Remote Sensing in the Social Sciences

Fall 2021

Contents

Introduction **1**

PreLab: Getting Started **2**

Overview . . . 2

0.1 Data and Methods . . . 4

0.2 Images and Image Collections . . . 5

0.3 Geometries . . . 6

0.4 Features and Feature Collections . . . 7

0.5 Methods: Reducers . . . 8

0.6 Joins and Arrays . . . 10

0.7 Additional Resources . . . 11

1 Remote Sensing Background **11**

Overview . . . 11

1.1 What is a digital image? . . . 12

1.2 Spatial Resolution . . . 15

1.3 Spectral Resolution . . . 18

1.4 Temporal Resolution . . . 20

1.5 Radiometric Resolution . . . 21

1.6 Resampling and ReProjection . . . 21

1.7 Additional Exercises . . . 22

Where to submit . . . 23

Introduction

Research involving remote sensing data acquisition and analysis has evolved significantly in the past few decades. In the earliest years of satellite-based remote sensing analysis, only a handful of governments had the capability to deploy satellites and reliably process satellite imagery, and its use was largely limited to the military and intelligence communities.

In the late 1950s, the US and Europe established the National Aeronautics and Space Administration (NASA) and the (predecessors to) the European Space Agency (ESA) so as to support a civilian space program as well as space and aeronautics research.

Even then, however, data access was unwieldy and often costly. For example, even if a researcher had identified the data they wanted to work with, they would have had to go through the time-intensive steps of downloading the data on a computer with sufficient memory and performing a series of pre-processing steps (e.g., ortho-rectification and atmospheric corrections), all before they began to assess their main questions of interest.

Why Google Earth Engine (GEE)

As part of Google's quest to make the world's information universally accessible and useful, Google Earth Engine emerged in 2010 to aid in organizing and simplifying geospatial data in a way that supports an end-to-end solution for analysis.

This resource simplifies many of the historical problems that remote sensing researchers have struggled with, including:

1. GEE now features petabytes of imagery from both public and private sources, including the most-used Landsat, MODIS, and Sentinel data.
2. GEE allows users to process the data and conduct sophisticated analysis on their data of choice within Google's Cloud environment (for example even allowing advanced Machine Learning using TensorFlow Processing Units).
3. GEE geo-rectifies the image and provides pre-built algorithms that facilitate analysis. In case you need to build your own algorithms, Google Earth Engine has built functionality within JavaScript and Python, which in turn extends the opportunities for processing data and displaying results.
4. Users can even import their own data and work with it within GEE while still maintaining ownership of the analysis and functions written within GEE (i.e., use it for noncommercial purposes).

As scholars interested in using remote sensing data for public social science research questions, Google Earth Engine can open up a variety of new resources for your analysis.

License and Attribution

The foundation of the first series of lab exercises were generously shared with us by Nicholas Clinton of Google and Dr. David Saah of the University of San Francisco, Geospatial Analysis Lab. We (Elinor Benami and Ozzy Campos) thank them for this great public good and take responsibility for any errors that arose from our adaptation.

PreLab: Getting Started

Overview

The purpose of this lab is to introduce some of the functionality and structure of Google Earth Engine (GEE) before we get into the practical labs. This tutorial will provide a brief introduction to the GEE Javascript interface (the Code Editor) and using GEE resources. At the completion of the lab, you will be able to access GEE imagery, upload your own assets, and explore the metadata for a given feature.

0.0.0.1 Learning Outcomes

- Navigate basic GEE Resources
- Describe the major GEE data types and their associated methods

0.0.0.2 Setting up an Account

To begin, ensure you sign-up for the Google Earth Engine here. Registration is free and straightforward, but it takes approximately 24 hours to be approved to use the code editor. While waiting, let's get familiar with the Google Earth Engine. The video below is a quick introduction to Google Earth Engine that Ozzy assembled to to get you familiar with the available resources.

Video

0.0.0.3 Importing data

In addition to the petabytes of satellite imagery and products that GEE has available, Google Earth Engine also allows you to work with your own raster, vector, and tabular data. This is process is automatically linked to the Google Drive account that signed up for GEE.

If you are not familiar with Google Drive, the 'Getting Started Guide' reviews the basics of initializing and organizing your Google Drive account. Although Google Cloud Platform Storage is beyond the scope of this course, below is some additional helpful documentation on working with external data.

- Managing Assets
- Import Raster
- Import Vector / Tabular Data ** Note that GEE only supports Shapefiles and `.csv`files ***
- Exporting Data

0.0.0.4 Gecomputation with GEE: Server vs. Client

Understanding the basics of how Google Earth Engine works is critical for its effective use. The Developer's overview provides much more detail on the intricacies of how GEE processes data on the Google Cloud Platform, but in the simplest terms, there are two sides to the process - the `client` side and `server` side.

When you open your web browser and begin to work in the code editor, that is considered the `client` side. You can write JavaScript code in the editor and the code will be processed within your browser. The code below simply creates a variables `x` and `y`, adds them together as the variable `z` and prints the result, which shows up in the console of the code editor. Even though the code is written in the GEE editor, it plays no role in the execution of this code - your browser executes it.

```
var x = 1; var y = 2;
var z = x + y;
print(z)
```

To begin using the cloud computing resources of GEE effectively, we can then call upon the server side of the operations. Let's say we want to import an image collection. In the snippet below, you can see that there is an `ee` before the `ImageCollection` constructor. In simple terms, this signals to Earth Engine that we will be using its resources. Without that indicator, GEE will cede operations to the server.

```
var sentinelCollection = ee.ImageCollection('COPERNICUS/S2_SR');
```

Over time, you will gain experience understanding the role of working with JavaScript on the `client` side and the `server` side, but the main point in this section is that when programming, we will be building 'packages' that draw upon GEE resources to complete their operations.

An extension of this topic is listed here, along with discussions of programming specific topics (i.e., mapping instead of looping).

0.0.0.5 JavaScript

The intent of this course is not to teach the intricacies of programming within JavaScript. JavaScript is the core language for web development, and you will likely find that many of the tutorials and resources you find will not be directly relevant to the type of JavaScript that you will need to work in Earth Engine (ie, working with React, JQuery, dynamic app development, etc). JavaScript was

chosen because it is an extremely popular language (~97% of websites use it in some fashion) and as an object-oriented language, it is well-suited to pair objects (in this case, imagery provided by Google Earth Engine) with methods (such as using the `reduce` function to summarize the analytical information from a processed image).

