python_data_workshop

June 14, 2020

1 Overview

This notebook will provide a quick start guide to downloading, visualizing, and analyzing remote sensing data commonly used in the cryosphere community. We will focus on using Python to programmatically transform and visualize the data in order to derive products that either provide physical insight directly or are suitable for inputs to other computational tools. For example, we will explore and build tools to efficiently extract regions of interest from large datasets, warp raster images between different projection systems, and apply image transformation algorithms to highlight certain features in the data. Here, we will use Landsat 8 optical imagery and Sentinel-1 synthetic aperture radar (SAR) imagery to demonstrate the advantages and challenges of each data type.

1.1 What we will not cover:

This notebook is not intended to be an introduction to Python or core numerical packages like `numpy`, `scipy`, `matplotlib`, etc. There exist many excellent online resources that cover these packages in great detail:

1. `numpy`: Quickstart tutorial
2. `scipy`: User guide for scipy.ndimage
3. `matplotlib`: Tutorial for pyplot

We will use several of the capabilities of those packages in this notebook, but hopefully the context of our exercises will make the use of those functions easy to understand. Prior experience with a language like MATLAB can help your understanding of the material presented here.

Additionally, we will not be discussing any of the powerful interactive GIS platforms like QGIS (https://www.qgis.org/en/site/), which is an excellent tool for raster data exploration, layering multiple data sources, and visualization. However, those platforms can often have steep learning curves since all of their functionality is immediately visible to users. Instead, we will try to build an intuition on the content and structure of satellite imagery by working directly with raster and geographic metadata programmatically.

2 Background information

The primary data structure we will be covering here is the geospatial raster. A raster is essentially a grid/array of values quantifying some quantity of interest (example image source: National Ecological Observatory Network):

These quantities may be direct measurements of surface properties (e.g., terrain height), derived quantities (e.g., surface strain), or class labels (e.g., vegetated or bare earth). The three key attributes of raster data are:

1. **Extent**: the spatial extent that the raster covers (geographical bounding box).
2. **Resolution**: the spatial size of each grid element in the raster.
3. **Spatial reference system**: the spatial reference (coordinate system) that the above two attributes are defined in.

You are likely familiar with geographical coordinates provided in latitute and longitude. What you may not know are that these coordinates are tied to a specific projection defined by the WGS84 oblate spheroid model. Another reference system you will likely have come across is the Mercator projection, which is a cylindrical projection that allows for lines of constant heading to be represented as a straight line in a map. This projection distorts Greenland and Antarctica such that they appear much larger than they actually are. For many glaciological studies, we use what's called the polar stereographic coordinate system which allows for minimal distortion in the polar regions. There are many different ways to represent a spatial reference system, and the one we will use in this notebook is the EPSG code. For example, EPSG:4326 is the common WGS84 latitude/longitude, EPSG:3413 is polar stereographic for the Northern hemisphere, and EPSG:3031 is polar stereographic for the Southern hemisphere.

Considering the above attributes, a geospatial raster file will have roughly the following structure:

Depending on the raster file format, the header metadata may be in the same file or in a separate ASCII file. To simplify handling of raster data of different file formats, we will use a Python package called `icutils` (GitHub link), which relies on the Geospatial Data Abstraction Library (GDAL) to do the heavy lifting. `icutils` will allow us to read in raster data, transform the data between different reference systems, and visualize the data in a programmatic fashion using Python.

3 Landsat 8 Image Analysis

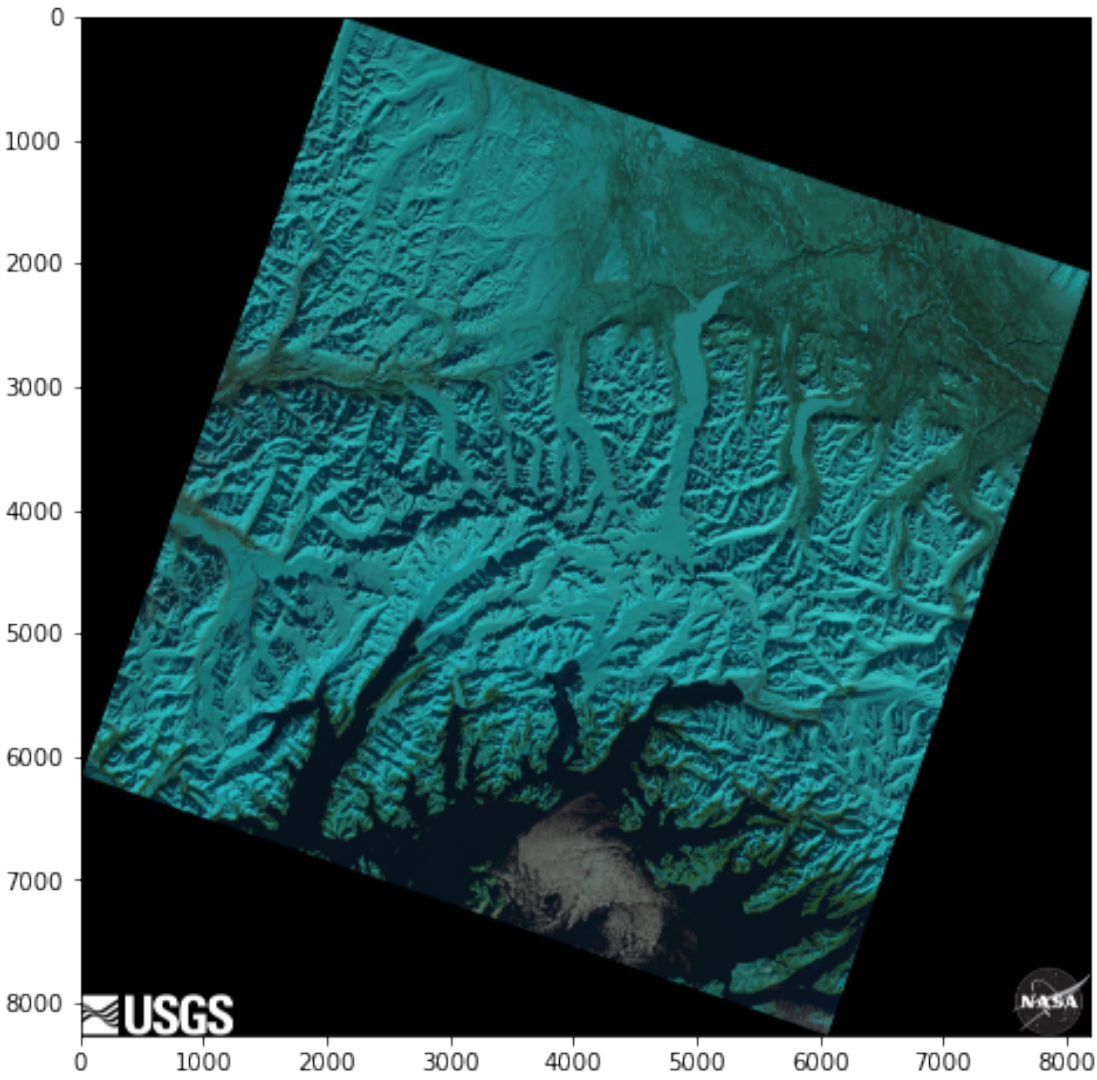
The Landsat program is a joint NASA/USGS mission that has been acquiring satellite imagery of the Earth's surface for five decades. Landsat 8 is the most recent satellite in the program and was launched on February 11, 2013. The satellite has a two-sensor payload consisting of the Operational Land Imager (OLI) and the Thermal InfraRed Sensor (TIRS). These two sensors combine to provide imagery for 11 different spectral bands ranging from visible light to long-wavelength infrared. The imagery itself is acquired at a 30 meter spatial resolution with a 16-day repeat time and is delivered in "tiles" with size roughly 185 km x 180 km. For the first example, we will look at a Landsat 8 tile over Columbia Glacier in Alaska.

[1]:
```
# Let's first import all necessary packages

# Numpy and matplotlib for array analysis and visualization
import numpy as np
import matplotlib.pyplot as plt

# scipy.ndimage and opencv for image processing
import scipy.ndimage as ndimage
import cv2 as cv
```

```
# iceutils for raster support
import iceutils as ice

# Other python defaults
import os
```

Let's first look at an image of our study area which can be downloaded USGS EarthExplorer. This image is a basic JPEG that we're all likely familiar with.

[2]:
```
# Load image
image = plt.imread('LC08_L1TP_067017_20190301_20190309_01_T1.png')

# View it
fig, ax = plt.subplots(figsize=(10,8))
im = ax.imshow(image)
```

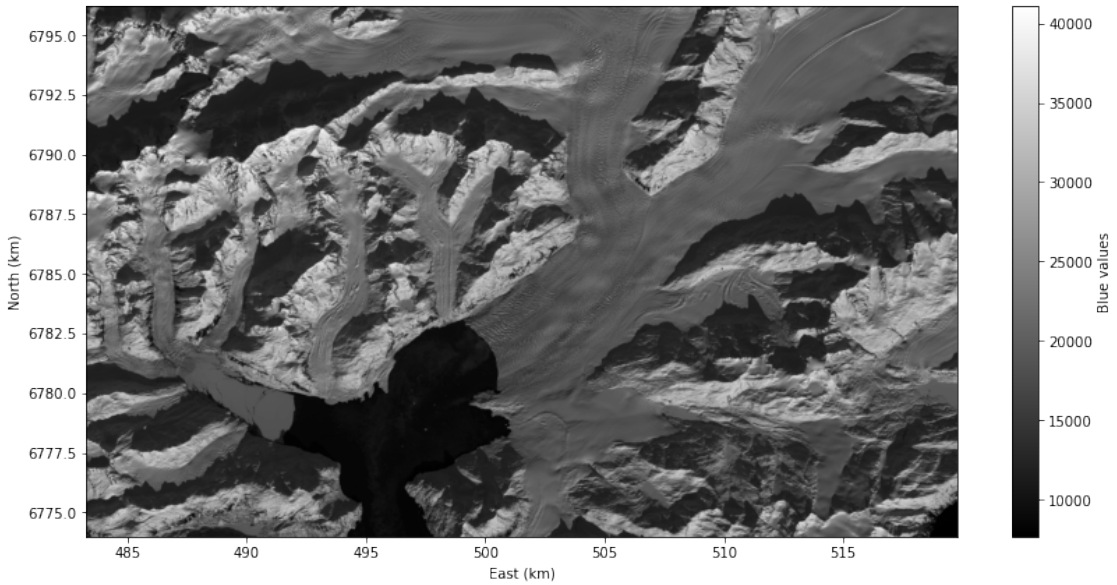


The above image is described as a "true color" image but has an unnatural blue hue to it. Additionally, the JPEG has no data about *where* the image is located on Earth. Therefore, let's work directly with individual Landsat-8 spectral bands formatted as GeoTIFF files.

