEX ON ne_10m_admin_0_countries USING admin"
```

Two new files with extension .idm (index metadata, a XML file listing the available indices) and .idm (the index itself) will be created. In that example, the dataset being of modest size, building indices will not bring dramatic performance improvement.

The -sql option can be used to specify a full SELECT SQL statement. For example, let's find the countries with more than 50 million inhabitants and order it by descreasing population, and compute a pseudo density (in inhabitants / square degrees) :

```
$ ogrinfo ne_10m_admin_0_countries.shp -geom=no -sql \
   "SELECT admin, continent, pop_est, ogr_geom_area, \
           pop_est / ogr_geom_area AS pseudo_density \
    FROM ne_10m_admin_0_countries WHERE pop_est > 50e6 ORDER BY pop_est desc"
```

Note that the geometry field is automatically retrieved (it is not mentionned in the list of fields of the SELECT). This behaviour is particular to the OGR SQL language. In particular when using the SQLite dialect, the geometry field must be explicitely requested.

Equivalent in Python:

```
from osgeo import ogr
ds = ogr.Open('ne_10m_admin_0_countries.shp')
sql = "SELECT admin, continent, pop_est, ogr_geom_area, "
sql += "      pop_est / ogr_geom_area AS pseudo_density "
sql += "FROM ne_10m_admin_0_countries WHERE pop_est > 50e6 "
sql += "ORDER BY pop_est desc"
lyr = ds.ExecuteSQL(sql)
lyr_defn = lyr.GetLayerDefn()
for feat in lyr:
    print('Feature %d' % feat.GetFID())
    for i in range(feat.GetFieldCount()):
        if feat.IsFieldSet(i):
            name = feat.GetFieldDefnRef(i).GetName()
            print('Field %s: %s' % (name, str(feat.GetField(i))))
    print('')

ds.ReleaseResultSet(lyr) # must be explicitely released to free memory
```

The SELECT statements supported by OGR are a subset of the full SQL 92 standard. It is documented at http://gdal.org/1.11/ogr/ogr_sql.html

Simple joins (by attributes) can be done. For example, let's create the following CSV file, capitals.csv, that associates the country name with its city :

```
country,capital
China,Beijing
France,Paris
India,Mumbay
United States of America,Washington
Indonesia,Jakarta
```

And now run:

```
$ ogrinfo ne_10m_admin_0_countries.shp -geom=no -sql \
   "SELECT admin, capital, continent, pop_est, ogr_geom_area, \
           pop_est / ogr_geom_area AS pseudo_density \
    FROM ne_10m_admin_0_countries countries \
    JOIN 'capitals.csv'.capitals \
    ON countries.admin = capitals.country \
    WHERE pop_est > 50e6 ORDER BY pop_est desc"
```

A Python port of ogrinfo is also available at http://svn.osgeo.org/gdal/branches/1.11/gdal/swig/python/samples/ogrinfo.py

3.2 Converting vector datasets

ogr2ogr is the utility to be used to convert between vector formats and/or do reprojections.

Documentation of the ogr2ogr utility : http://gdal.org/1.11/ogr2ogr.html

To start with, let's convert from Shapefile to Shapefile and use a projection that preserves areas such as Lambert Azimuthal Equal Area with :

```
$ ogr2ogr out.shp ne_10m_admin_0_countries.shp -t_srs "+proj=laea"
```

Note that the target dataset is placed before the source dataset (contrary to gdal_translate or gdalwarp).

We now can compute an exact surface in km² and a density:

```
$ ogr2ogr countries_with_density.shp out.shp -sql \
   "SELECT admin, capital, continent, pop_est, \
           ogr_geom_area / 1e6 as surf_km2, \
           pop_est / ogr_geom_area * 1e6 AS densitykm2 \
    FROM out countries \
    JOIN 'capitals.csv'.capitals \
    ON countries.admin = capitals.country \
    WHERE pop_est > 50e6 ORDER BY pop_est desc"
```

You may have noticed the following warnings:

```
Warning 1: Value 1338612970 of field pop_est of feature 41 not successfully
written. Possibly due to too larger number with respect to field width
Warning 1: Value 1166079220 of field pop_est of feature 103 not successfully
written. Possibly due to too larger number with respect to field width
```

They are due to the declared field width of the input dataset being one character too small to output with 6 decimal figures. This can be ignored in that context since only integer values make sense for the population. This is one of the annoyance of the Shapefile format that stores number under decimal form.

Let us proceed to the conversion to the Spatialite format (http://gdal.org/1.11/ogr/drv_sqlite.html) :