Several excellent resources exist that can help you in working with JavaScript. One such resource is Javascript.info, which provides a thorough overview of working with JavaScript. In this tutorial, focus on part I, as part II and III are focused on web development.

W3Schools provides good information on each individual component of working with JavaScript. For instance, if you see the word `var` and wanted more information on it, W3Schools has some helpful definitions and code snippets that will be of use.

Finally, JavaScript & JQuery is an excellent, well-designed book that goes through the fundamentals of working with JavaScript and provides helpful illustrations and use cases. The second half of the book is outside the scope of this course, but if you did want to extend your skillset, this book is a great starting point.

0.1 Data and Methods

Core Components of Google Earth Engine Operations

Most Google Earth Engine tutorials begin with an introduction to the data structures and the operations you can use to analyze your data structures. To work effectively with GEE, it is essential that you understand these core components and how to complete basic operations with each of them.

Intro to Data

- `Image`
  - Raster Image, a fundamental data type within Earth Engine
- `ImageCollection`
  - A "stack" or sequence of images with the same attributes
- `Geometry`
  - Vector data either built within Earth Engine or imported
- `Feature`
  - `Geometry` with specific attributes.
- `FeatureCollection`
  - Set of features that share a similar theme
- `Reducer`
  - A method used to compute statistics or perform aggregations on the data over space, time, bands, arrays, and other data structures.
- `Join`
  - A method to combine datasets (`Image` or `Feature` collections) based on time, location, or another specified attribute
- `Array`
  - A flexible (albeit sometimes inefficient) data structure that can be used for multi-dimensional analyses.

0.2 Images and Image Collections

0.2.1 Images

Images are **Raster** objects composed of:

- Bands, or layers with a unique:
  - Name
  - Data type
  - Scale
  - Mask
  - Projection
- Metadata, stored as a set of properties for that band.

You can create images from constants, lists, or other objects. In the code editor 'docs', you'll find numerous processes you can apply to images.

Ensure that you do not confuse an individual image with an image collection, which is a set of images grouped together, most often as a time series, and often known as a `stack`.

0.2.2 Image Collections

Let's analyze the code below, which is an established method of extracting one individual image from an image collection. You can copy and paste this code snippet into the code editor to follow along.

On the first line, we see that we are creating a JavaScript variable named `first`, and then using `ee` in front of `ImageCollection`, which signifies we are requesting information from GEE. The data we are importing ('COPERNICUS/S2_SR') is the Sentinel-2 MSI: MultiSpectral Instrument, Level-2A, with more information found in the dataset documentation.

The next four steps further refine the extraction of an image from an image collection.

1. `.filterBounds` filters data to the area specified, in this case a geometry Point that was created within GEE.
2. `.filterDate` filters between the two dates specified (filtering down to images collected in 2019)
3. `.sort` organizes the images in descending order based upon the perentage of cloudy pixels (this is an attribute of the image, which can be found in the 'Image Properties' tab in the dataset documentation)
4. `.first` is a JavaScript method of choosing the first image in the list of sorted images

As a result, we can now use the JavaScript variable 'first' to visualize the image.

`Map.centerObject()` centers the map on the image, and the number is the amount of zoom. The higher that value is, the more zoomed in the image is - you'll likely have to adjust via trial-and-error to find the best fit.

`Map.addLayer()` adds the visualization layer to the map. Image/image collections will each have a unique naming convention of their bands, so you will have to reference the documentation. GEE uses Red-Green-Blue ordering (as opposed to the popular Computer Vision framework, OpenCV, which uses a Blue-Green-Red convention). `min` and `max` are the values that normalize the value of each pixel to the conventional 0-255 color scale. In this case, although the maximum value of a pixel in all three of those bands is 2000, for visualization purposes GEE will normalize that to 255, the max value in a standard 8-bit image.

There is a comprehensive guide to working on visualization with different types of imagery that goes quite in-depth on working with different types of imagery. It is a worthwhile read, and covers some interesting topics such as false-color composites, mosaicking and single-band visualization. Work with some of the code-snippets to understand how to build visualizations for different sets of imagery.

```
var first = ee.ImageCollection('COPERNICUS/S2_SR')
                .filterBounds(ee.Geometry.Point(-70.48, 43.3631))
                .filterDate('2019-01-01', '2019-12-31')
                .sort('CLOUDY_PIXEL_PERCENTAGE')
                .first();
Map.centerObject(first, 11);
Map.addLayer(first, {bands: ['B4', 'B3', 'B2'], min: 0, max: 2000}, 'first');
```

0.2.3 Sensed versus Derived Imagery

One additional note: GEE provides a rich suite of datasets, and while many of them are traditional sensed imagery, others are derived datasets. For instance, the *Global Map of Oil Palm Plantations* dataset provides is derived from analysis on the Sentinel composite imagery. If you look at the 'Bands', there are only three values, which refer to categories of palm plantations. Datasets such as these will have different methods for visualizing the data or working as a mosaic.

0.3 Geometries

Google Earth Engine handles vector data with the Geometry type. Traditionally, this means

- Point
- Line
- Polygon

However, GEE has several different nuances.

- `Point`
- `LineString`
  - List of Points that do not start and end at the same location
- `LinearRing`
  - LineString which does start and end at the same location
- `Polygon`
  - List of LinearRing's - first item of the list is the outer shell and other components of the list are interior shells

GEE also recognizes `MultiPoint`, `MultiLineString` and `MultiPolygon`, which are simply collections of more than one element. Additionally, you can combine any of these together to form a `MultiGeometry`. Here is a quick video of working with the Geometry tools within GEE.

Once you have a set of geometries, there are geospatial operations you can use for analysis, such as building buffer zones, area analysis, rasterization, etc. The documentation contains some basic examples to show you how to get started, although there are many more functions listed under the 'Docs' tab in the Code Editor.