[3]:
```
# The filenames for the different bands
blue_file = 'LC08_L1TP_067017_20190301_20190309_01_T1_B2.TIF'
green_file = 'LC08_L1TP_067017_20190301_20190309_01_T1_B3.TIF'
red_file = 'LC08_L1TP_067017_20190301_20190309_01_T1_B4.TIF'

# Load only RasterInfo (metadata) for one of the bands
hdr = ice.RasterInfo(blue_file)
# Print out the geographic bounds for this band
# Since the projection system of these rasters is UTM 6N, the
# coordinates printed to the screen will be in meters
print('Geographic extent [X_min, X_max, Y_min, Y_max]:', hdr.extent)

# The latitude-longitude coordinates of area of interest
lat_min = 61.10
lat_max = 61.30
lon_min = -147.31
lon_max = -146.63

# Convert to UTM 6N coordinates
east_min, north_min = ice.transform_coordinates(lon_min, lat_min, 4326, hdr.
 ↪epsg)
east_max, north_max = ice.transform_coordinates(lon_max, lat_max, 4326, hdr.
 ↪epsg)

# Create a UTM bounding box ("projection window" in GDAL terminology)
# projWin = [upper_left_X, upper_left_Y, lower_right_x, lower_right_y]
projWin = [east_min, north_max, east_max, north_min]

# Read raster data for area of interest for blue band
blue_raster = ice.Raster(blue_file, projWin=projWin)

# Get the coordinates for our area (convert to km)
extent = 1.0e-3 * blue_raster.hdr.extent

fig, ax = plt.subplots(figsize=(14,10))
im = ax.imshow(blue_raster.data, cmap='gray', extent=extent)
cbar = plt.colorbar(im, ax=ax, shrink=0.7)
cbar.set_label('Blue values')
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

```
Geographic extent [X_min, X_max, Y_min, Y_max]: [ 379185.  624885. 6691515.
6939315.]
```

[3]:
```
Text(0, 0.5, 'North (km)')
```

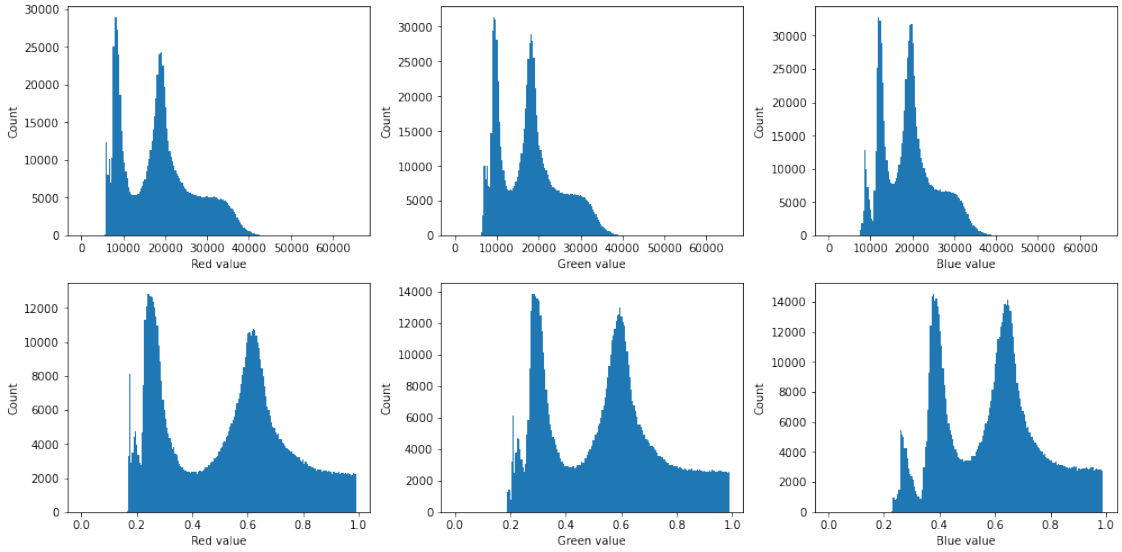


We have now seen how to use two of the primary classes in `iceutils`: `Raster` and `RasterInfo`. The `Raster` class is the main class that handles reading in raster data and contains an instance of `RasterInfo`. The raster data itself are stored in a NumPy array and can be accessed via the `data` attribute, e.g. `blue_raster.data`. The `RasterInfo` class is a lightweight class that encapsulates all the metadata regarding spatial extent, resolution, and reference systems. This information can be accessed via the `hdr` attribute, e.g. `blue_raster.hdr`. A `RasterInfo` object can also be created separately without reading in the full raster data, which we did above with the local `hdr` instance to retieve the EPSG code for the Landsat image.

Note that in the above image, we are only looking at surface reflectance values for blue visible light (0.45 micrometer wavelength). When viewing the image, we can use any colormap we like (see this link for more details), and we happen to be using a gray colormap. Colormaps create a mapping from an image value to a three-element RGB (Red-Blue-Green) code. Thus, the above image is showing the blue reflectance values, but we are viewing it with a gray colormap. In order to render an image that is as close as possible to "true color", e.g. what we would be able to see with our own eyes, we must manually create a 3D RGB array using the blue, green, and red reflectance bands. But first, let's look at how we can tweak the histograms of the three bands for better visual clarity.

[4]:
```
# Load the other bands
green_raster = ice.Raster(green_file, projWin=projWin)
red_raster = ice.Raster(red_file, projWin=projWin)

# Define utility function for scaling data to the range [0, 1]
def make_uniform_data(data, min_value=0, max_value=65535):
```

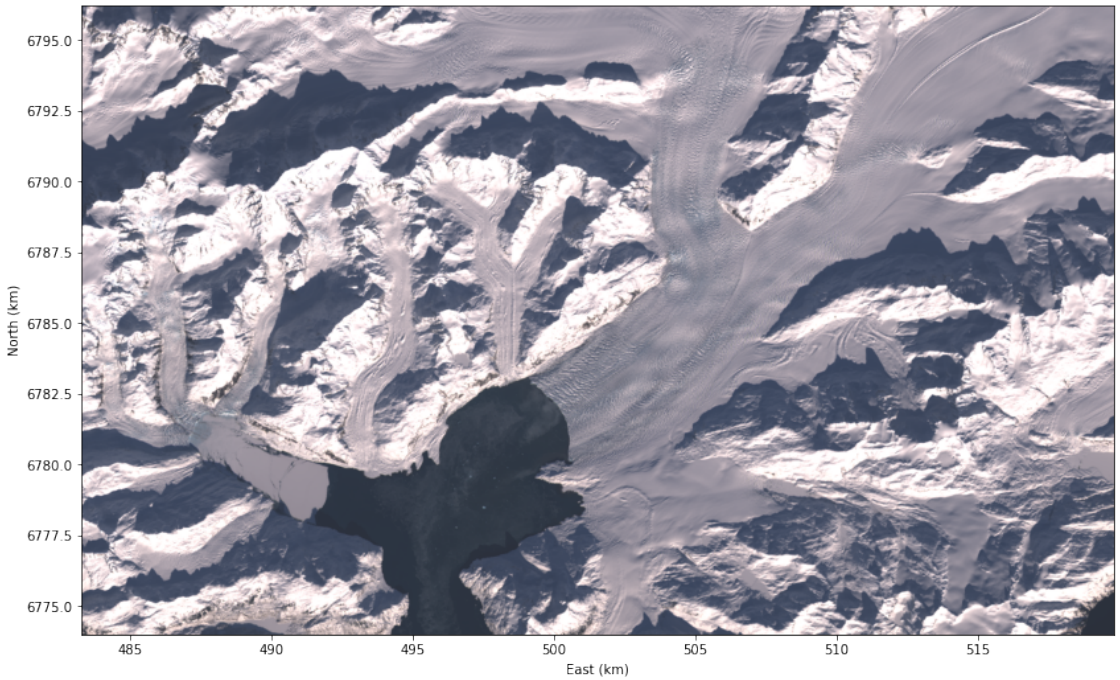
```
    # Scale the data (convert to float32 before scaling)
    data_norm = (data.astype(np.float32) - min_value) / (max_value - min_value)
    # Clip values to be between [0, 1]
    data_clipped = np.clip(data_norm, 0, 1)
    return data_clipped

# Apply function to all bands
blue_norm = make_uniform_data(blue_raster.data, min_value=1000, max_value=30000)
green_norm = make_uniform_data(green_raster.data, min_value=1000,␣
 ↪max_value=30000)
red_norm = make_uniform_data(red_raster.data, min_value=1000, max_value=30000)