```
$ ogr2ogr -f SQLite countries.db ne_10m_admin_0_countries.shp \
          -dsco SPATIALITE=YES -nln countries
```

« -f SQLite » asks to convert to SQLite format

-dsco SPATIALITE=YES is a DataSet Creation Option specific of the SQLite driver

-nln countries is New Layer Name

Actually the above command fails with :

```
ERROR 1: sqlite3_step() failed:
  constraint failed (19)
ERROR 1: ROLLBACK transaction failed: cannot rollback - no transaction is active
ERROR 1: Unable to write feature 2 from layer ne_10m_admin_0_countries.

ERROR 1: Terminating translation prematurely after failed
translation of layer ne_10m_admin_0_countries (use -skipfailures to skip errors)
```

This is due to shapefiles mixing indifferently polygons and multipolygons. Other data formats such as Spatialite, GeoPackage or PostGIS do not allow such mix of geometry types.

This can be easily solved by asking explicitely conversion to the appropriate Multi type with -nlt PROMOTE_TO_MULTI (in that instance -nlt MULTIPOLYGON) :

```
$ ogr2ogr -f SQLite countries.db ne_10m_admin_0_countries.shp \
          -dsco SPATIALITE=YES -nln countries -nlt PROMOTE_TO_MULTI -overwrite
```

The -overwrite flag is needed to recreate the « countries » table that already exists.

A Python port of ogr2ogr is also available at http://svn.osgeo.org/gdal/branches/1.11/gdal/swig/python/samples/ogr2ogr.py

3.3 SQLite dialect

As we mentionned before, the SQL natively supported by OGR is a subset of the SQL 92 standard. It is possible to also use the SQL engine of the SQLite database, with the spatial extensions offered by Spatialite. This is documented on http://gdal.org/1.11/ogr/ogr_sql_sqlite.html

The list of spatial functions available can be found at http://www.gaia-gis.it/gaia-sins/libspatialite-4.1.0-RC1/spatialite-sql-4.1.0.html

To enable the use of the SQLite dialect, « -dialect SQLite » must be passed on the ogrinfo or ogr2ogr command lines.

For example, let's try to compute the average distance between the capitals of the same continent. We have not the coordinates of the capitals, but we can use ST_PointOnSurface() that will compute a pseudo-centroid, that is guaranteed to be on the surface (contrary to ST_Centroid())

```
$ ogr2ogr centroids.shp ne_10m_admin_0_countries.shp -dialect SQLite -sql \
 "SELECT ST_PointOnSurface(geometry), admin, continent FROM \
 ne_10m_admin_0_countries"
```

Now we can run :

```
$ ogrinfo -ro -q centroids.shp -dialect SQLite -sql "SELECT c1.continent, \
     AVG(ST_Distance(c1.geometry, c2.geometry, 1) / 1e3) AS distance_km FROM \
     centroids c1, centroids c2 ON c1.continent = c2.continent  AND \
     c1.admin < c2.admin GROUP BY c1.continent ORDER BY distance_km DESC"
```

And get the following results :

```
Layer name: SELECT
OGRFeature(SELECT):0
 continent (String) = Seven seas (open ocean)
 distance_km (Real) = 7795.27243529042

OGRFeature(SELECT):1
 continent (String) = Asia
 distance_km (Real) = 4260.92149049736

OGRFeature(SELECT):2
 continent (String) = Oceania
 distance_km (Real) = 3837.78945728018

OGRFeature(SELECT):3
 continent (String) = Africa
 distance_km (Real) = 3427.40131119273

OGRFeature(SELECT):4
 continent (String) = South America
 distance_km (Real) = 2921.30044937971

OGRFeature(SELECT):5
 continent (String) = North America
 distance_km (Real) = 2100.82141445802

OGRFeature(SELECT):6
 continent (String) = Europe
 distance_km (Real) = 1667.27499916546
```

SQLite dialect is also available in Python by specifying dialect = 'SQLite' to the ExecuteSQL() method.