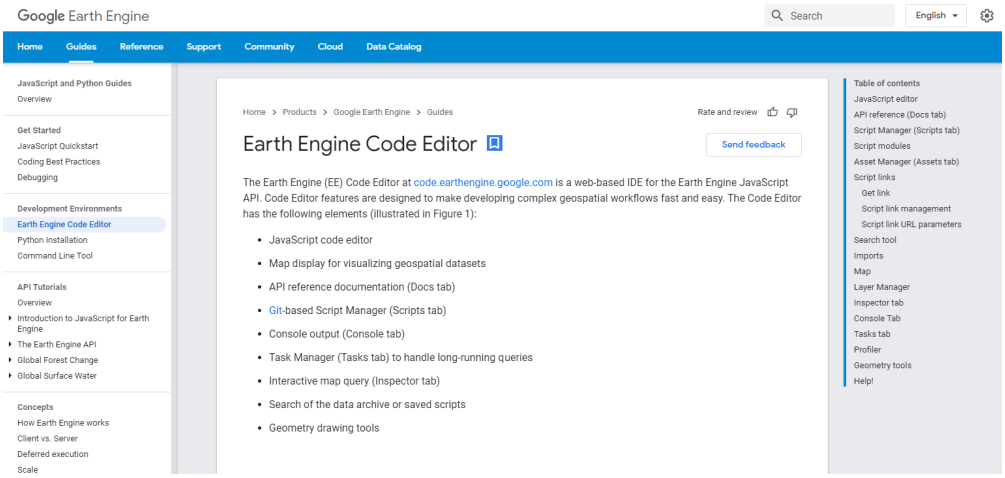


Figure 1: A view of the google earth engine code editor

0.4 Features and Feature Collections

0.4.1 Features

At the most basic definition, a Feature in GEE is an object which stores a `geometry` property (`Point`, `Line`, `Polygon`) along with it's associated properties. GEE uses the GeoJSON format to store and transmit these features. In the previous video, we saw how to build geometries within Google Earth Engine, a feature adds meaningful information to it. This would be a good section to review working with dictionaries with JavaScript.

Let's say we created an individual point, which we want to associate with data that we collected. The first line establishes the variable `point`, which is then used as the `geometry` to create a `feature`. The curly braces represent a JavaScript dictionary, which creates Key:Value pairs, which in our case is the type of tree and a measurement of the size. this new variable, `treeFeature`, now contains geographic information along with attribute data about that point.

```
// geometry created from within GEE
var point = ee.Geometry.Point([-79.68, 42.06]);
// Create a Feature from the geometry
var treeFeature = ee.Feature(point, {type: 'Pine', size: 15});
```

Obviously this is just one point, but JavaScript and GEE engine provide functionality for bringing different data sources together and automatically associating geometries with attribute data. This can be done within GEE or outside, depending on your preferences.

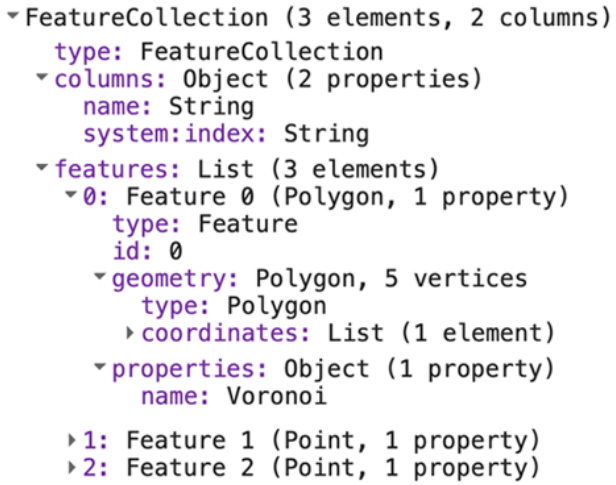
0.4.2 Feature Collections

Just like the relationship between images and image collections, Feature Collections are Features that can be grouped together for ease of use and analysis. They can be different types and combinations of geometry, as well as associated tabular data. The code segment from the documentation consolidates the operations discussed earlier. Each line has an interior layer which creates the geometry (`ee.Geometry.---()`), which is then associated with attribute data (information within the {}) and then converted to a Feature. This variable is a JavaScript`list`, which contains three separate features. This is then converted to a Feature Collection with the command `ee.FeatureCollection(features)`

```
// Make a list of Features.
var features = [
  ee.Feature(ee.Geometry.Rectangle(30.01, 59.80, 30.59, 60.15), {name: 'Voronoi'}),
  ee.Feature(ee.Geometry.Point(-73.96, 40.781), {name: 'Thiessen'}),
  ee.Feature(ee.Geometry.Point(6.4806, 50.8012), {name: 'Dirichlet'})
];

// Create a FeatureCollection from the list and print it.
var fromList = ee.FeatureCollection(features);
print(fromList);
```

If you take this code block and run it in Google Earth Engine, you can see the information that is contained within the FeatureCollection, which has three elements (Features) and two columns (the `index` and the `properties`). By clicking on the dropdown next to each one, you can see that the first feature is a Polygon that has the name of 'Voronoi'.



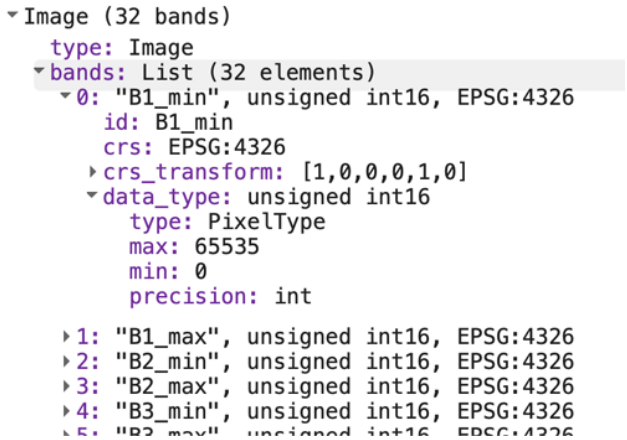
Once you have information in a Feature Collection, you can filter it to find specific information, such as the name of an object or based on the size of a polygon, or provide aggregated analysis. The documentation on working with Feature Collections is comprehensive, and provides many ideas on how to use them efficiently in in your analysis.