# View histograms for all bands
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(14,7))
_ = axes[0,0].hist(red_raster.data.ravel(), 200, range=[0,65535])
_ = axes[0,1].hist(green_raster.data.ravel(), 200, range=[0,65535])
_ = axes[0,2].hist(blue_raster.data.ravel(), 200, range=[0,65535])
_ = axes[1,0].hist(red_norm.ravel(), 200, range=[0, 0.99])      # Make upper␣
 ↪bound 0.99 to avoid clipped values
_ = axes[1,1].hist(green_norm.ravel(), 200, range=[0, 0.99])
_ = axes[1,2].hist(blue_norm.ravel(), 200, range=[0, 0.99])
for i in range(2):
    axes[i,0].set_xlabel('Red value')
    axes[i,1].set_xlabel('Green value')
    axes[i,2].set_xlabel('Blue value')
    for j in range(3):
        axes[i,j].set_ylabel('Count')
fig.set_tight_layout(True)
```



We have now scaled the values to lie in the range [0, 1] and to span as much of the [0, 1] space as possible. You can think of this as a type of photo editing where we are increasing the "exposure" and bringing up the "shadows". Let's now form a 3D RGB array and view it with matplotlib.

[5]:
```
# Stack the scaled bands along 3rd axis to create "true color" RGB
rgb = np.dstack((red_norm, green_norm, blue_norm))

# View it
fig, ax = plt.subplots(figsize=(14,10))
im = ax.imshow(rgb, extent=extent)
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

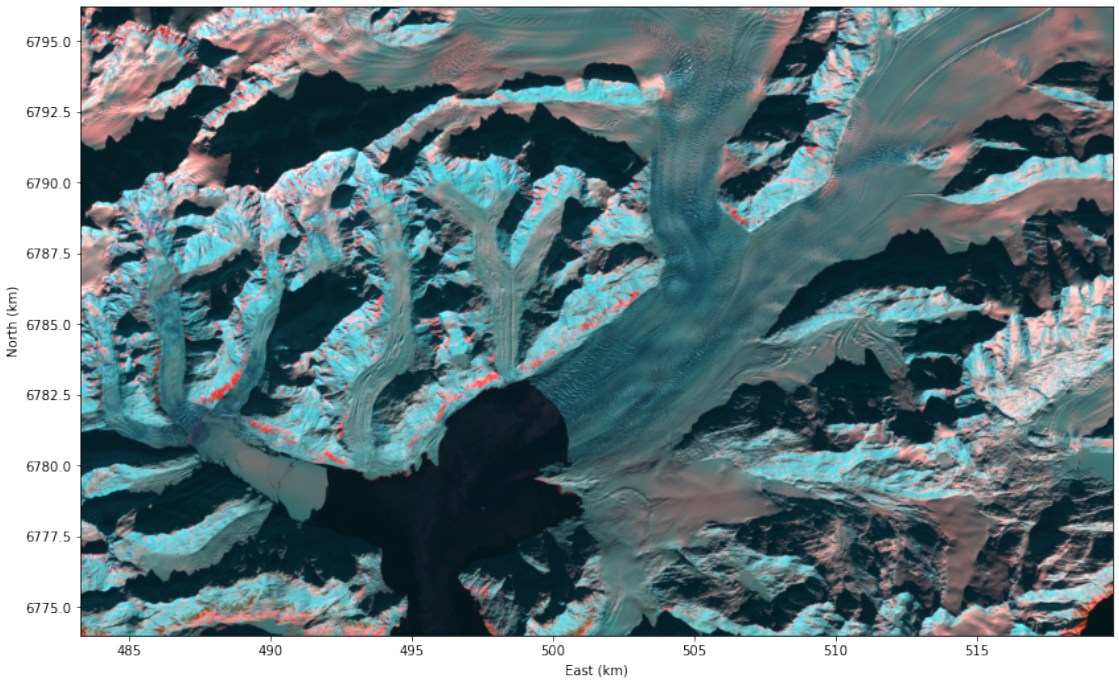
[5]:
```
Text(0, 0.5, 'North (km)')
```



You have likely seen false color satellite images that do not represent what the eye can see but are meant to highlight certain variations in surface properties. For example, since Landsat-8 records infrared wavelengths, it may be possible to discern areas with different moisture content. Water absorbs more energy in near infrared (NIR) and shortwave infrared (SWIR) while non-water reflects more energy. Thus, let's create another RGB image, but let's combine bands 4, 5, and 6. Band 5 is NIR and band 6 is SWIR. By making band 6 correspond to the "red" band in an RGB array, any areas with a high red reflectance likely indicates frozen ice. Alternatively, areas that are dark blue may contain significant meltwater.

[6]:
```
# Load bands 4, 5, and 6 with the same projWin as before
b4 = ice.Raster('LC08_L1TP_067017_20190301_20190309_01_T1_B4.TIF',␣
 ↪projWin=projWin)
b5 = ice.Raster('LC08_L1TP_067017_20190301_20190309_01_T1_B5.TIF',␣
 ↪projWin=projWin)
b6 = ice.Raster('LC08_L1TP_067017_20190301_20190309_01_T1_B6.TIF',␣
 ↪projWin=projWin)

# Apply normalization function to all bands
b4_norm = make_uniform_data(b4.data, min_value=2000, max_value=40000)
b5_norm = make_uniform_data(b5.data, min_value=2000, max_value=40000)
b6_norm = make_uniform_data(b6.data, min_value=5000, max_value=7000)

# Stack the scaled bands along 3rd axis to create false color RGB
rgb = np.dstack((b6_norm, b5_norm, b4_norm))

# View it
fig, ax = plt.subplots(figsize=(14,10))
im = ax.imshow(rgb, extent=extent)
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

[6]:
```
Text(0, 0.5, 'North (km)')
```

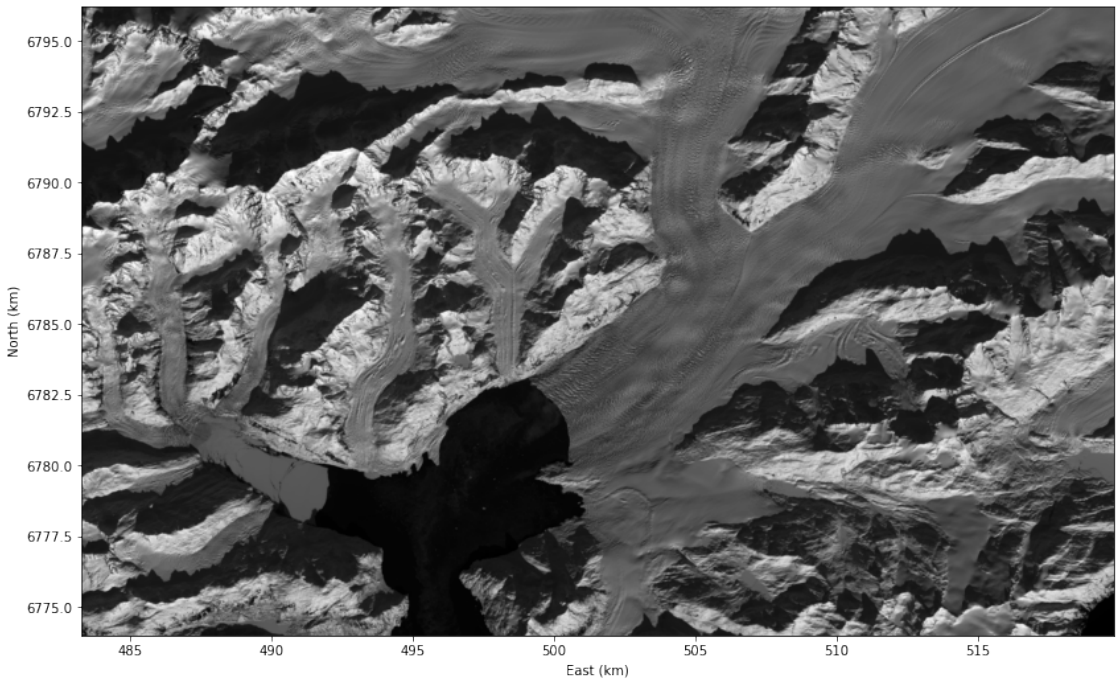


While color images are great for visualization and analysis of surface properties, for a lot of the

quantitative analysis we do, we often only need to work with a single band. Let's create a synthetic grayscale image by taking the norm of the blue, green, and red bands.

[7]:
```
# Create grayscale (make sure to convert integer arrays to float32)
# Use original, unclipped raster data (preserve as much information as possible)
gray = np.sqrt(blue_raster.data.astype(np.float32)**2 +
               green_raster.data.astype(np.float32)**2 +
               red_raster.data.astype(np.float32)**2)

# Scale values to [0, 1]
gray = make_uniform_data(gray, 10000, 70000)

fig, ax = plt.subplots(figsize=(14,10))
im = ax.imshow(gray, cmap='gray', extent=extent)
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