```
from osgeo import ogr
ds = ogr.Open('centroids.shp')
sql = "SELECT c1.continent, "
sql += "AVG(ST_Distance(c1.geometry, c2.geometry, 1) / 1e3) AS distance_km "
sql += "FROM centroids c1, centroids c2 ON c1.continent = c2.continent  AND "
sql += "c1.admin < c2.admin GROUP BY c1.continent ORDER BY distance_km DESC"
lyr = ds.ExecuteSQL(sql, dialect = 'SQLite')
for feat in lyr:
    feat.DumpReadable()

ds.ReleaseResultSet(lyr) # must be explicitely released to free memory
```

3.4 Geocoding API

Foreword: geocoding services are generally subject to term of uses that you must comply with. For the OpenStreetMap Nominatim service, this implementation will make sure that no more than one request is sent by second, but there might be other restrictions that you must follow by other means.

Let's create the following file, como.csv :

```
id,placename
1,"Politecnico Di Milano, Como, Italy"
2,"Hotel Como, Como, Italy"
```

```
3,"Hotel Tre Re, Como, Italy"
```

And now run:

```
$ ogrinfo como.csv -q -dialect sqlite -sql \
 "SELECT ST_Centroid(ogr_geocode(placename)) AS geom, \
 ogr_geocode_reverse(ST_Centroid(ogr_geocode(placename)),'display_name') \
 AS full_address FROM como"
```

Output :

```
Layer name: SELECT
OGRFeature(SELECT):0
 full_address (String) = Politecnico di Milano, 42, Via Francesco Anzani, Lora,
Como, CO, LOM, 22100, Italia
 POINT (9.092718606205288 45.801190422897783)

OGRFeature(SELECT):1
 full_address (String) = Hotel Como, Viale Giulio Cesare, Camerlata, Como,
CO,LOM, 22034, Italia
 POINT (9.089716011122285 45.801943148453489)

OGRFeature(SELECT):2
 full_address (String) = Hotel Tre Re, Via Vittani, Camerlata, Como, CO, LOM,
22034, Italia
 POINT (9.0814552 45.8111478)
```

Run it again. The answer is instanteously. This is becaused they are cached into a file called ogr_geocode_cache.sqlite.

'display_name' is the full address, but more structured reverse geocoding can be achieved by explicitely requesting a particular field of the output result :

```
$ ogrinfo :memory: -q -sql  \
 "SELECT ogr_geocode_reverse(2, 49, 'raw') as raw_xml, \
        ogr_geocode_reverse(2, 49, 'road') as road,\
        ogr_geocode_reverse(2, 49, 'postcode') as postcode,
        ogr_geocode_reverse(2, 49, 'town') as town, \
        ogr_geocode_reverse(2, 49, 'country') as country"
```

The use of :memory: is a conveniency, since :memory: is a in-memory SQLite database, which avoids the need to use a random dataset and having to explicitely specify -dialect SQLite.

Output :

```
Layer name: SELECT
OGRFeature(SELECT):0
  raw_xml (String) = <?xml version="1.0" encoding="UTF-8" ?> <reversegeocode
timestamp='Tue, 26 May 15 21:01:51 +0000' attribution='Data © OpenStreetMap
contributors, ODbL 1.0. http://www.openstreetmap.org/copyright'
querystring='format=xml&amp;lat=49.00000000&amp;lon=2.00000000'> <result
place_id="64691903" osm_type="way" osm_id="38621743" ref="Chemin du Cordon"
lat="49.0005236" lon="1.993969">Chemin du Cordon, Triel-sur-Seine, Saint-
Germain-en-Laye, Yvelines, Île-de-France, France métropolitaine, 78510,
France</result><addressparts><road>Chemin du Cordon</road><town>Triel-sur-
Seine</town><county>Saint-Germain-en-Laye</county><state>Île-de-
France</state><country>France</country><postcode>78510</postcode><country_code>f
r</country_code></addressparts></reversegeocode>
```

```
  road (String) = Chemin du Cordon
  postcode (String) = 78510
  town (String) = Triel-sur-Seine
  country (String) = France
```

By default the geocoding service used is OpenStreetMap Nominatim. Other services may be used, or specific options might be specified as extra arguments at the end of the ogr_geocode() or ogr_geocode_reverse() functions : see http://gdal.org/1.11/ogr/ogr__geocoding_8h.html#ac3e12320a8046248b992fd0ce4731903

For example to request the MapQuest Nominatim service, with answers preferably in English language (rather than in the language of the country)