0.5 Methods: Reducers

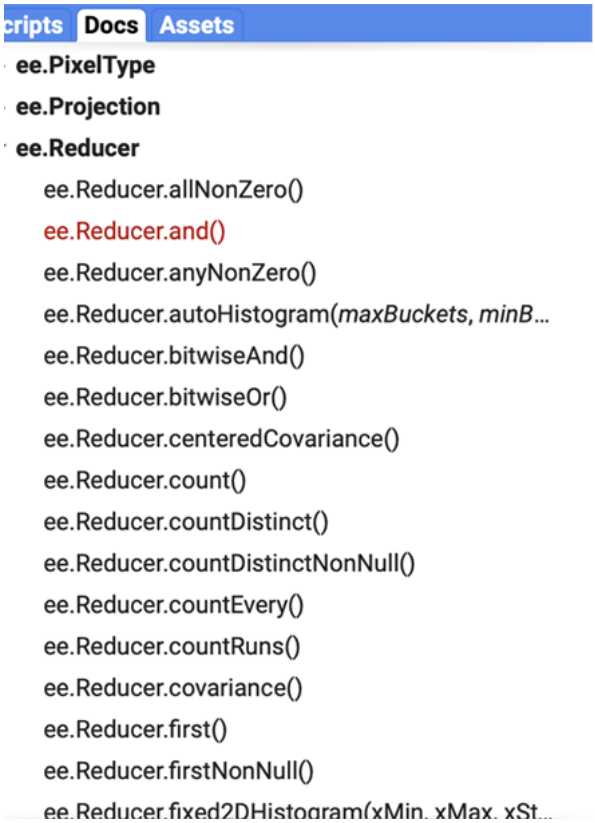
Up until now, we have focused on objects: Images, Features, and Geometries. Reducers are a method of aggregating data for analysis. For instance, we could take an Image Collection and use `reducer` to find the average value at each pixel, resulting in a single layer. Or we could reduce an image to a set of regions, grouping similar data together to create a simplifed map. The applications of Reducer are endless, and can be applied to both Images and Features. There are different functions for different object types, and Reducer can be both combined and sequenced to create a chain of analysis. From the documentation, the code chunk below creates the variable `collection` which is a collection that is filtered to the year 2016 and defined to a specific point. The variable `extrema` then reduces the dataset to identify the minimum and maximum value at that specific point for every band.

```
// Load and filter the Sentinel-2 image collection.
var collection = ee.ImageCollection('COPERNICUS/S2')
    .filterDate('2016-01-01', '2016-12-31')
    .filterBounds(ee.Geometry.Point([-81.31, 29.90]));
// Reduce the collection.
var extrema = collection.reduce(ee.Reducer.minMax());
```

If you print `extrema` in the console, you can see that the result is 32 separate 'bands', which represents the minimum and maximum value for all 16 bands in the Sentinel data. In the screenshot below, you can expand the first 'band', which identifies the attributes of the minimum value of Band 1.



There are hundreds of different operations for using `Reducer`, with the functions listed on the left hand table under 'Docs'. Certain functions will only work with specific object types, but follow along with the Reducer documentation to get a better understanding of how to aggregate data and extract meaningful results. Getting familiar with Reducer is an essential component to working with Google Earth Engine.



0.6 Joins and Arrays

0.6.1 Join

If you have programmed in the past, joining data togetgher is likely a familiar concept. This process assicates information from one set of data with relevant data from another set on a specific attribute. Let's say you have an Image Collection of Landsat data that is filtered to the first six months of the year 2016 and a bounding box of your area of study. You also have a table of Redwood tree locations that is filtered to the same area of study, although it contains information over the past decade. You can use a Join to associate information about the trees from the Feature Collection and include it in the Image Collection, keeping only the relevant data. You now have a dataset with useful information from both the Image Collection and Feature Collection in one location. Although there are different types of joins, the process brings information together, keeping only relevant information. The documentation on Joins goes over specific examples and concepts, but a crucial component is understanding the type of join you need the three most prominent within GEE are:

- Left Join
  - Keeps all the information from the primary dataset, and only information that joins from the secondary datset
- Inner Join
  - Keeps only the information where the primary and secondary data match
- Spatial Join
  - A join based on spatial location (ie, keep only the geometry points that fall within a specified buffer)

GEE provides some unique types of joins, including 'Save-All', 'Save-Best' and 'Save-First', which are useful if you want to look at a specific area.

0.6.2 Arrays

Arrays are a collection of data where information is stored contiguously - matrices are a multi-dimensional array. For instance, an image might have 1024 rows and 1024 columns. Each row is an array, each column is an array, and taken together, you have a 2-dimensional array, also known as a matrix. If the image has three separate color channels, then that is a 3-dimensional array. Some of the terminology changes depending on discipline (ie, physics vs. computer science), but if you are familiar with working with matrices and arrays in programming languages such as Matlab or OpenCV, it is important to understand the role of arrays within GEE.

In fact, Google Earth Engine states that working with arrays outside of the established functions that they have built is not recommended, as GEE is not specifically designed for array-based math, and will lead to unoptimized performance.

There is a very informative video that delves into the engineering behind Google Earth Engine, but in this course we will only be doing a limited amount with array transformations and Eigen Analysis. In many cases, you will probably be better off aggregating the specific data and then conducting array mathematics with programming frameworks geared to that context.

0.7 Additional Resources

- Google Earth Engine link
- Code Editor Map – what all the features on the code editor mean
- Datasets
- Case Studies
- Google Earth Engine Blog
- Video tutorials on using GEE (from the Earth Engine Users' Summit)

1 Remote Sensing Background

Overview

The purpose of this lab is to introduce digital images, datum, and projections, as well as demonstrate concepts of spatial, spectral, temporal and radiometric resolution. You will be introduced to image data from several sensors aboard various platforms. At the completion of the lab, you will be able to understand the difference between remotely sensed datasets based on sensor characteristics and how to choose an appropriate dataset based on these concepts.

1.0.0.1 Learning Outcomes

1. Describe the following terms:
    - Digital image
    - Datum
    - Projection
    - Resolution (spatial, spectral, temporal, radiometric)
2. Navigate the Google Earth Engine console to gather information about a digital image
3. Evaluate the projection and attributes of a digital image
4. Apply image visualization code in GEE to visualize a digital image

1.1 What is a digital image?

A digital image is a matrix of same-sized pixels that are each defined by two main attributes: (1) the position, as defined by rows and columns and (2) the a value associated with that position.

A digital image 8 pixels wide by 8 pixels tall could thus look like the image below. Note though you can reference the position from a given axis, typically, image processing uses the top-left of an image as the reference point, as in the below image.

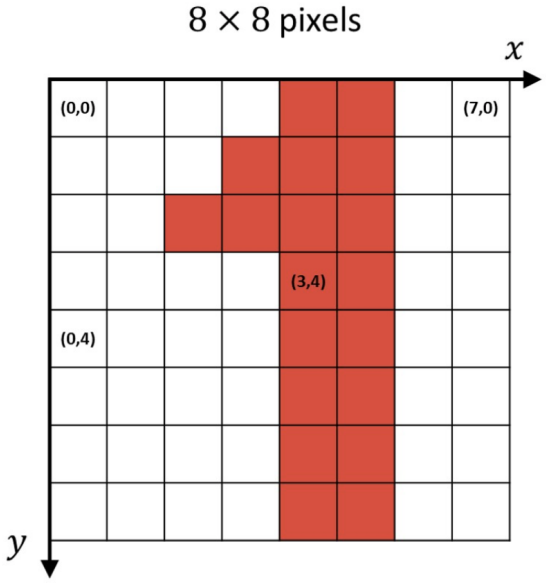


Figure 2: Digital Image Example

A "traditional" optical photograph typically represents three layers (often the brightness values represented in the Red, Blue, and Green portions of the electromagnetic spectrum). Together, these three layers create a full-color photograph that is represented by a three dimensional matrix where pixel position is characterized by the (1) row (2) column (3) *and* layer.