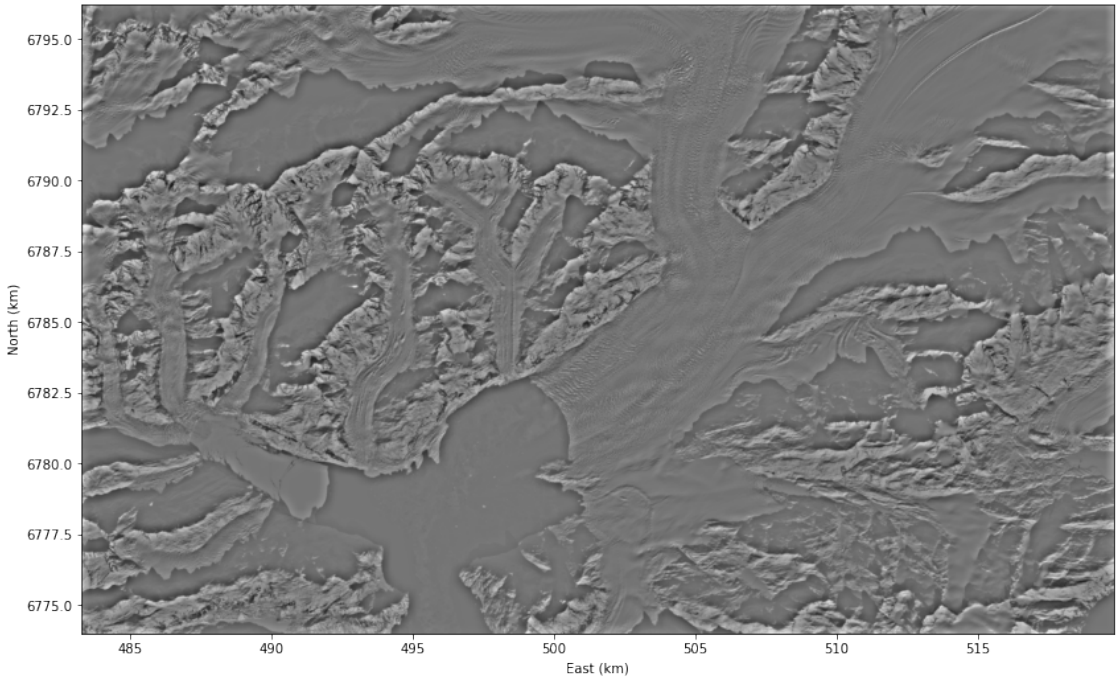
[7]:
```
Text(0, 0.5, 'North (km)')
```



For many types of analyses, we'll often want to perform some type of image processing in order to isolate certain features or details in images. For example, high-pass filtering remove long-wavelength features in images and retains short-wavelength (high frequency) features. This can be useful for operations like feature tracking. Let's look at how to do high-pass filtering using `opencv`.

[8]:
```
# Build up 2D filter for high-pass filtering
WallisFilterWidth = 21
kernel = -1.0 * np.ones((WallisFilterWidth, WallisFilterWidth), dtype=np.
 ↪float32)
kernel[(WallisFilterWidth - 1) // 2, (WallisFilterWidth - 1) // 2] = kernel.
 ↪size - 1
kernel = kernel / kernel.size

# Perform filtering
gray_high_pass = cv.filter2D(gray, -1, kernel, borderType=cv.BORDER_CONSTANT)

fig, ax = plt.subplots(figsize=(14,10))
im = ax.imshow(gray_high_pass, cmap='gray', extent=extent)
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

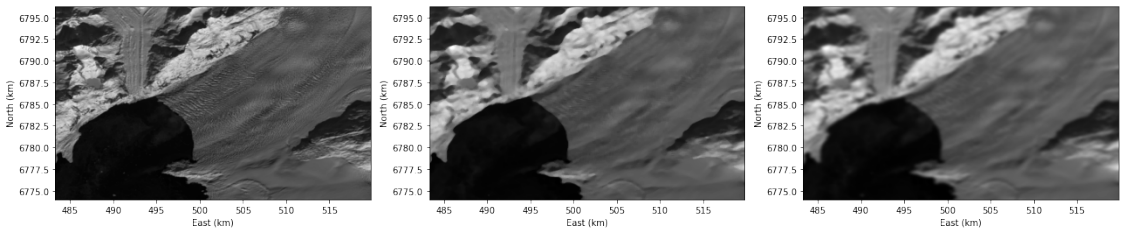
[8]:
```
Text(0, 0.5, 'North (km)')
```



On the other hand, for certain types of analyses, we may want to remove high frequency details to obtain a "smoother" image. This type of smoothing can be useful for pre-processing ice velocity maps in order to lower the effective resolution to a scale that is commensurate with ice physics. Let's look at two different types of smoothing: i) median filter; and ii) Gaussian filter. Both techniques use a sliding window of some specified size and perform a math operation on the pixels within that sliding window. For the median filter, the median of the pixels within the window is computed. For the Gaussian filter, a 2D Gaussian template is convolved with the pixels within the

window.

[9]:
```
# Median filtering
gray_median_filtered = ndimage.median_filter(gray, size=5)

# Gaussian filtering (smoothing)
gray_gaussian_filtered = ndimage.gaussian_filter(gray, sigma=2.0)

# View sub-region for original and two filering approaches
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(18,10))
im1 = ax1.imshow(gray[300:600, 400:800], cmap='gray', extent=extent)
im2 = ax2.imshow(gray_median_filtered[300:600, 400:800], cmap='gray',␣
 ↪extent=extent)
im3 = ax3.imshow(gray_gaussian_filtered[300:600, 400:800], cmap='gray',␣
 ↪extent=extent)
for ax in (ax1, ax2, ax3):
    ax.set_xlabel('East (km)')
    ax.set_ylabel('North (km)')
fig.set_tight_layout(True)
```



Notice that even though the smoothed images have roughly the same resolution, there are differences in image quality, particularly in the mountain areas. Median filtering is known to be edge-preserving, so proper choice of filteirng operation will depend on your particular application.

4 Sentinel-1A SAR Imagery

Let's now examine some synthetic aperture radar (SAR) imagery. SAR is a fundamentally different imaging system than optical systems like Landsat. Instead of "passively" measuring reflected sunlight at different spectral bands, SAR systems "actively" send radar pulses at wavelengths much longer than those recorded by optical systems. The radar pulses will scatter off the surface where scattering is influenced by features such as surface roughness, moisture content, dielectric properties, etc. The SAR receiver then records the returning pulses. Since SAR systems emit radiation at longer wavelengths, they can image through clouds and at any time of day.

To get a feel for SAR imagery, we will look at a Sentinel-1A Level 1.5 image over Upernavik Isstrom in West Greenland. For these products, full resolution radar images have been multilooked (averaged down) to a resolution of 40 m x 40 m (for extensive information about Sentinel-1, see this link). A preview of this scene is below:

One important difference between this radar image and the optical images we viewed before is that the former is defined in radar coordinates, which means the first line in the image corresponds to the first reflected radar pulse recorded by the satellite. Thus, for a satellite flying from South to North (as is the case for the above image), the radar image will be vertically flipped compared to an optical image. All this means that for radar images, we need to perform some extra steps in order to get the images into a projection that's useful for our analysis.

The SAR image shown above is actually quite large, and it's not a good idea to load it into memory and view it all at once. If we run `gdalinfo` on the image:

```
$ gdalinfo s1a-iw-grd-hh-20150103t204800-20150103t204823-004012-004d51-001.tiff

Driver: GTiff/GeoTIFF
Files: s1a-iw-grd-hh-20150103t204800-20150103t204823-004012-004d51-001.tiff
Size is 25274, 15263
ERROR 1: PROJ: proj_create_from_database: ellipsoid not found
ERROR 1: PROJ: proj_create_from_database: ellipsoid not found
GCP Projection =
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["unnamed",6378137,298.25722356049,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4326]]
Data axis to CRS axis mapping: 2,1
GCP[  0]: Id=1, Info=
          (0,0) -> (-56.8448916221808,72.3191765191589,192.296114596538)
GCP[  1]: Id=2, Info=
          (1264,0) -> (-56.4894792301104,72.3535844968259,192.295865926892)
GCP[  2]: Id=3, Info=
          (2528,0) -> (-56.132722208647,72.3873542656289,192.295623748563
...
Metadata:
  AREA_OR_POINT=Area
  TIFFTAG_DATETIME=2015:01:03 23:50:32
  TIFFTAG_IMAGEDESCRIPTION=Sentinel-1A IW GRD HR L1
  TIFFTAG_SOFTWARE=Sentinel-1 IPF 002.36
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (    0.0,    0.0)
```

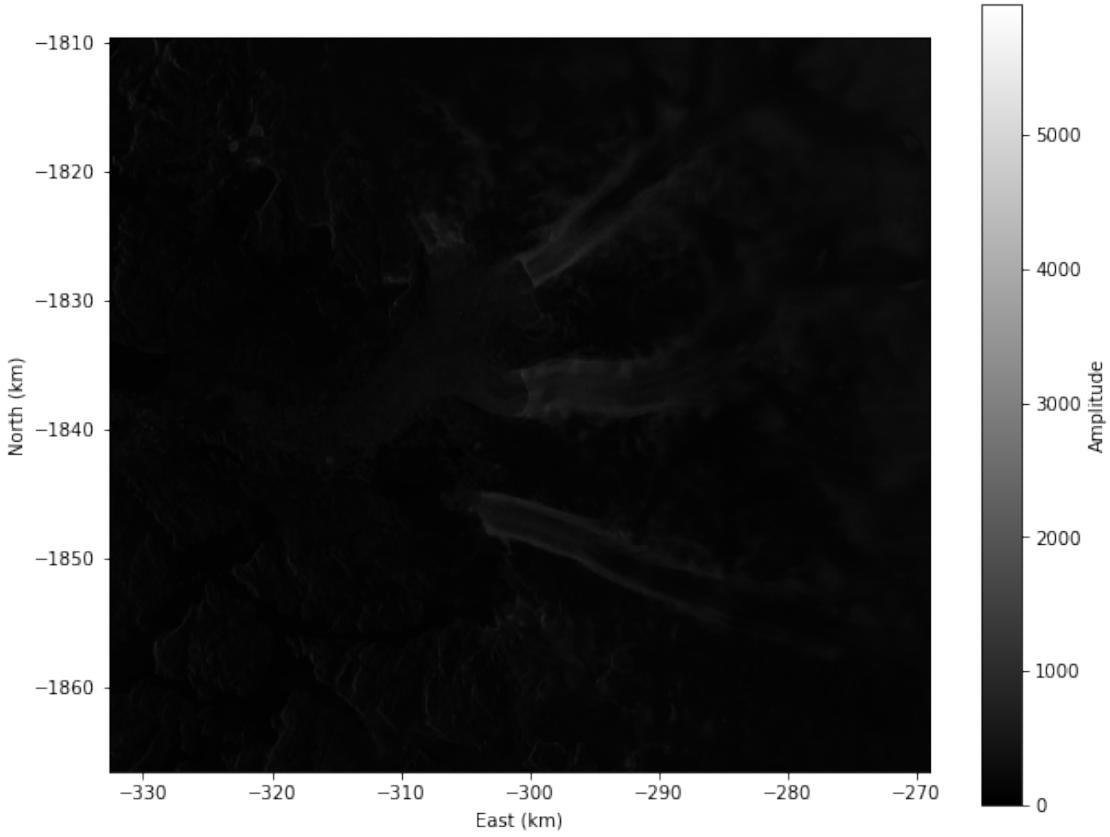
```
Lower Left  (    0.0,15263.0)
Upper Right (25274.0,    0.0)
Lower Right (25274.0,15263.0)
Center      (12637.0, 7631.5)
Band 1 Block=25274x1 Type=UInt16, ColorInterp=Gray
```

we can see that the coordinates of the image are not geographical. Instead, ground control points (GCPs) are provided in the header, which provide the mapping between radar coordinates to geographical coordinates for a few distinct points in the scene. We will use these GCPs to extract geographic areas of interest from the radar image and interpolate it to a geographical grid.

[10]:
```
# Load metadata (in RasterInfo object) for full Sentinel image
ref_hdr = ice.
 ↪RasterInfo('s1a-iw-grd-hh-20150103t204800-20150103t204823-004012-004d51-001.
 ↪tiff')