```
$ ogrinfo :memory: -dialect sqlite -q \
      -sql "SELECT ogr_geocode_reverse(ST_Centroid(ogr_geocode('Paris', \
      'geometry', 'SERVICE=MAPQUEST_NOMINATIM', 'LANGUAGE=en')), 'raw', \
      'SERVICE=MAPQUEST_NOMINATIM', 'LANGUAGE=en')"
```

Output :

```
Layer name: SELECT
OGRFeature(SELECT):0
  ogr_geocode_reverse(st_centroid(ogr_geocode('Paris', 'geometry',
'SERVICE=MAPQUEST_NOMINATIM', 'LANGUAGE=en')), 'raw',
'SERVICE=MAPQUEST_NOMINATIM', 'LANGUAGE=en') (String) = <?xml version="1.0"
encoding="UTF-8" ?> <reversegeocode timestamp='Tue, 26 May 15 21:11:07 +0000'
attribution='Data © OpenStreetMap contributors, ODbL 1.0.
http://www.openstreetmap.org/copyright'
querystring='format=xml&amp;lat=48.85650560&amp;lon=2.35213340&amp;accept-
language=en'> <result place_id="151685769"
ref="75001;75002;75003;75004;75005;75006;75007;75008;75009;75010;75011;75012;750
13;75014;75015;75016;75017;75018;75019;75020" lat="48.8565056"
lon="2.3521334">75001;75002;75003;75004;75005;75006;75007;75008;75009;75010;7501
1;75012;75013;75014;75015;75016;75017;75018;75019;75020, Place de l'Hôtel de
Ville - Esplanade de la Libération, Beaubourg, St-Merri, 4th Arrondissement,
Paris, Ile-de-France, Metropolitan France,
75001;75002;75003;75004;75005;75006;75007;75008;75009;75010;75011;75012;75013;75
014;75015;75016;75017;75018;75019;75020,
France</result><addressparts><postcode>75001;75002;75003;75004;75005;75006;75007
;75008;75009;75010;75011;75012;75013;75014;75015;75016;75017;75018;75019;75020</
postcode><pedestrian>Place de l'Hôtel de Ville - Esplanade de la
Libération</pedestrian><neighbourhood>Beaubourg</neighbourhood><suburb>St-
Merri</suburb><city_district>4th
Arrondissement</city_district><city>Paris</city><county>Paris</county><state>Ile
-de-
France</state><country>France</country><country_code>fr</country_code></addressp
arts></reversegeocode>
```

Documentation of the geocoding API can be found at http://gdal.org/1.11/ogr/ogr_sql_sqlite.html

3.5 Virtual format

Similarly to GDAL Virtual format, there is a OGR Virtual format as well. Its documentation can be found at http://gdal.org/1.11/ogr/drv_vrt.html. A XML Schema of the format can be found at http://svn.osgeo.org/gdal/branches/1.11/gdal/data/ogrvrt.xsd

A convenient way of learning the OGR VRT syntax is to use the ogr2vrt.py utility : http://svn.osgeo.org/gdal/branches/1.11/gdal/swig/python/samples/ogr2vrt.py

```
$ python ogr2vrt.py ne_10m_admin_0_countries.shp countries.vrt
```

Let's examine the content of the generated countries.vrt :