Digital images are also often called rasters, and ESRI has a great overview of rasters used in geospatial analysis featured here.

1.1.1 From digital image to geospatial image

A digital image is a flat, square surface. However, the earth is round (spherical).

Thus to make use of the synoptic properties of remote sensing data, we need to align the pixels in our image to a real-world location. There's quite a bit of mathematics involved in this process, but we will focus on two main components - establishing a Geographic Coordinate System (GCS) and a Projected Coordinate System (PCS).

The GCS defines the spherical location of the image whereas the PCS defines how the grid around that location is constructed. Because the earth is not a perfect sphere, there are different GCS for different regions, such as 'North American Datum: 83' which is used to accurately define North America, and 'World Geodetic System of 1984', which is used globally.

The PCS then constructs a flat grid around the GCS in which you can create a relationship between each pixel of a 2-dimensional image to the corresponding area on the world. Some of the common PCS formats include EPSG, Albers Conic, Lambert, Eckert, Equidistant, etc. Different types of PCS's are designed for

different formats, as the needs of a petroleum engineer working over a few square miles will differ from than a climate change researcher at the scope of the planet, for example.

ESRI (the makers of ArcGIS) has an article discussing the difference between GCS and PCS that provides further context. While you should be aware of the differences between GCS and PCS's – especially when you intend to run analyses on the data you download from GEE in another system such as R, Python, or Arc – GEE takes care of much of the complexity of these differences behind the scenes. Further documentation on the GEE methodology can be found here. In our first exercise, we will show you how to identify the PCS so you can understand the underlying structure.

Furthermore, remote sensing data often consists of more than the three Red-Green-Bluye layers we're used to seeing visualized in traditional photography. For instance, the Landsat 8 sensor has eleven bands capturing information from eleven different portions of the electromagentic spectrum, including near infrared (NIR) and thermal bands that are invisible to the human eye. Many Machine Learning projects also involve normalizing or transforming the information contained within each of these layers, which we will return to in subsequent labs.

In sum, understanding the bands available in your datasets, identifying which bands are necessary (and appropriate) for your analysis, and ensuring that these data represent consistent spatial locations is essential. While GEE simplifies many complex calculations behind the scenes, this lab will help us unpack the products available to us and their essential characteristics.

1.1.1.1 Summary Each pixel has a position, measured with respect to the axes of some coordinate reference system (CRS), such as a geographic coordinate system. A CRS in Earth Engine is often referred to as a projection, since it combines a shape of the Earth with a datum and a transformation from that spherical shape to a flat map, called a projection.

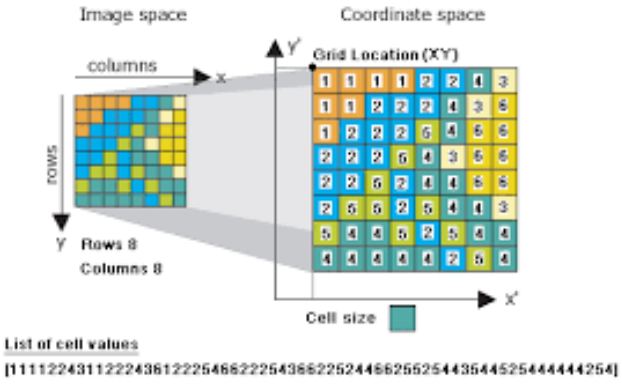


Figure 3: A pixel, raster, and a CRS

1.1.2 Visualize a Digital Image

Let's view a digital image in GEE to better understand this concept:

1. In the map window of GEE, click on the Point geometry tool using the geometry drawing tools to define your area of interest (for the purposes of consistency in this exercise, place a point on the Virginia Tech Drillfield, which will bring you roughly to `[-80.42,37.23]`). As a reminder, you can find more information on geometry drawing tools in GEE's Guides. Name the import `point`.

2. Import NAIP imagery by searching for 'naip' and choosing the *'NAIP: National Agriculture Imagery Program'* raster dataset. Name the import `naip`.

3. Get a single, recent NAIP image over your study area and inspect it:

```
//  Get a single NAIP image over the area of interest.
var  image = ee.Image(naip
                      .filterBounds(point)
                      .sort('system:time_start', false)
                      .first());
//  Print the image to the console.
print('Inspect the image object:', image);
//  Display the image with the default visualization.
Map.centerObject(point, 18);
Map.addLayer(image, {}, 'Original image');
```

4. Expand the image object that is printed to the console by clicking on the dropdown triangles. Expand the property called `bands` and expand one of the bands (0, for example). Note that the CRS transform is stored in the `crs_transform` property underneath the band dropdown and the CRS is stored in the `crs` property, which references an EPSG code.

   > **EPSG Codes** are 4-5 digit numbers that represent CRS definitions. The acronym EPGS, comes from the (now defunct) European Petroleum Survey Group. The CRS of this image is EPSG:26917. You can often learn more about those EPSG codes from thespatialreference.org or from the ESPG homepage.
   >
   > The CRS transform is a list `[m00, m01, m02, m10, m11, m12]` in the notation of this reference. The CRS transform defines how to map pixel coordinates to their associated spherical coordinate through an affine transformation. While affine transformations are beyond the scope of this class, more information can be found at Rasterio, which provides detailed documentation for the popular Python library designed for working with geospatial data.

5. In addition to using the dropdowns, you can also access these data programmatically with the `.projection()` method:

```
// Display the projection of band 0
print('Inspect the projection of band 0:', image.select(0).projection());
```

6. Note that the projection can differ by band, which is why it's good practice to inspect the projection of individual image bands.

7. (If you call `.projection()` on an image for which the projection differs by band, you'll get an error.) Exchange the NAIP imagery with the Planet SkySat MultiSpectral image collection, and note that the error occurs because the 'P' band has a different pixel size than the others.

8. Explore the `ee.Projection` docs to learn about useful methods offered by the `Projection` object. To play with projections offline, try this tool.