# Read GCP information (specify conversion of GCP coordinates to EPSG:3413)
# Internally, this creates 2D spline interpolators for GCPs
ref_hdr.read_GCPs(gcp_epsg=4326, epsg_out=3413)

# Define our area of interest in latitude and longitude
lon_min, lon_max = -55.4151, -53.1988
lat_min, lat_max = 72.7187, 73.1329

# Convert to polar stereographic
x_min, y_max = ice.transform_coordinates(lon_min, lat_max, 4326, 3413)
x_max, y_min = ice.transform_coordinates(lon_max, lat_min, 4326, 3413)

# Determine correpsonding radar image coordinates for our area of interest
cols = []; rows = []
for x, y in ((x_min, y_min), (x_max, y_min), (x_max, y_max), (x_min, y_max)):
    row, col = ref_hdr.xy_to_imagecoord_gcp(x, y)
    rows.append(row)
    cols.append(col)

# Create slice objects for the radar image subset
islice = slice(int(np.min(rows)), int(np.max(rows)))
jslice = slice(int(np.min(cols)), int(np.max(cols)))

# Read radar subset
raster = ice.
 ↪Raster('s1a-iw-grd-hh-20150103t204800-20150103t204823-004012-004d51-001.
 ↪tiff',
                    islice=islice, jslice=jslice)

# Define output grid in polar stereographic (25 m spacing)
x_array = np.arange(x_min, x_max, 25.0)
y_array = np.arange(y_max, y_min, -25.0)
```

```
X, Y = np.meshgrid(x_array, y_array)

# Warp the radar image
raster_polar = ice.warp_with_gcp_splines(raster, ref_hdr, x=X, y=Y)

# View warped result
extent = 1.0e-3 * raster_polar.hdr.extent
fig, ax = plt.subplots(figsize=(10,8))
im = ax.imshow(raster_polar.data, cmap='gray', extent=extent)
cbar = plt.colorbar(im, ax=ax, orientation='vertical')
cbar.set_label('Amplitude')
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

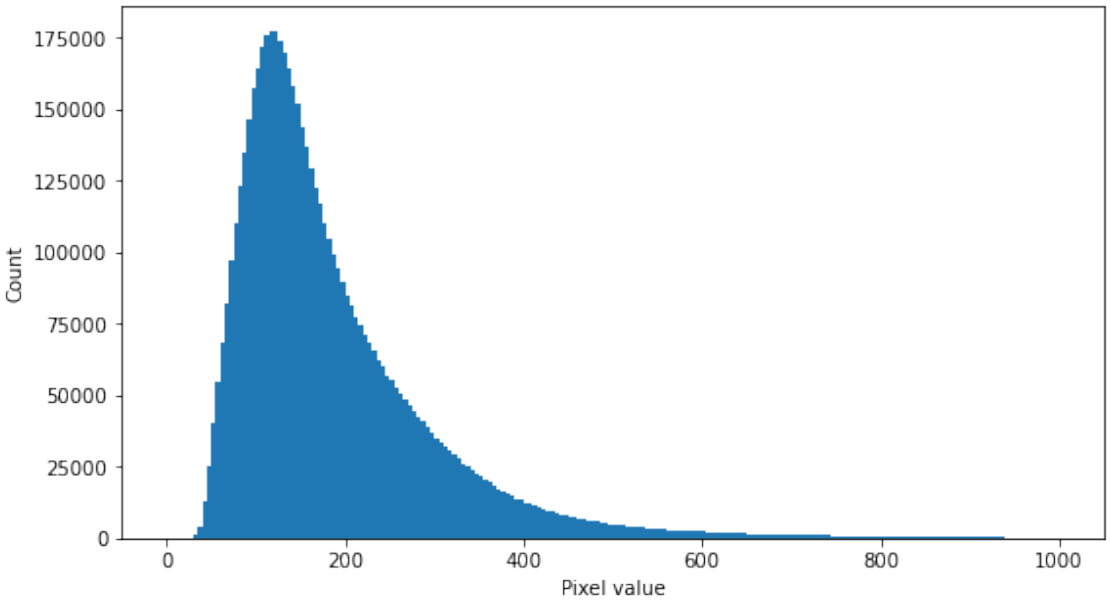
[10]: `Text(0, 0.5, 'North (km)')`



Notice that the above image is quite dark. Similar to the analysis we did above with Landsat, we can look at the histogram of the pixel values.

[11]:
```
# Histogram of raster values
fig, ax = plt.subplots(figsize=(9,5))
ax.hist(raster_polar.data.ravel(), 200, range=[0, 1000])
ax.set_xlabel('Pixel value')
ax.set_ylabel('Count')
```

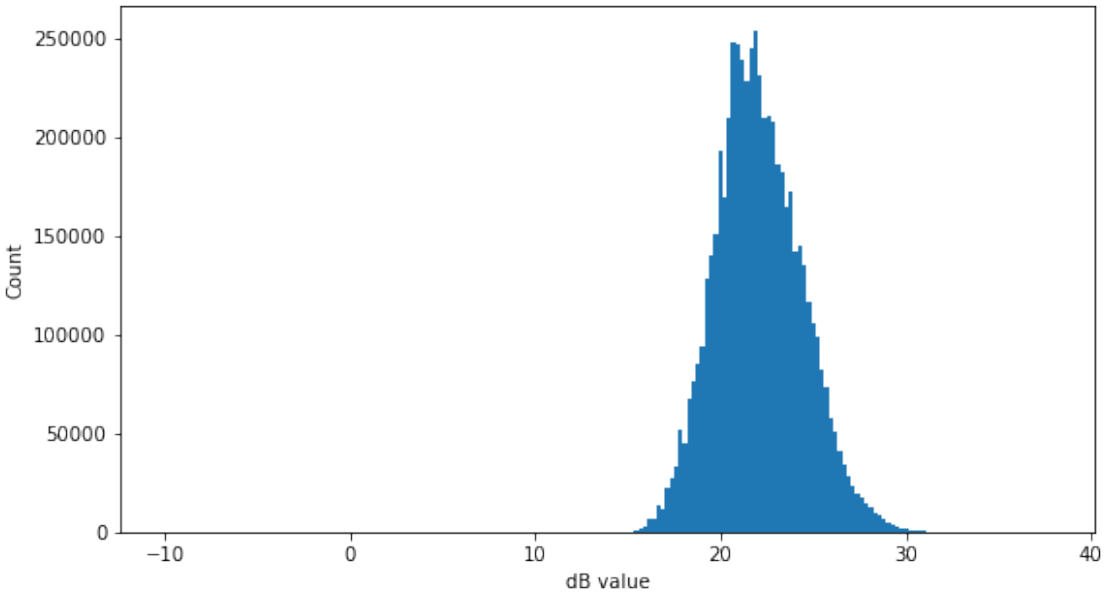
[11]:
```
Text(0, 0.5, 'Count')
```



From the histogram, we can see that the distribution of values has a "long tail", meaning there are a non-negligible number of pixels with much larger values. This distribution is due to the fact that the SAR image is recording squared amplitudes of the returned radar pulses, which will have a non-Gaussian distribution. Long story short, we can convert these values to decibels (dB) which is essentially just taking the natural log of the image.