```
<OGRVRTDataSource>
 <OGRVRTLayer name="ne_10m_admin_0_countries">
 <SrcDataSource relativeToVRT="0"
shared="1">ne_10m_admin_0_countries.shp</SrcDataSource>
 <SrcLayer>ne_10m_admin_0_countries</SrcLayer>
 <GeometryType>wkbPolygon</GeometryType>
 <LayerSRS>GEOGCS[&quot;GCS_WGS_1984&quot;,DATUM[&quot;WGS_1984&quot;,SPHEROID[&
quot;WGS_84&quot;,6378137.0,298.257223563]],PRIMEM[&quot;Greenwich&quot;,0.0],UN
IT[&quot;Degree&quot;,0.0174532925199433]]</LayerSRS>
 <Field name="scalerank" type="Integer" src="scalerank" width="4"/>
 <Field name="featurecla" type="String" src="featurecla" width="30"/>
 <Field name="labelrank" type="Real" src="labelrank" width="16"
precision="6"/>
 <Field name="sovereignt" type="String" src="sovereignt" width="254"/>
 <Field name="sov_a3" type="String" src="sov_a3" width="254"/>
 <Field name="adm0_dif" type="Real" src="adm0_dif" width="16" precision="6"/>
 <Field name="level" type="Real" src="level" width="16" precision="6"/>
 <Field name="type" type="String" src="type" width="254"/>
 <Field name="admin" type="String" src="admin" width="254"/>
 […]
 <Field name="long_len" type="Real" src="long_len" width="16" precision="6"/>
 <Field name="abbrev_len" type="Real" src="abbrev_len" width="16"
precision="6"/>
 <Field name="tiny" type="Real" src="tiny" width="16" precision="6"/>
 <Field name="homepart" type="Real" src="homepart" width="16" precision="6"/>
 </OGRVRTLayer>
</OGRVRTDataSource>
```

We can find all the elements that defines a data source and its layers. It is possible to override each elements, allowing layer and field renames, change of types, width, precision etc.., removal of non desirable fields.

The above mentioned documentation page mentions how to create a geometry column of points from a CSV with a longitude and latitude column.

It is also possible to build minimalistic files, that are just wrapper over the filename of the datasource. This can be convenient when using a /vsicurl/ or /vsizip/ dataset in software that only accept « real » files :

```
<OGRVRTDataSource>
 <OGRVRTLayer name="ne_10m_admin_0_countries">
 <SrcDataSource
 relativeToVRT="0">ne_10m_admin_0_countries.shp</SrcDataSource>
 </OGRVRTLayer>
</OGRVRTDataSource>
```

On-the-fly reprojection can be achieved. Create the following file warped.vrt :

```
<OGRVRTDataSource>
  <OGRVRTWarpedLayer>
```

```
    <OGRVRTLayer name="ne_10m_admin_0_countries">
     <SrcDataSource
 relativeToVRT="0">ne_10m_admin_0_countries.shp</SrcDataSource>
    </OGRVRTLayer>
    <TargetSRS>+proj=moll</TargetSRS>
  </OGRVRTWarpedLayer>
</OGRVRTDataSource>
```

Open it in QGIS.

OGR VRT has also the capability of unioning files. This might be very convenient when wanting to convert vector datasets available as « tiles » into a format that does not accept appending, without having to use an intermediate file that merges all the tiles.

For the purpose of this workshop, let's start by creating a few tile datasets :

```
$ ogr2ogr europe.shp ne_10m_admin_0_countries.shp -where "continent = 'Europe'"
$ ogr2ogr africa.shp ne_10m_admin_0_countries.shp -where "continent = 'Africa'"
```

And for the fun, let's make rest of the world a virtual dataset (and with only a subset of all attributes) ! Create rest_of_the_world.vrt with the following content :

```
<OGRVRTDataSource>
 <OGRVRTLayer name="rest_of_the_world">
 <SrcDataSource
 relativeToVRT="0">ne_10m_admin_0_countries.shp</SrcDataSource>
 <SrcSQL>SELECT admin, continent, 'coming_from_rest_of_the_world' extra_arg
 FROM ne_10m_admin_0_countries
 WHERE continent NOT IN ('Europe', 'Africa')</SrcSQL>
 </OGRVRTLayer>
</OGRVRTDataSource>
```

This demonstrates the capability of using a SQL result layer. The SQLite dialect can be used by adding the dialect='SQLite' attribute in the <SrcSQL> element.

And finally create world.vrt :

```
<OGRVRTDataSource>
  <OGRVRTUnionLayer name="world">
    <OGRVRTLayer name="europe">
      <SrcDataSource>europe.shp</SrcDataSource>
    </OGRVRTLayer>
    <OGRVRTLayer name="africa">
      <SrcDataSource>africa.shp</SrcDataSource>
    </OGRVRTLayer>
    <OGRVRTLayer name="rest_of_the_world">
      <SrcDataSource>rest_of_the_world.vrt</SrcDataSource>
    </OGRVRTLayer>
  </OGRVRTUnionLayer>
</OGRVRTDataSource>
```

Open world.vrt with QGIS. You can check that countries not in Europe and Africa have only their admin and continent fields set, as well as the string 'coming_from_rest_of_the_world' as an extra field. On the contrary Europe and Africa countries will have that latter attribute null. This demonstrates that some heterogenity in data sources can be possible.

3.6 Layer algebra

Layer algebra covers a set of spatial algorithm that takes a whole layer (the input layer) to be combined with another layer (the method layer), according to various spatial predicates, in order to produce an output layer as result set.

This is extensively described at https://trac.osgeo.org/gdal/wiki/LayerAlgebra

In Python, the practical way of using it is the following script : http://svn.osgeo.org/gdal/branches/1.11/gdal/swig/python/samples/ogr_layer_algebra.py

Let's prepare an extract of most Western Europe with some bits of North Africa, and keep only a few attributes :

```
$ ogr2ogr countries_eur.shp ne_10m_admin_0_countries.shp -clipdst -10 30 20 60 \
          -select admin,continent,pop_est
```

And an extract of provinces of France (including oversea territories), Italy and Germany :

```
$ ogr2ogr provinces.shp ne_10m_admin_1_states_provinces_shp.shp \
 -select name,type_en -where "admin in ('France', 'Italy', 'Germany')"
```

Let's begin with the « Identity » algorithm.

```
$ python ogr_layer_algebra.py Identity -input_ds provinces.shp \
         -method_ds countries_eur.shp -output_ds identity.shp
```

Open identify.shp in QGIS. You'll notice that the geometry shapes are the ones of provinces.shp (theoretically they could have been subdivided, but this is not the case here as they are refinements from countries_eur shapes), but if you look at details, you'll see that they have been extended with the fields of the matching countries (so you can now know to which contry a province belongs to). If you look at French Guyana (or other oversea territories) details, the admin, continent and pop_est values are NULL.

Let's experiment with a few other algorithms.

Union combines objects that are in both layers and where they intersect, it creates a dedicated identity with attributes of both layers.

```
$ python ogr_layer_algebra.py Union -input_ds provinces.shp \
 -method_ds countries_eur.shp -output_ds union.shp
```

Intersection only retains the part of objects that intersects, and retain attributes of both layers in those overlapping areas.

```
$ python ogr_layer_algebra.py Intersection -input_ds provinces.shp \
 -method_ds countries_eur.shp -output_ds intersection.shp
```

SymDifference will only retain part of objects that do not overlap. Or said otherwise areas for which we have the information from the input layer or the method layer, but not both. In that instance, countries shape excluding the ones of France, Germany and Italy, but with French oversea territories.

```
$ python ogr_layer_algebra.py SymDifference -input_ds provinces.shp \
 -method_ds countries_eur.shp -output_ds symdifference.shp
```

You can notice that the result is not « clean ». This is due to boundaries of both input datasets not strictly overlapping.

Let's look at the areas of the resulting polygons :

```
$ ogrinfo symdifference.shp -sql "select *, ogr_geom_area from symdifference
order by ogr_geom_area desc" -geom=no
```

You can notice a break between the last valid identy (Vatican, area ~= 1.3e-6 square degrees), and below results with are very very thin polygons of areas below 1e-8 square degrees. So we are going to clean up the result with :

```
$ ogr2ogr symdifference_cleaned.shp symdifference.shp \
          -where "ogr_geom_area > 1e-8"
```

Union, Intersection and SymDifference are symetric operations, i.e. if you switch the input and method layer, you will get the same results (the order of attributes excluded)

```
$ python ogr_layer_algebra.py Clip -input_ds provinces.shp \
 -method_ds countries_eur.shp -output_ds clip.shp -opt SKIP_FAILURES=YES
```

The -opt SKIP_FAILURES=YES is to avoid the process to abort at the first error is encountered. This can happen when odd geometries are produced as the result of the spatial intersections.

If you compare the results of Clip and Intersection, they will look similar, except that in Clip, only attributes of the input layers are kept by default. To better understand how clip works, let's reverse the input and method layers :

```
$ python ogr_layer_algebra.py Clip -input_ds countries_eur.shp\
 -method_ds provinces.shp -output_ds clip_reversed.shp
```

The result is the subset of the countries for which we have province information (and if we add only partial province coverage, the country shapes would have been clipped to that partial coverage)

The set of attributes in the resulting layer can be configured with the -input_fields and -method_fields arguments.

3.7 WFS driver

A Web Feature Service publishes geospatial objects over the Web as geometry and attributes. The OGR WFS driver is the client part that connects to a server, such as open source software MapServer, GeoServer or Deegree. It supports the WFS 1.0, 1.1 and 2.0 servers.

Documentation of the OGR WFS driver is at http://gdal.org/1.11/ogr/drv_wfs.html

First let's start by copying the world.map file that defines the layers that MapServer will expose to /tmp

```
$ cp world.map /tmp
```

Connect to the service with:

```
$ ogrinfo -ro "WFS:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map"
```

Output:

```
INFO: Open of `WFS:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map'
 using driver `WFS' successful.
1: ne_10m_admin_0_countries
2: ne_10m_admin_1_states_provinces_shp
```

Those are the 2 layers we have worked previously. You can get more information with:

```
$ ogrinfo -ro -al -so "WFS:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map"
```

We can issue attribute and/or spatial requests with the -where or -spat options.

```
$ ogrinfo -ro "wfs:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map" \
 ne_10m_admin_0_countries -where "admin = 'France'" -geom=no --debug on
```

The --debug on option enables debug traces, which shows that the filter is well transmitted to the server.

```
$ ogr2ogr out.shp "wfs:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map" \
  ne_10m_admin_0_countries -overwrite
```

Let's display it in QGIS. Ouch! This doesn't look right. This is due to a misunderstanding between the client and server related how to interpret the order of the components in a coordinate pair, whether it is longitude,latitude or latitude,longitude. MapServer as of v6.4.1 and WFS 1.1 advertizes the coordinate system in a way that confuses OGR. There are 2 ways of fixing this.

Explicitly request use of the WFS 1.0 protocol that always use longitude,latitude order:

```
$ ogr2ogr out.shp -overwrite \
"wfs:http://localhost/cgi-bin/mapserv?VERSION=1.0.0&MAP=/tmp/world.map" \
 ne_10m_admin_0_countries
```

or tune the OGR WFS driver to tell him that EPSG:XXXX strings returned by the server must be interpreted as urn:ogc:def:crs:EPSG::XXXX with the GML_CONSIDER_EPSG_AS_URN configuration option (documented at http://gdal.org/1.11/ogr/drv_gml.html).

```
$ ogr2ogr out.shp -overwrite --config GML_CONSIDER_EPSG_AS_URN YES \
"wfs:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map" \
ne_10m_admin_0_countries
```

Note: this issue will no longer be encountered with MapServer 7 in WFS 2.0.

Now let's convert the ne_10m_admin_1_states_provinces_shp layer:

```
$ ogr2ogr out2.shp -overwrite --config GML_CONSIDER_EPSG_AS_URN YES \
"wfs:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map" \
ne_10m_admin_1_states_provinces_shp
```

And display out2.shp in QGIS. Hum, we only have a few countries. The reason is that the server has been configured with a limit in the number of features returned in a single request (in that example, 300), in order to avoid abuse by clients. The WFS 2.0 protocol has introduced a mechanism to scroll in the result, and a few servers such as MapServer or GeoServer have retrofitted this capability as an extension. We can enable this mode in the client by setting the OGR_WFS_PAGING_ALLOWED configuration option.

```
$ ogr2ogr out2.shp -overwrite --config GML_CONSIDER_EPSG_AS_URN YES \
"wfs:http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map" \
ne_10m_admin_1_states_provinces_shp --config OGR_WFS_PAGING_ALLOWED YES
```

We now have all our features !

Note : in GDAL 2.0, with WFS 2.0 servers that expose paging capabilities, explicitely setting OGR_WFS_PAGING_ALLOWED is no longer necessary.

For ease of use, it is possible to create a service description file. For example let's create test_wfs.xml with the following content :

```
<OGRWFSDataSource>
  <URL>http://localhost/cgi-bin/mapserv?MAP=/tmp/world.map&amp;VERSION=1.0.0</URL>
 <PagingAllowed>ON</PagingAllowed>
 <PageSize>300</PageSize>
</OGRWFSDataSource>
```

Run :

```
$ ogrinfo -ro test_wfs.xml
```

If you refresh test_wfs.xml, you'll see it has been updated with the capabilities of the server, as well as the layer definitions. This saves some time when operating against remote servers to avoid getting those basic information over and over.