1.1.3 Digital Image Visualization and Stretching

You've learned about how an image stores pixel data in each band as digital numbers (DNs) and how the pixels are organized spatially. When you add an image to the map, Earth Engine handles the spatial display for you by recognizing the projection and putting all the pixels in the right place. However, you must specify how to stretch the DNs to make an 8-bit display image (e.g., the `min` and `max` visualization parameters). Specifying `min` and `max` applies (where DN' is the displayed value):

$$DN' = \frac{255(DN - min)}{(max - min)}$$

1. To apply a gamma correction (DN' = DN_{γ}), use:

```
// Display gamma stretches of the input image.
Map.addLayer(image.visualize({gamma: 0.5}), {}, 'gamma = 0.5');
Map.addLayer(image.visualize({gamma: 1.5}), {}, 'gamma = 1.5');
```

   Note that gamma is supplied as an argument to image.visualize() so that you can click on the map to see the difference in pixel values (try it!). It's possible to specify `gamma`, `min`, and `max` to achieve other unique visualizations.

2. To apply a histogram equalization stretch, use the `sldStyle()` method

```
// Define a RasterSymbolizer element with '_enhance_' for a placeholder.
  var histogram_sld =
    '<RasterSymbolizer>' +
      '<ContrastEnhancement><Histogram/></ContrastEnhancement>' +
      '<ChannelSelection>' +
        '<RedChannel>' +
          '<SourceChannelName>R</SourceChannelName>' +
        '</RedChannel>' +
        '<GreenChannel>' +
          '<SourceChannelName>G</SourceChannelName>' +
        '</GreenChannel>' +
        '<BlueChannel>' +
          '<SourceChannelName>B</SourceChannelName>' +
        '</BlueChannel>' +
      '</ChannelSelection>' +
    '</RasterSymbolizer>';

  // Display the image with a histogram equalization stretch.
  Map.addLayer(image.sldStyle(histogram_sld), {}, 'Equalized');
```

   The `sldStyle()` method requires image statistics to be computed in a region (to determine the histogram).

1.2 Spatial Resolution

In the present context, spatial resolution often means pixel size. In practice, spatial resolution depends on the projection of the sensor's instantaneous field of view (IFOV) on the ground and how a set of radiometric measurements are resampled into a regular grid. To see the difference in spatial resolution resulting from different sensors, let's visualize data at different scales from different sensors.

1.2.1 MODIS

There are two Moderate Resolution Imaging Spectro-Radiometers (MODIS) aboard the Terra and Aqua satellites. Different MODIS bands produce data at different spatial resolutions. For the visible bands, the lowest common resolution is 500 meters (red and NIR are 250 meters). Data from the MODIS platforms are used to produce a large number of data sets having daily, weekly, 16-day, monthly, and annual data sets. Outside this lab, you can find a list of MODIS land products here.

1. Search for 'MYD09GA' and import '*MYD09GA.006 Aqua Surface Reflectance Daily Global 1km and 500m*'. Name the import `myd09`.

2. Zoom the map to San Francisco (SFO) airport:

```
// Define a region of interest as a point at SFO airport.
var sfoPoint = ee.Geometry.Point(-122.3774, 37.6194);

// Center the map at that point.
Map.centerObject(sfoPoint, 16);
```

3. To display a false-color MODIS image, select an image acquired by the Aqua MODIS sensor and display it for SFO:

```
// Get a surface reflectance image from the MODIS MYD09GA collection.
var modisImage = ee.Image(myd09.filterDate('2017-07-01').first());

// Use these MODIS bands for red, green, blue, respectively.
var modisBands = ['sur_refl_b01', 'sur_refl_b04', 'sur_refl_b03'];

// Define visualization parameters for MODIS.
var modisVis = {bands: modisBands, min: 0, max: 3000};

// Add the MODIS image to the map
Map.addLayer(modisImage, modisVis, 'MODIS');
```

4. Note the size of pixels with respect to objects on the ground. (It may help to turn on the satellite basemap to see high-resolution data for comparison.) Print the size of the pixels (in meters) with:

```
// Get the scale of the data from the first band's projection:
var modisScale = modisImage.select('sur_refl_b01')
.projection().nominalScale();

print('MODIS scale:', modisScale);
```

5. Note these `MYD09` data are surface reflectance scaled by 10000 (not TOA reflectance), meaning that clever NASA scientists have done a fancy atmospheric correction for you!

1.2.2 Multispectral Scanners

Multi-spectral scanners were flown aboard Landsats 1-5. (MSS) data have a spatial resolution of 60 meters.

1. Search for 'landsat 5 mss' and import the result called *'USGS Landsat 5 MSS Collection 1 Tier 2 Raw Scenes'*. Name the import `mss`.

2. To visualize MSS data over SFO (for a relatively cloud-free) image, use:

```
// Filter MSS imagery by location, date and cloudiness.
var mssImage = ee.Image(mss
                        .filterBounds(Map.getCenter())
                        .filterDate('2011-05-01',  '2011-10-01')
                        .sort('CLOUD_COVER')
                        //  Get the least cloudy image.
                        .first());

// Display the MSS image as a color-IR composite.
Map.addLayer(mssImage, {bands: ['B3', 'B2', 'B1'], min: 0, max: 200}, 'MSS');
```

3. Check the scale (in meters) as before:

```
// Get the scale of the MSS data from its projection:
var mssScale = mssImage.select('B1').projection().nominalScale();
print('MSS scale:', mssScale);
```

1.2.3 Thematic Mapper (TM)

The Thematic Mapper (TM) was flown aboard Landsats 4-5. (It was succeeded by the Enhanced Thematic Mapper (ETM+) aboard Landsat 7 and the Operational Land Imager (OLI) / Thermal Infrared Sensor (TIRS) sensors aboard Landsat 8.) TM data have a spatial resolution of 30 meters.

1. Search for 'landsat 5 toa' and import the first result (which should be '*USGS Landsat 5 TM Collection 1 Tier 1 TOA Reflectance*'. Name the import `tm`.

2. To visualize TM data over SFO, for approximately the same time as the MODIS image, use:

```
// Filter TM imagery by location, date and cloudiness.
var tmImage = ee.Image(tm
            .filterBounds(Map.getCenter())
            .filterDate('2011-05-01', '2011-10-01')
            .sort('CLOUD_COVER')
            .first());

// Display the TM image as a color-IR composite.
Map.addLayer(tmImage, {bands: ['B4', 'B3', 'B2'], min: 0, max: 0.4}, 'TM');
```

3. For some hints about why the TM data is not the same date as the MSS data, see this page.

4. Check the scale (in meters) as previously:

```
// Get the scale of the TM data from its projection:
var tmScale = tmImage.select('B1').projection().nominalScale();
print('TM scale:', tmScale);
```

Question 1: By assigning the NIR, red, and green bands in RGB (4-3-2), what features appear bright red in a Landsat 5 image and why?

1.2.4 National Agriculture Imagery Program (NAIP)

The National Agriculture Imagery Program (NAIP) is an effort to acquire imagery over the continental US on a 3-year rotation using airborne sensors. The imagery has a spatial resolution of 1-2 meters.

1. Search for 'naip' and import the data set for *'NAIP: National Agriculture Imagery Program'*. Name the import naip. Since NAIP imagery is distributed as quarters of Digital Ortho Quads at irregular cadence, load everything from the closest year to the examples in its acquisition cycle (2012) over the study area and mosaic() it:

```
// Get NAIP images for the study period and region of interest.

var naipImages = naip.filterDate('2012-01-01', '2012-12-31')
.filterBounds(Map.getCenter());

// Mosaic adjacent images into a single image.
var naipImage = naipImages.mosaic();

// Display the NAIP mosaic as a color-IR composite.
Map.addLayer(naipImage, {bands: ['N', 'R', 'G']}, 'NAIP');
```

2. Check the scale by getting the first image from the mosaic (a mosaic doesn't know what its projection is, since the mosaicked images might all have different projections), getting its projection, and getting its scale (meters):

```
// Get the NAIP resolution from the first image in the mosaic.
var naipScale = ee.Image(naipImages.first()).
              projection().nominalScale();

print('NAIP scale:', naipScale);
```

Question 2: What is the scale of the most recent round of NAIP imagery for the sample area (2018), and how did you determine the scale?

1.3 Spectral Resolution

Spectral resolution refers to the number and width of spectral bands in which the sensor takes measurements. You can think of the width of spectral bands as the wavelength intervals for each band. A sensor that measures radiance in multiple bands is called a *multispectral* sensor (generally 3-10 bands), while a sensor with many bands (possibly hundreds) is called a *hyperspectral* sensor (these are not hard and fast definitions). For example, compare the multi-spectral OLI aboard Landsat 8 to Hyperion, a hyperspectral sensor aboard the EO-1 satellite.

A figure representing common optical sensors and their spectral resolution can be viewed below (image source):

There is an easy way to check the number of bands in Earth Engine, but no way to get an understanding of the relative *spectral response* of the bands, where spectral response is a function measured in the laboratory to characterize the detector.

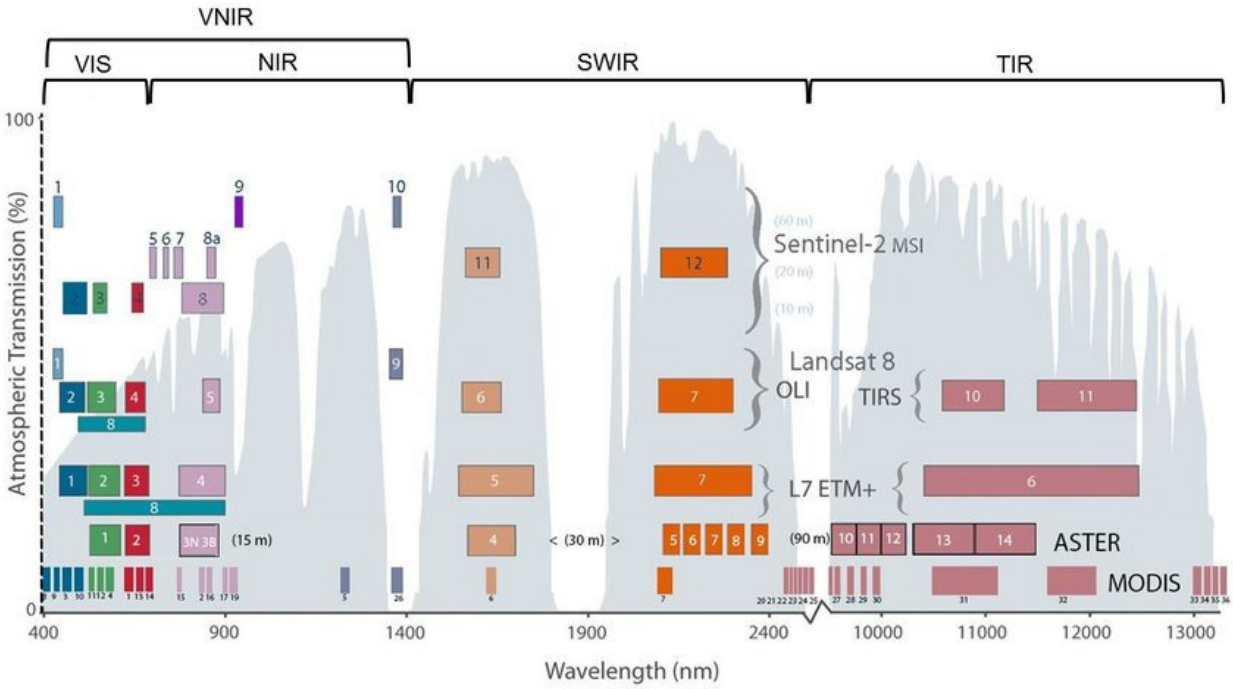


Figure 4: Common Optical Sensors and their Spectral Resolution

1. To see the number of bands in an image, use:

```
// Get the MODIS band names as a List
var modisBands = modisImage.bandNames();

// Print the list.
print('MODIS bands:', modisBands);

// Print the length of the list.
print('Length of the bands list:', modisBands.length());
```

2. Note that only some of those bands contain radiometric data. Lots of them have other information, like quality control data. So the band listing isn't necessarily an indicator of spectral resolution, but can inform your investigation of the spectral resolution of the dataset. Try printing the bands from some of the other sensors to get a sense of spectral resolution.

Question 3.1: What is the spectral resolution of the MODIS instrument, and how did you determine it?

Question 3.2: Investigate the bands available for the USDA NASS Cropland Data Layers (CDL). What does the band information for the CDL represent? Which band(s) would you select if you were interested in evaluating the extent of pasture areas in the US?

1.4 Temporal Resolution

Temporal resolution refers to the *revisit time*, or temporal *cadence* of a particular sensor's image stream. Think of this as the frequency of pixels in a time series at a given location.

1.4.1 MODIS

MODIS (either Terra or Aqua) produces imagery at approximately a daily cadence. To see the time series of images at a location, you can `print()` the `ImageCollection`, filtered to your area and date range of interest. For example, to see the MODIS images in 2011:

```
// Filter the MODIS mosaics to one year.
var modisSeries = myd09.filterDate('2011-01-01', '2011-12-31');

// Print the filtered  MODIS ImageCollection.
print('MODIS series:', modisSeries);
```

Expand the `features` property of the printed `ImageCollection` to see a `List` of all the images in the collection. Observe that the date of each image is part of the filename. Note the daily cadence. Observe that each MODIS image is a global mosaic, so there's no need to filter by location.

1.4.2 Landsat

Landsats (5 and later) produce imagery at 16-day cadence. TM and MSS are on the same satellite (Landsat 5), so it suffices to print the TM series to see the temporal resolution. Unlike MODIS, data from these sensors is produced on a scene basis, so to see a time series, it's necessary to filter by location in addition to time:

```
 // Filter to get a year's worth of TM scenes.
var tmSeries = tm
.filterBounds(Map.getCenter())
.filterDate('2011-01-01', '2011-12-31');

// Print the filtered TM ImageCollection.
print('TM series:', tmSeries);
```

1. Again expand the `features` property of the printed `ImageCollection`. Note that a careful parsing of the TM image IDs indicates the day of year (DOY) on which the image was collected. A slightly more cumbersome method involves expanding each Image in the list, expanding its properties and looking for the 'DATE_ACQUIRED' property.

2. To make this into a nicer list of dates, map() a function over the ImageCollection. First define a function to get a Date from the metadata of each image, using the system properties:

```
var getDate = function(image) {
// Note that you need to cast the argument
var time = ee.Image(image).get('system:time_start');
// Return the time (in milliseconds since Jan 1, 1970) as a Date
return ee.Date(time);
};
```

3. Turn the `ImageCollection` into a `List` andmap() the function over it:

```
var dates = tmSeries.toList(100).map(getDate);
```

4. Print the result:

```
print(dates);
```

Question 4 What is the temporal resolution of the Sentinel-2 satellites? How can you determine this from within GEE?

1.5 Radiometric Resolution

Radiometric resolution refers to the ability of an imaging system to record many levels of brightness: *coarse* radiometric resolution would record a scene with only a few brightness levels, whereas *fine* radiometric resolution would record the same scene using many levels of brightness. Some also consider radiometric resolution to refer to the *precision* of the sensing, or the level of *quantization*.

Radiometric resolution is determined from the minimum radiance to which the detector is sensitive (Lmin), the maximum radiance at which the sensor saturates (Lmax), and the number of bits used to store the DNs (Q):

$$\text{Radiometric resolution} = \frac{(L_{max} - L_{min})}{2^Q}$$

It might be possible to dig around in the metadata to find values for Lmin and Lmax, but computing radiometric resolution is generally not necessary unless you're studying phenomena that are distinguished by very subtle changes in radiance.

1.6 Resampling and ReProjection

Earth Engine makes every effort to handle projection and scale so that you don't have to. However, there are occasions where an understanding of projections is important to get the output you need. As an example, it's time to demystify the reproject() calls in the previous examples. Earth Engine requests inputs to your computations in the projection and scale of the output. The map attached to the playground has a Maps Mercator projection.

The scale is determined from the map's zoom level. When you add something to this map, Earth Engine secretly reprojects the input data to Mercator, resampling (with nearest neighbor) to screen resolution pixels based on the map's zoom level, then does all the computations with the reprojected, resampled imagery. In the previous examples, the reproject() calls force the computations to be done at the resolution of the input pixels: 1 meter.

1. Re-run the edge detection code with and without the reprojection (Comment out all previous `Map.addLayer()` calls except for the original one)

```
// Zoom all the way in.
Map.centerObject(point, 21);
// Display edges computed on a reprojected image.
Map.addLayer(image.convolve(laplacianKernel), {min: 0, max: 255},
       'Edges with little screen pixels');
// Display edges computed on the image at native resolution.
Map.addLayer(edges, {min: 0, max: 255},
       'Edges with 1 meter pixels');
```

What's happening here is that the projection specified in `reproject()` propagates backwards to the input, forcing all the computations to be performed in that projection. If you don't specify, the computations are performed in the projection and scale of the map (Mercator) at screen resolution.

2. You can control how Earth Engine resamples the input with `resample()`. By default, all resampling is done with the nearest neighbor. To change that, call `resample()` on the *inputs*. Compare the input image, resampled to screen resolution with a bilinear and bicubic resampling:

```
// Resample the image with bilinear instead of the nearest neighbor.
var bilinearResampled = image.resample('bilinear');
Map.addLayer(bilinearResampled, {}, 'input image, bilinear resampling');

// Resample the image with bicubic instead of the nearest neighbor.
var bicubicResampled = image.resample('bicubic');
Map.addLayer(bicubicResampled, {}, 'input image, bicubic resampling');
```

3. Try zooming in and out, comparing to the input image resampled with the nearest with nearest neighbor (i.e. without `resample()` called on it).

   You should rarely, if ever, have to use `reproject()` and `resample()`. Do not use `reproject()` or `resample()` unless necessary. They are only used here for demonstration purposes.

1.7 Additional Exercises

Now that we have some familiarity with higher quality images, lets look at a few from the (broken) Landsat 7 satellite. Using your downloading skills, now select an image that contains the Blacksburg area with minimal cloud cover from Landsat 7 (for now, using the Collection 1 Tier 1 calibrated top-of-atmosphere (TOA) reflectance data product). Look at the image.

Question 5: What is the obvious (hint: post-2003) problem with the Landsat 7 image? What is the nature of that problem and what have some researchers done to try to correct it? (please research online in addition to using what you have learned in class/from the book)

Question 6: Name three major changes you can view in the Blacksburg Area in the last decade using any of the above imagery (and state the source).

Conduct a search to compare the technical characteristics of the following sensors:

(i) MODIS (NASA) versus Sentinel (ESA), and

(ii) AVHRR (NASA) versus IRS-P6 (or choose another Indian Remote Sensing satellite)

Question 7: Based on the characteristics you describe, for which applications is one sensor likely to be more suitable than the other ones?

Note: when using the internet to answer this question, be sure to cite your sources and ensure that you are obtaining information from an official, reputable source!

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, September 8. All students who have been attending class have already been enrolled in Gradescope, although if for some reason you need to sign up again, the access code for our course is `6PEW3W`.