[12]:
```
# Convert values to decibel (add small value to avoid log(0.0))
dB = 10.0 * np.log10(raster_polar.data + 0.1)
dB_raster = ice.Raster(data=dB, hdr=raster_polar.hdr)

# View histogram of dB values
fig, ax = plt.subplots(figsize=(9,5))
ax.hist(dB.ravel(), 200)
ax.set_xlabel('dB value')
ax.set_ylabel('Count')
```

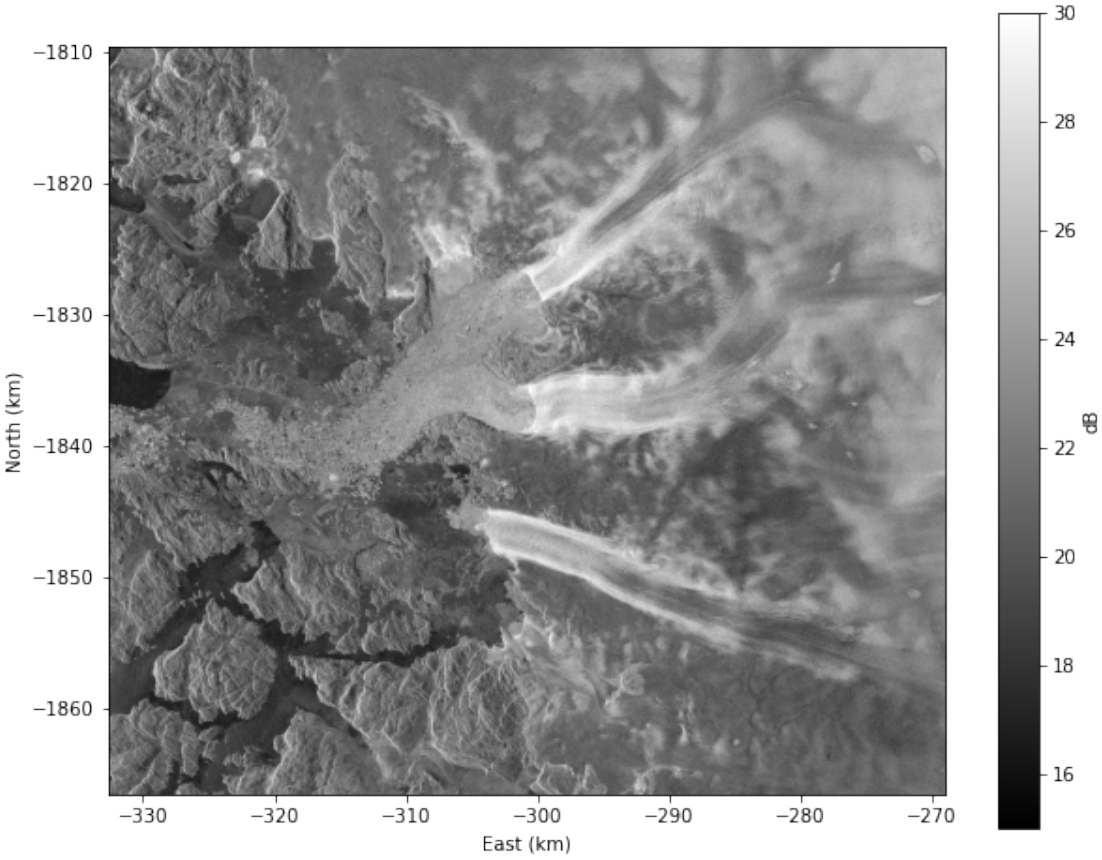
[12]:
```
Text(0, 0.5, 'Count')
```



Now we have a histogram that's much closer to a Gaussian distribution. By applying the right color limits, we arrive at a much better looking SAR image.

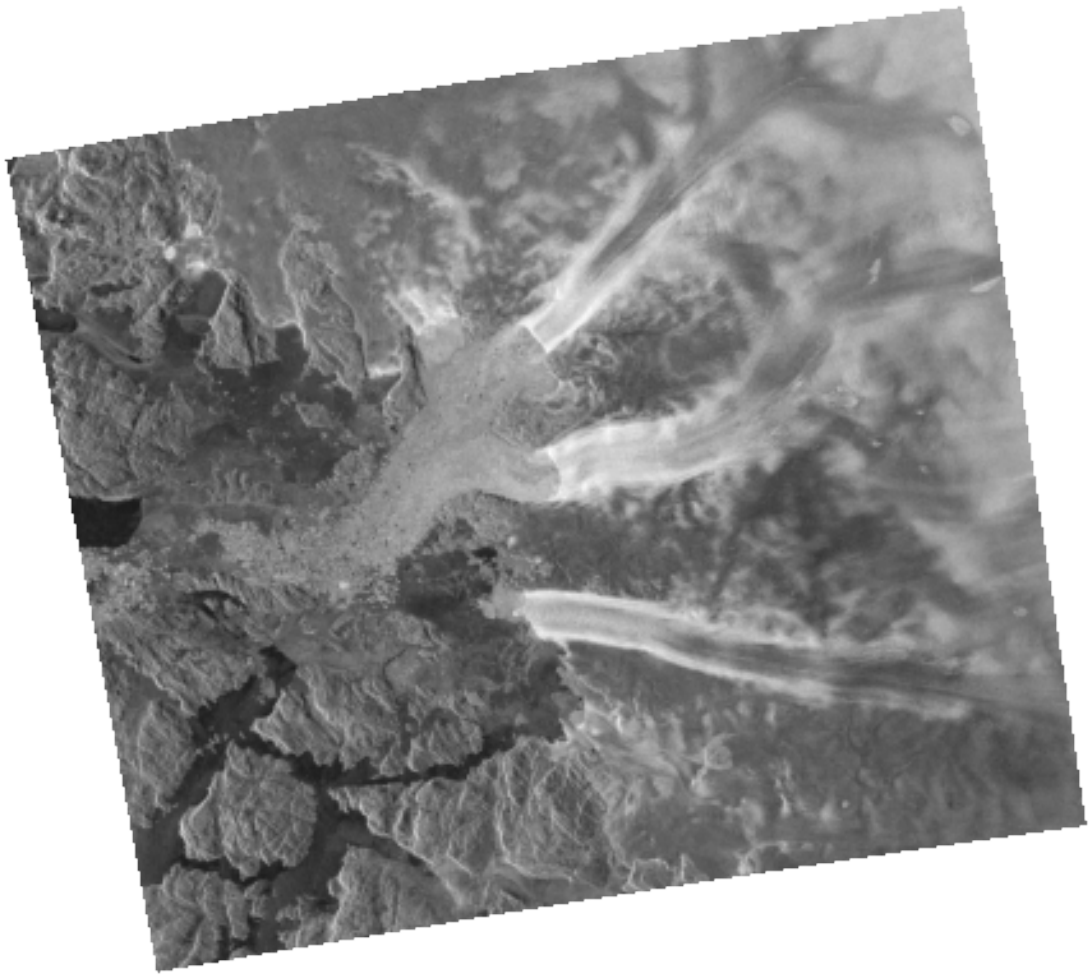
[13]:
```
# View dB image
extent = 1.0e-3 * raster_polar.hdr.extent
fig, ax = plt.subplots(figsize=(10,8))
im = ax.imshow(dB, cmap='gray', extent=extent, clim=(15, 30))
cbar = plt.colorbar(im, ax=ax, orientation='vertical')
cbar.set_label('dB')
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')
```

[13]:
```
Text(0, 0.5, 'North (km)')
```



We can check the warping from radar coordinates to geographic coordinates by making a Google Earth KML for the above image. Since Google Earth works in latitude/longitude coordinates, we will first warp the image from polar stereographic to latitude/longitude (i.e., EPSG 3413 -> 4326).

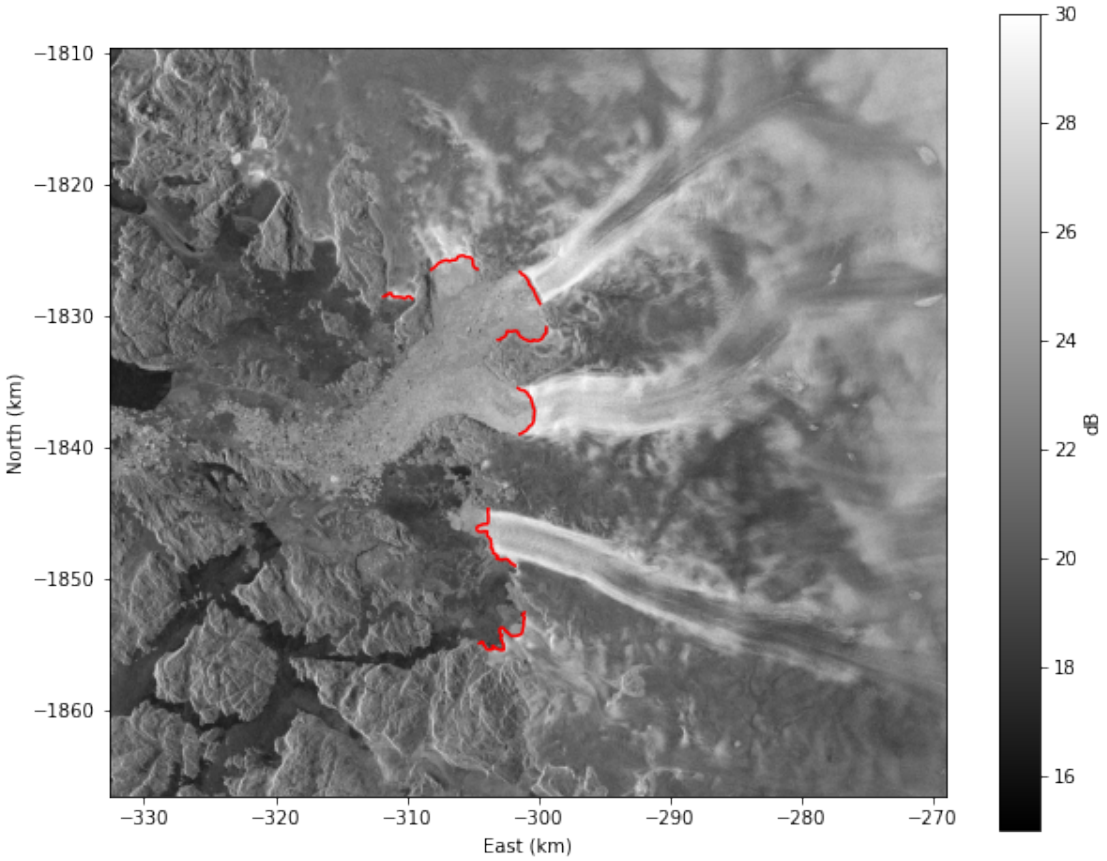
[14]:
```
# Warp radar image to EPSG:4326
raster_llh = ice.warp(dB_raster, target_epsg=4326, target_dims=dB.shape)

# Create a KML object
ice.render_kml(raster_llh, 'upernavik.kml', cmap='gray', clim=(15, 30))
```



Finally, we often need to view raster images together with known vector geometries like glacier boundaries, grounding lines, calving front locations, station locations, etc. Since we are in a Python environment with `matplotlib`, we simply need to load those geometries, convert them to the projection system of our map, and plot them on top of our map image.

[15]:
```
import gdal

# Initialize a plot as before
fig, ax = plt.subplots(figsize=(10,8))
im = ax.imshow(dB, cmap='gray', extent=extent, clim=(15, 30))
cbar = plt.colorbar(im, ax=ax, orientation='vertical')
cbar.set_label('dB')
ax.set_xlabel('East (km)')
ax.set_ylabel('North (km)')

# Open shapefile
```

```
path = 'cfl_upernavik/cfl_Upernavik_Isstroem_G305731E72859N_20141210T204802.shp'
ds = gdal.OpenEx(path, gdal.OF_VECTOR)

# Get the layer object in the shapefile
layer = ds.GetLayer()

# Loop over features in layer
for feat in layer:
    # Get the geometry
    geom = feat.GetGeometryRef()
    # Get the points in the geometry and convert to Numpy arrays
    points = geom.GetPoints()
    lon, lat = [np.array(values) for values in zip(*points)]
    # Convert longitude/latitude to polar stereographic
    x, y = ice.transform_coordinates(lon, lat, epsg_in=4326, epsg_out=3413)
    # Add to plot as red lines (make sure to convert coordinates to km)
    ax.plot(1.0e-3*x, 1.0e-3*y, '-r')

# Close the dataset
ds = None
```

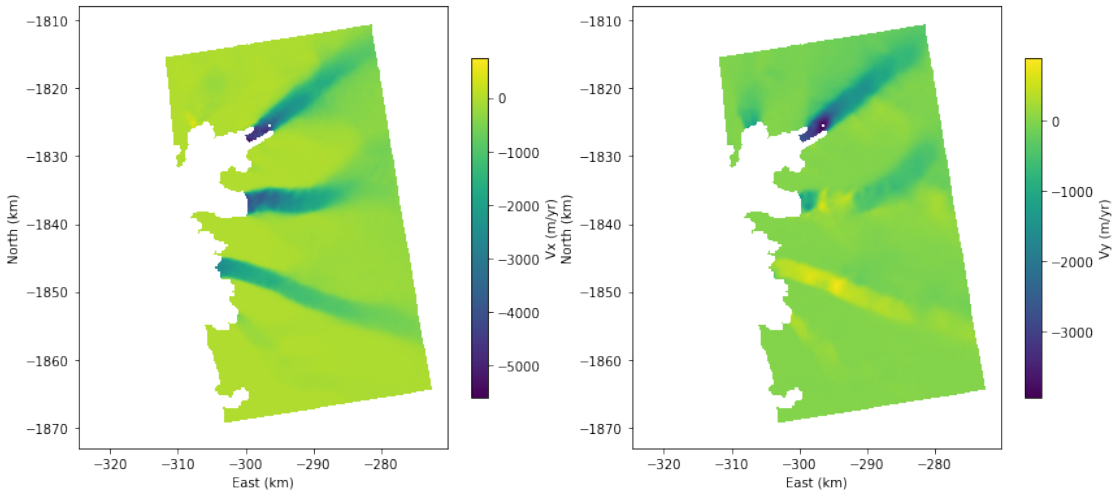


5 Ice surface velocity maps

The optical and radar images we viewed above, as well as the transformation and image processing we explored, form the fundamental building blocks for deriving products necessary for learning about ice dynamics. The two primary products are measurements of ice surface velocity and elevation over time, which allow scientists to quantify how glaciers respond to climate change. While the details of using satellite imagery to derive velocity and elevation are beyond the scope of this notebook, let's explore what velocity rasters look like and what kinds of analysis we can perform with them.

For Greenland, the best way to acquire velocity data is through the National Snow & Ice Data Center (NSIDC) Greenland Ice Sheet Mapping Project (GIMP). This public repository regularly produces multiple datasets for velocity and elevation using a number of different remote sensing datasets. Here, let's look at rasters of horizontal velocity over Upernavik Isstrom.

[16]:
```
# Load the velocity rasters
vx = ice.Raster('TSX_W72.90N_03Feb15_14Feb15_20-34-10_vx_v02.0.tif')
vy = ice.Raster('TSX_W72.90N_03Feb15_14Feb15_20-34-10_vy_v02.0.tif')
extent = 1.0e-3 * vx.hdr.extent

# Convert invalid values to NaN
vx.data[vx.data < -1e8] = np.nan
vy.data[vy.data < -1e8] = np.nan

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(14,8))
im1 = ax1.imshow(vx.data, extent=extent)
im2 = ax2.imshow(vy.data, extent=extent)
cbar1 = plt.colorbar(im1, ax=ax1, orientation='vertical', shrink=0.6)
cbar2 = plt.colorbar(im2, ax=ax2, orientation='vertical', shrink=0.6)
for ax in (ax1, ax2):
    ax.set_xlabel('East (km)')
    ax.set_ylabel('North (km)')
cbar1.set_label('Vx (m/yr)')
cbar2.set_label('Vy (m/yr)')
```



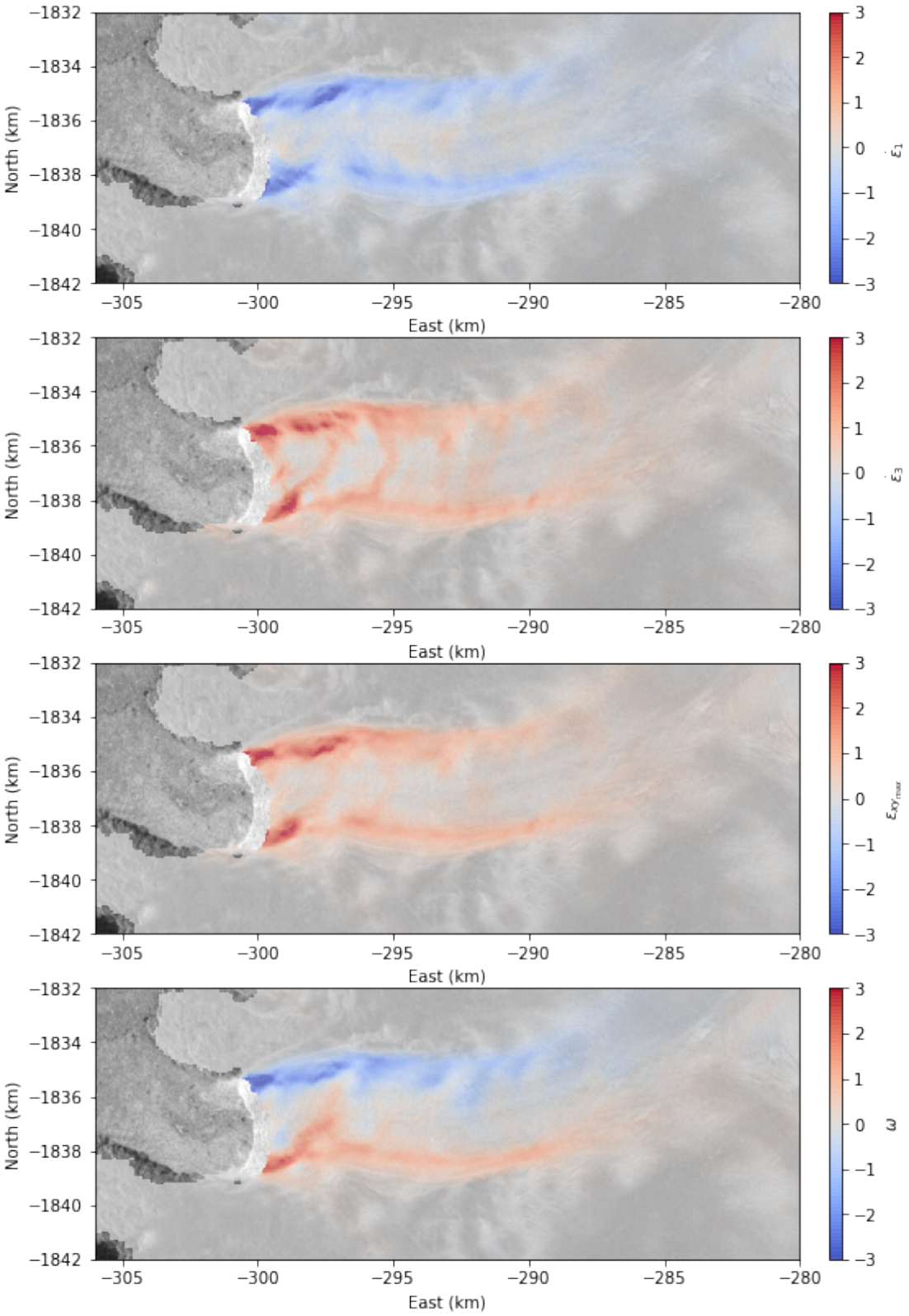
With spatially continuous velocity fields, one of the key things we would like to quantify are surface strain rates. Strain rates are non-linearly related to ice stress (through a constitutive law), so by looking at the distribution of strain rates, we can get an idea of what kinds of stresses a glacier is under. Strain rates are related to velocity gradients as:

$$\begin{bmatrix} \dot{\epsilon}_x & \dot{\epsilon}_{xy} \\ \dot{\epsilon}_{xy} & \dot{\epsilon}_y \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{\partial v}{\partial y} \end{bmatrix}$$

where x and y correspond to the East and North directions, respectively, and u and v correspond to velocities in the East and North directions, respectively. From the strain rate tensor components, we can then compute quantities like principal strain rates and rotation rates. Let's compute some of these quantities for the Upernavik Isstrom velocity fields, and we'll first focus on the middle glacier (at roughly -1835 km North).

[17]:
```
# Apply smoothing
vx_filt = ndimage.median_filter(vx.data, size=5)
vy_filt = ndimage.median_filter(vy.data, size=5)

# Compute gradients using finite differences
du_dx = np.gradient(vx_filt, vx.hdr.dx, axis=1)
du_dy = np.gradient(vx_filt, vx.hdr.dy, axis=0)
dv_dx = np.gradient(vy_filt, vx.hdr.dx, axis=1)
dv_dy = np.gradient(vy_filt, vx.hdr.dy, axis=0)

# Construct strain rate tensor components
e_x = du_dx
e_y = dv_dy
e_xy = 0.5 * (du_dy + dv_dx)
rotation = 0.5 * (dv_dx - du_dy)

# Compute principal strain rates
e1 = 0.5 * (e_x + e_y) - np.sqrt(0.25 * (e_x - e_y)**2 + e_xy**2)
e3 = 0.5 * (e_x + e_y) + np.sqrt(0.25 * (e_x - e_y)**2 + e_xy**2)
e_xy_max = np.sqrt(0.25 * (e_x - e_y)**2 + e_xy**2)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, figsize=(16,14))

# First let's plot the SAR dB image in the background
for ax in (ax1, ax2, ax3, ax4):
    db_extent = 1.0e-3 * raster_polar.hdr.extent
    ax.imshow(dB, cmap='gray', extent=db_extent, clim=(15, 30))

# Now plot the strain components
im1 = ax1.imshow(e1, extent=extent, cmap='coolwarm', clim=(-3, 3), alpha=0.7)
```

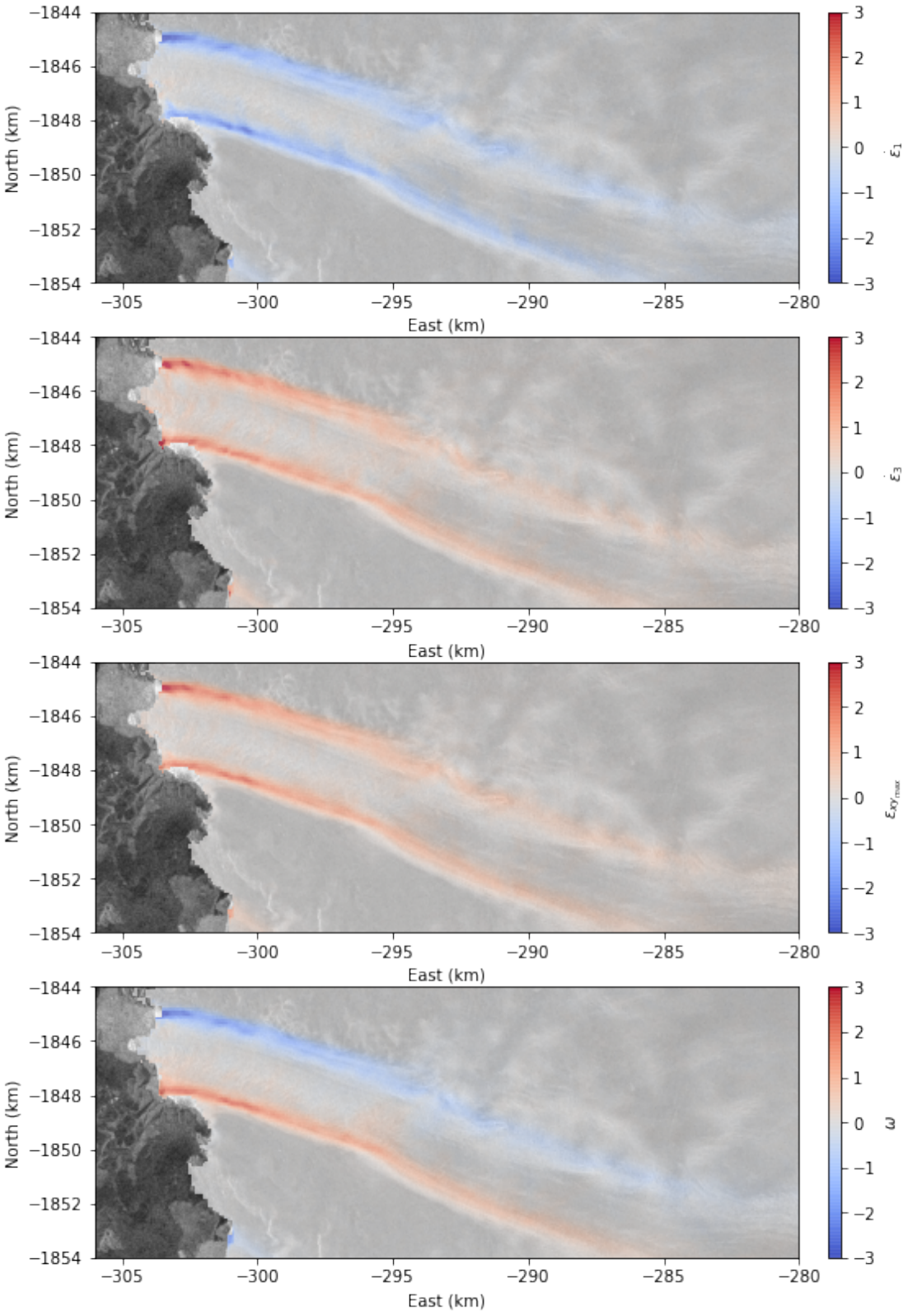
```
im2 = ax2.imshow(e3, extent=extent, cmap='coolwarm', clim=(-3, 3), alpha=0.7)
im3 = ax3.imshow(e_xy_max, extent=extent, cmap='coolwarm', clim=(-3, 3),␣
 ↪alpha=0.7)
im4 = ax4.imshow(rotation, extent=extent, cmap='coolwarm', clim=(-3, 3),␣
 ↪alpha=0.7)

# Colorbar labels
labels = [r'$\dot{\epsilon}_1$', r'$\dot{\epsilon}_3$',␣
 ↪r'$\epsilon_{xy_{max}}$', r'$\omega$']

# Some decorations
for ax, im, label in zip((ax1, ax2, ax3, ax4), (im1, im2, im3, im4), labels):
    cbar = plt.colorbar(im, ax=ax, orientation='vertical', pad=0.02)
    cbar.set_label(label)
    ax.set_xlabel('East (km)')
    ax.set_ylabel('North (km)')
    ax.set_xlim(-306, -280)
    ax.set_ylim(-1842, -1832)
```


North (km)
East (km)
−1832
−1834
−1836
−1838
−1840
−1842
−305
−300
−295
−290
−285
−280
3
2
1
0
−1
−2
−3
$\dot{\varepsilon}_1$
$\dot{\varepsilon}_3$
$\dot{\varepsilon}_{xy_{max}}$
ω

We can see that the defining feature for this glacier is the concentration of strain in the margins where shear strains dominate. This is expected since to a good approximation, velocity is zero at the margins (no slip condition), which results in a shearing deformation mode. From the map of the maximum tensile principal strain rate, $\dot{\epsilon}_3$, we can also observe bands of extensional strain rate closer to the terminus.

Let's now look at the strain rates for the glacier just to the south of this one.

[18]:
```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, figsize=(16,14))

# First let's plot the SAR dB image in the background
for ax in (ax1, ax2, ax3, ax4):
    db_extent = 1.0e-3 * raster_polar.hdr.extent
    ax.imshow(dB, cmap='gray', extent=db_extent, clim=(15, 30))

# Now plot the strain components
im1 = ax1.imshow(e1, extent=extent, cmap='coolwarm', clim=(-3, 3), alpha=0.7)
im2 = ax2.imshow(e3, extent=extent, cmap='coolwarm', clim=(-3, 3), alpha=0.7)
im3 = ax3.imshow(e_xy_max, extent=extent, cmap='coolwarm', clim=(-3, 3),␣
 ↪alpha=0.7)
im4 = ax4.imshow(rotation, extent=extent, cmap='coolwarm', clim=(-3, 3),␣
 ↪alpha=0.7)

# Colorbar labels
labels = [r'$\dot{\epsilon}_1$', r'$\dot{\epsilon}_3$',␣
 ↪r'$\epsilon_{xy_{max}}$', r'$\omega$']

# Some decorations
for ax, im, label in zip((ax1, ax2, ax3, ax4), (im1, im2, im3, im4), labels):
    cbar = plt.colorbar(im, ax=ax, orientation='vertical', pad=0.02)
    cbar.set_label(label)
    ax.set_xlabel('East (km)')
    ax.set_ylabel('North (km)')
    ax.set_xlim(-306, -280)
    ax.set_ylim(-1854, -1844)
```


North (km)
East (km)
−1844
−1846
−1848
−1850
−1852
−1854
−305
−300
−295
−290
−285
−280
3
2
1
0
−1
−2
−3
$\dot{\varepsilon}_1$
$\dot{\varepsilon}_3$
$\dot{\varepsilon}_{xy_{max}}$
ω

Similar to the glacier to the north, the strain rates are dominated by shear strain. The maximum tensile principal strain rates do not show as strong banding features, which may indicate a weaker bed or more homogenous distribution of bed friction.

[ ]: