PyNotes GEE

Andres Patrignani

March 18, 2024

Table of contents

Introduction

"PyNotes GEE" is a guide for agricultural professionals, environmental scientists, graduate students, and data enthusiasts looking to leverage the power of Python in conjunction with Google Earth Engine (GEE) to make better data-driven decisions, and understand and manage our planet's agricultural resources.

Google Earth Engine is an advanced cloud-based platform designed for petabyte-scale analysis of geospatial data, particularly satellite imagery. It provides a comprehensive set of tools for analyzing and visualizing data, facilitating the extraction of meaningful insights from the Earth's surface over time. The powerful computing infrastructure behind GEE allows researchers to execute complex geospatial analyses that would be computationally intensive or infeasible on conventional laptops and desktop computers. Through its accessible interface, GEE enables automated processing pipelines using Python.

Through a series of short, practical, and step-by-step tutorials in the form of Jupyter notebooks, this compendium provides a comprehensive toolkit describing common techniques for extracting, processing, and analyzing Earth data using Python libraries like Xarray, Numpy, Pandas, Matplotlib, GeoPandas. The content is aimed at learners that are familiar with the Python programming language, but are just getting started with Google Earth Engine.

If you find any mistakes in the code or have suggestions for other tutorials, please create an issue in the Github repository

Getting Started with Google Earth Engine (GEE)

The Google Earth Engine platform can be used with two programming languages: Javascript and Python. This entire series of tutorials is based solely in the Python programming language.

First and foremost, here are two relevant links to the Google Earth Engine official documentation:

- Main Google Earth Engine website
- Developer Guides

Python environemnt

If you do not have Python installed in you machine, download the Anaconda package, which includes a ton of curated Python libraries for data science.

Other required packages

The tutorials were developed with the Anaconda package in mind, however, some tutorial make use of additional libraries. Use the command `pip install <package_name>` to add the following packages:

- **folium**: Library for creating interactive maps.
- **rasterio**: Library for raster data analysis.
- **geopandas**: Extends the pandas library to allow spatial operations particularly for vector data.

You can try to install the packages in one go using: `pip install folium rasterio geopandas`

GEE Account Setup:

1. **Create a Google Account**: If you don't have one already, sign up for a Google Account.
2. **Join Google Earth Engine**: Visit the Google Earth Engine website and register for commercial or non-commercial use (free for academic and research use).
3. **Project Creation**: Once your access is approved, create a new project in the Google Developers Console.

Authentication:

1. **Install the Earth Engine Python API**: Use `pip install earthengine-api` to install the Earth Engine Python API.
2. Open a new Jupyter notebook and import the Google Earth Engine module using `import ee`. We will do this at the beginning of each tutorial.
3. **Authenticate with Earth Engine**: Run `ee.Authenticate()` and follow the instructions to authenticate your account.

Other resources:

awesome-gee-community-catalog: Community-driven catalog of GEE datasets.

geemap: This is an outstanding and comprehensive set of tutorials for those that want a ready-to-use, well-documented, and hassle-free module with tons of videos.

Leafmap: A library built on `folium` for spatial analysis and interactive mapping int he notebook.

cartoee: Library for creating publication quality figures.

1 Basic concepts

In geospatial data analysis, data is represented in two primary forms: vector and raster. Both formats are used to capture, store, and visualize geographical information, but they do so in fundamentally different ways.

1.1 Basic concepts of geographic information systems

Vector Data: represents geographic features as discrete points, lines, and polygons. This format is used to capture details with precise boundaries and locations. For example:

- **Points** could represent locations such as weather stations, wells, or trees.
- **Lines** might delineate features like rivers, roads, or railway tracks.
- **Polygons** are used to define areas such as lakes, watersheds, or agricultural fields.

Vector data is advantageous for mapping features that have clear boundaries and for tasks that require precise measurements (like distance, perimeter, or area).

Raster Data: Raster data, on the other hand, is a grid of cells (or pixels), where each cell contains a value representing information, such as temperature, elevation, or land cover. The raster format is effective for representing continuous phenomena. Raster data is often used in: - Satellite imagery, where each pixel has spectral data values. - Digital elevation models (DEMs), where each pixel represents ground elevation. - Thematic maps, such as precipitation or land use maps, where each pixel's value corresponds to a specific category or quantity.

Raster data is especially useful for analyzing spatial variations across a region and for modeling environmental and earth surface processes.

1.2 GeoJSON

GeoJSON is a light-weight and widely used open standard format designed for representing geographical features. It is based on JSON (JavaScript Object Notation), which is very much like a Python dictionary, making it easy to read and parse by both humans and machines.

A GeoJSON supports various geometric types such as Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Each geometry can also contain

properties in the form of key-value pairs, allowing for the addition non-spatial attributes associated with the geographic features.

The following example is a Feature Collection showing the GeoJSON data obtained from GEE for two counties. The keys on the very left (`columns`, `features`, `id`, `properties`, `type`, and `version`) are keys of the Feature Collection. Note that the `features` key has a list of dictionaries as its value. These two dictinaries contain the information for each county. Since each county is a dictionary, then they also have key-value pairs, like `geometry`, `id`, `properties`, and `type`. This information is repeated in each item. If you look further, you will notice that even the `geometry` key has another dictionary inside, which contains the `coordinates` and the `type` of the geometry.

Certainly there are multiple levels and quite a bit of nesting, but the information has a clear, organized, and human-readable structure.

> **Tip**
>
> The secret to understand GeoJSON objects is to pay attention to:
>
> 1) indentation,
> 2) the opening `{` and closing `}` curly braces,
> 3) opening `[` and closing `]` square brackets, and
> 4) the commas `,` that separate individual structures.

The `.getInfo()` method in GEE allows you to print this information for any object in the notebook. You can also use the Pretty Print (`pprint`) module in Python to ensure the data is displayed clearly, like the example below. This tutorial will help you get started.

```
{'columns': {'ALAND': 'Long',
             'AWATER': 'Long',
             'COUNTYFP': 'String',
             'NAME': 'String',
             'STATEFP': 'String',
             'system:index': 'String'},
 'features': [{'geometry': {'coordinates': [[[-100.9542311520017,36.57269908219634],
                                             [-100.95409647655981,36.557977388454546],
                                             [-100.95409647655981,36.557842673053706],
                                             ...]],
                            'type': 'Polygon'},
               'id': '00000000000000000515',
               'properties': {'ALAND': 4700042738,
                              'AWATER': 7347161,
                              'COUNTYFP': '007',
                              'NAME': 'Beaver',
```

```
                            'NAMELSAD': 'Beaver County',
                            'STATEFP': '40'},
              'type': 'Feature'},
             {'geometry': {'coordinates': [[[-100.0038723945186,36.75510474601957],
                                            [-100.00382751270251,36.75353381242449],
                                            [-100.00382751270251,36.75223223032575],
                                            ...]],
                           'type': 'Polygon'},
              'id': '00000000000000000957',
              'properties': {'ALAND': 2691041230,
                             'AWATER': 5259447,
                             'COUNTYFP': '059',
                             'NAME': 'Harper',
                             'NAMELSAD': 'Harper County',
                             'STATEFP': '40'},
              'type': 'Feature'}],
 'id': 'TIGER/2016/Counties',
 'properties': {'date_range': [1451606400000, 1483315200000],
                'description': 'The United States Census Bureau TIGER dataset '
                               'contains the 2016 boundaries\n'
                               'for primary legal divisions of US states...',
                'period': 0,
                'title': 'TIGER: US Census Counties 2016'},
 'type': 'FeatureCollection',
 'version': 1566851207937615}
```


1.3 Google Earth building blocks

Google Earth Engine (GEE) is a powerful platform for analyzing geospatial data at scale, providing a vast library of satellite imagery and geospatial datasets. The fundamental data structures of GEE are: geometries, features, feature collections, images, and image collections. Understanding these structures is key to effectively utilizing GEE for any geospatial data analysis task.

1.3.1 Geometries

Geometries represent the simplest form of spatial data in GEE, describing points or shapes in geographic space. They can be points, lines, polygons, or even more complex shapes defining areas (e.g., a rectangle, a watershed, a county), routes, or specific locations on Earth's surface.

1.3.2 Features

A Feature in GEE is a geometry with associated properties. These properties can be metadata or attributes related to the geometry, such as the name of a location, its population, a label, or any other characteristic. Features combine spatial and descriptive information, making them useful for detailed geospatial analyses.

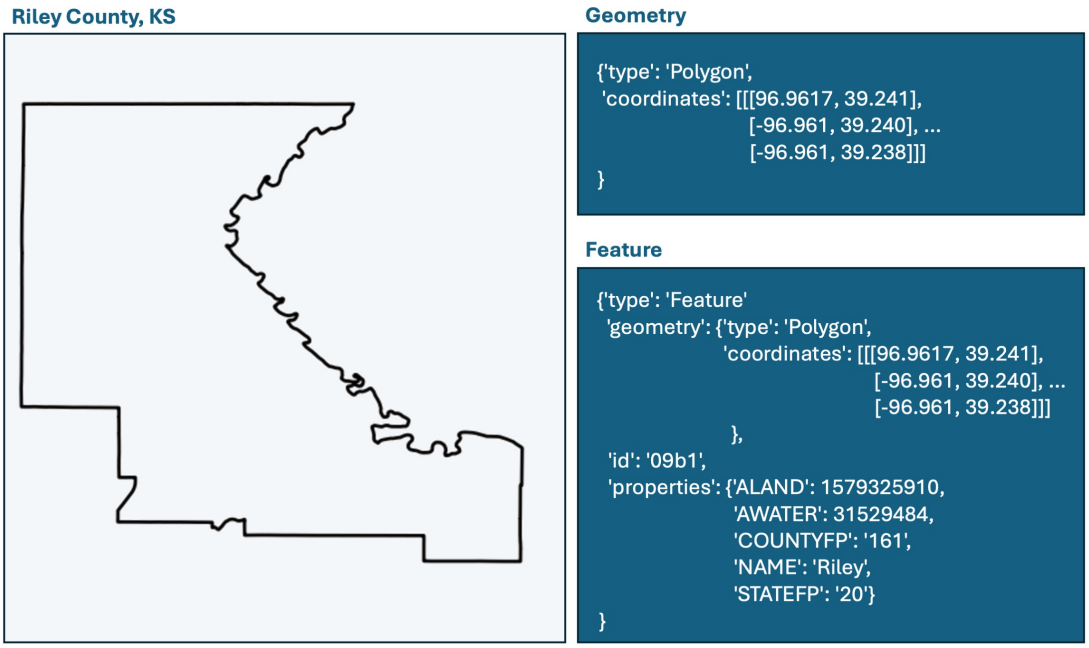


Figure 1.1: Geometry and Feature

1.3.3 Feature Collections

A Feature Collection is a group of features aggregated into a single data structure. This collection can represent a series of points, such as weather stations, or complex combinations of polygons, such as administrative boundaries, each with their own set of attributes. Feature Collections are instrumental in managing and analyzing related sets of features.

1.3.4 Images

An Image in GEE is a raster data structure, representing Earth data in grid format where each cell has a value. These images can depict various types of data, such as satellite imagery, temperature maps, elevation data, or derived indices (e.g., NDVI for vegetation analysis). An

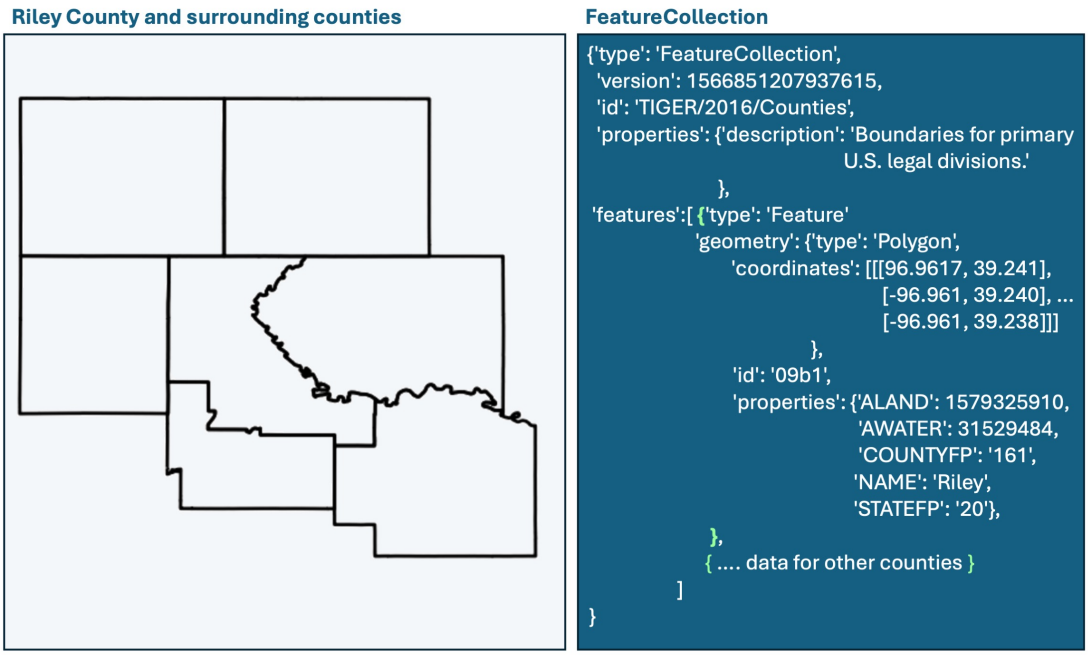


Figure 1.2: FeatureCollection

image may contain multiple bands, with each band representing a different dataset or time snapshot.

1.3.5 Image Collections

An Image Collection is a series of images grouped together, typically representing the same area over time. This could be a sequence of satellite images capturing changes throughout the seasons or years, or a collection of derived datasets like monthly precipitation maps. Image Collections enable temporal analyses and change detection studies over large geographic areas.

1.3.6 Commonalities and Differences

- **Spatial Representation**: Geometries and Features represent vector data, while Images represent raster data. Feature Collections aggregate vector data, and Image Collections aggregate raster data.
- **Data Complexity**: Geometries are the simplest, defining only shapes. Features add descriptive data to geometries. Images and Image Collections introduce the complexity of raster data, allowing for detailed analysis over discrete units of space and time

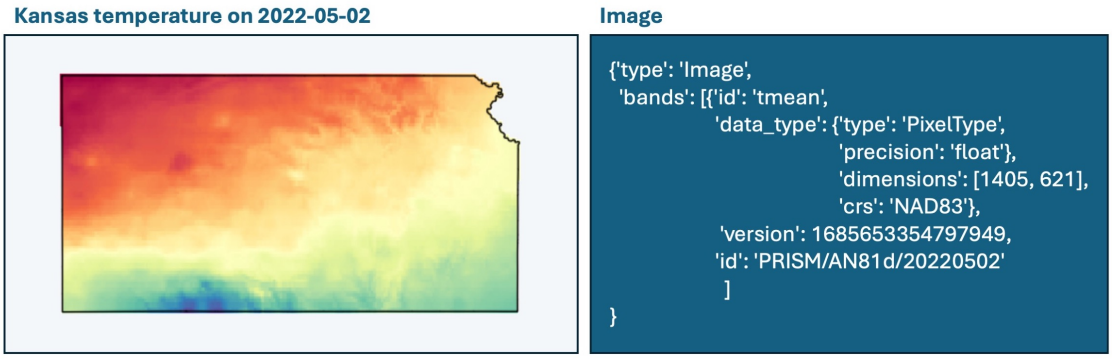


Figure 1.3: Image

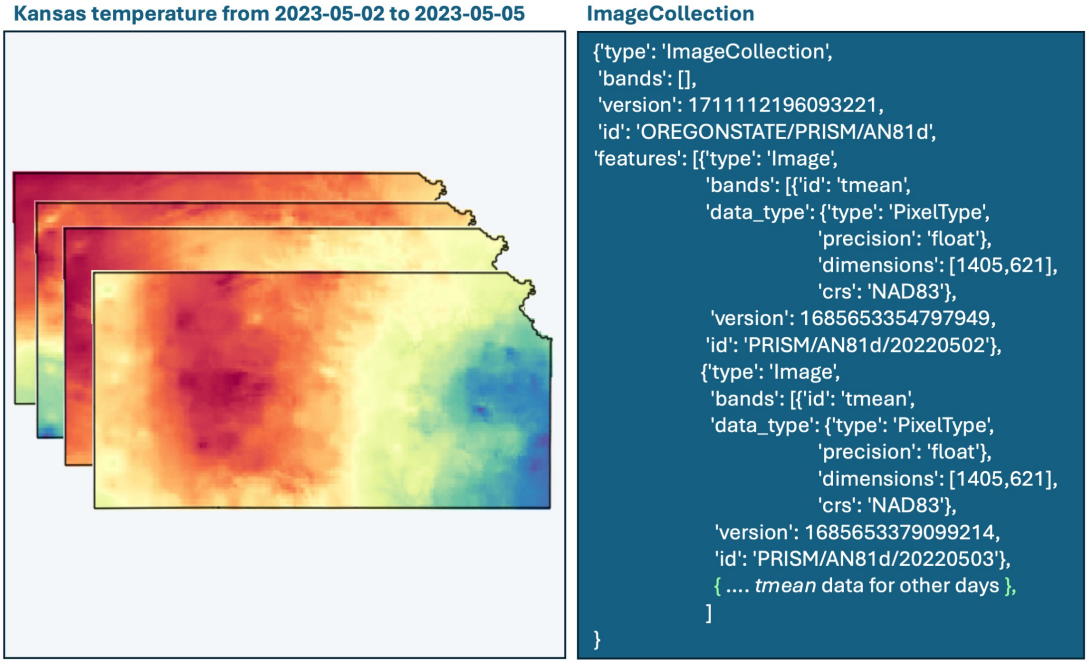


Figure 1.4: ImageCollection

- **Analysis Capabilities**: Features and Feature Collections are ideal for analyses that require descriptive data alongside spatial data. Images and Image Collections are suited for pixel-based analyses and time-series studies.

1.3.7 Summary

Let's summarize these concepts:

- **Geometries**: Basic shapes (point, line, polygon).
- **Features**: Shapes with associated properties (metadata) like names and unique identifiers.
- **Feature Collections**: Aggregated features.
- **Images**: Grid of values (raster), single band or multiple bands.
- **Image Collections**: Series of related images over time.

2 Colormaps

Colormaps are a common theme across all the notebooks. After all, colormaps are essential tools in data visualization by enabling the interpretation of numerical data through color.

A colormap is sequence of colors, where each color corresponds to a particular data value or range of values. By assigning specific colors to different data ranges, colormaps can highlight variations, patterns, and anomalies in spatial data. For example, in visualizing soil moisture, a colormap might transition from dark red (dry) to white (average moisture) to dark blue (wet) to illustrate soil moisture levels.

The structure of a colormap is often defined in terms of RGB (Red, Green, Blue), RGBA (Red, Green, Blue, Alpha), or HEX (hexadecimal) color codes. The alpha channel represents the transparency or opacity of the color.

To learn more about colormaps, I suggest checking the Matplotlib library

In this tutorial we will learn how to:

- Access predefined colormap from the Matplotlib library
- Convert colors from the RGB to HEX system
- Convert custom color palettes from the HEX to RGB system

> **Note**
>
> Visualization colormaps in Google Earth Engine need to be provided using the HEX system. To leverage existing colormaps in libraries like Matplotlib, we need to convert them or to create colormaps from scratch. Fortunately, Matplotlib provides the tools for these operations.

```
# Import modules
import numpy as np
from matplotlib import colors, colormaps
```

2.1 Pre-defined colormaps

Let's explore how to get a colormap, how to visualize them, how to reverse a colormap, and how to get a reduced version with fewer colors.

```
# Divergent colormap good for representing soil moisture conditions
colormaps.get_cmap('RdBu') # Red-Blue
```

```
# Increasing colormap good for representing sand or soil organic carbon
colormaps.get_cmap('YlOrBr') # Yellow-Orange-Brown
```

```
# Monochrome colormaps
colormaps.get_cmap('Greens') # Can also try Reds, Blues
```

```
# Discrete colormap good for representing categorical variables,
# like soil textural classes.
colormaps.get_cmap('Set1')
```

```
# Reverse a colormap with "_r" (run code again removing "_r")
colormaps.get_cmap('Spectral_r')
```

```
# Get number of colors in colormap
print(colormaps.get_cmap('Spectral').N)
```

```
256
```

```
# Resample to select fewer colors
colormaps.get_cmap('Spectral').resampled(10)
```

```
# Access single color

# First save the colormap into a variable
cmap = colormaps.get_cmap('Spectral').resampled(10)

# Explore cmap data type (note that this is not a list)
print(type(cmap))

# Access the first color
print(cmap(0))

# Access the fifth color
print(cmap(4))

# Access the last color
print(cmap(cmap.N))
```

```
<class 'matplotlib.colors.LinearSegmentedColormap'>
(0.6196078431372549, 0.00392156862745098, 0.25882352941176473, 1.0)
(0.9978213507625272, 0.9324618736383442, 0.6357298474945533, 1.0)
(0.3686274509803922, 0.30980392156862746, 0.6352941176470588, 1.0)
```

Note that the outputs of the colormap are on a scale of 0 to 1 rather than on a scale 0 to 255. Representing RGB colors using the 0 to 1 scale enables a more precise specification of color values, simpler calculations for gradients and blending of colors, compatibility with libraries, and a more intuitive represention of the scale of a given color (easier to understand that 0 to 1 than 0 to 255).

2.2 RGB to HEX

As mentioned earlier, we need to pass colormaps in the HEX system to GEE. Let's practice this.

```
# Get colormap from Matplotlib
rgb_cmap = colormaps.get_cmap('magma')

# Define number of colors
n = 7 # Use rgb_cmap.N for all the colors

# Split the cmap "n" colors. Here we create the index values
rgb_index = np.linspace(0, rgb_cmap.N-1, n).astype(int)

# Create a list of HEX colors
hex_cmap = [colors.rgb2hex(rgb_cmap(k)) for k in rgb_index]

print(hex_cmap)
```

```
['#000004', '#2c115f', '#721f81', '#b5367a', '#f1605d', '#feae77', '#fcfdbf']
```

2.3 Create a helper function

This will be handy to re-use our code in other tutorials or projects.

```
# Define function to retrieve colormaps
def get_hex_cmap(name,n=10):
    """
```

```
    Function to get list of HEX colors from a Matplotlib colormap.
    """
    rgb_cmap = colormaps.get_cmap(name)
    if n > rgb_cmap.N-1:
        raise ValueError(f"You select {n} colors, but {name} colormap only has {rgb_cmap.N
    else:
        rgb_index = np.linspace(0, rgb_cmap.N-1, n).astype(int)
        hex_cmap = [colors.rgb2hex(rgb_cmap(k)) for k in rgb_index]
        return hex_cmap
```

```
# Test function
get_hex_cmap('Set1', n=7)
```

```
['#e41a1c', '#377eb8', '#4daf4a', '#ff7f00', '#ffff33', '#a65628', '#999999']
```

2.4 HEX to RGB colors

This conversion sometimes becomes necessary for leveraging Python libraries, such as Rasterio, to generate local maps. For example, when consulting GEE documentation to construct a vegetation index map, you will probably encounter colormaps specified in HEX format. This assumes the use of GEE's plotting tools. However, to employ the same colormap with alternative Python libraries that operate in the RGB color system, a conversion from HEX to RGB can be useful.

```
# Paletter of colors for the Enhanced Vegetation Index
hex_palette = ['#FEFEFE','#CE7E45', '#DF923D', '#F1B555', '#FCD163', '#99B718', '#74A901',
               '#66A000', '#529400', '#3E8601', '#207401', '#056201', '#004C00', '#023B01',
               '#012E01', '#011D01', '#011301']

# Use the built-in ListedColormap function to do the conversion
rgb_cmap = colors.ListedColormap(hex_palette)
```

```
# Display our new colormap
rgb_cmap
```

2.5 RGB and HEX in more depth

If you made this far, let's delve into a greater level of detail about the RGB and HEX color systems.

The RGB color space specifies colors using three components: red, green, and blue, each varying from 0 to 255, or occasionally from 0 to 1 as demonstrated by Matplotlib's output.

In contrast, the HEX system employs six-digit hexadecimal numbers preceded by a hash (#), with each digit pair denoting the red, green, and blue components. For example, white is expressed as `(255, 255, 255)` in RGB and `#ffffff` in HEX. Consequently, the RGB system offers 256 intensity levels per component, totaling over 16 million possible colors ($256^3 = 16.7$ million).

Let's look at some examples. Green is depicted as `(0, 255, 0)` or `(0, 1, 0)` in RGB and as `#00ff00` in HEX. Here, `00` signifies the minimum intensity, while `ff` denotes the maximum intensity. So `ff` is equivalent to `255` or `1`. Utilizing ten numeric (0 to 9) and six alphabetic (a through f) characters results in 16 alphanumeric digits. Given that $\sqrt{256} = 16$, two characters are required to represent 256 combinations, with `00` for `0`, `01` for `1`, and so forth until `fe` for `254`, and `ff` for `255`.

```
# Recall our description of the HEX system above.
# Show that the first color (index zero) "#FEFEFE" is (254, 254, 254)

print('Scale 0-1:', np.array(rgb_cmap(0)))
print('Scale 0-255:', (np.array(rgb_cmap(0))*255).astype(int))

# The alpha channel does not have any transparency, so it has a value of 1 or 255
```

```
Scale 0-1: [0.99607843 0.99607843 0.99607843 1.        ]
Scale 0-255: [254 254 254 255]
```

3 Value at a point

Perhaps, one of the simplest and most basic operations in Google Earth Engine is to retrieve information for a single point. Typical examples include elevation, mean annual temperature, and soil properties. In other words, this type of operations is mostly intended at retrieving information that does not change with time.

```
# Import modules
import ee
from datetime import datetime
from pprint import pprint
```

```
# Authenticate
ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

> Tip
>
> While we need to run `ee.Initialize()` everytime we start a notebook or restart the Python kernel, we only need to authenticate once, so a handy tip is to mute the `ee.Initialize()` line with a `#` once you are done. Note that after several hours or days of inactivity you will need to authenticate again.

3.1 Example 1: Elevation

In the following example we will retrieve the elevation for a specific point on Earth.

Product: USGS/SRTMGL1_003

```
# Define geographic coordinates
lat = 39.186512 # This is y
lon = -96.576844 # This is x
```

```
# Convert coordinates into a Point geometry following the x,y notation
point = ee.Geometry.Point([lon, lat])

# Explore point geometry
point.getInfo()
```

```
{'type': 'Point', 'coordinates': [-96.576844, 39.186512]}
```

> **Note**
>
> In this tutorial we will use the `.getInfo()` method to print the underlying data and metadata of multiple objects, so that you can see the output and become more familiar with GEE data structures and the geoJSON format. This information is also valubale for debugging code errors. However, you don't typically want to clutter your notebooks with these long outputs, so we will skip some of these steps in future tutorials.

```
# Load digital elevation model (DEM) from Shuttle Radar Topography Mission (SRTM)
dem = ee.Image('USGS/SRTMGL1_003')
```

```
# Obtain some information about the DEM layer (output is long!)
pprint(dem.getInfo())
```

```
# Retrieve the elevation value: This step will get us close to the answer, but we are not
pprint(dem.sample(point, 1).getInfo())
```

```
{'columns': {'elevation': 'Short'},
 'features': [{'geometry': None,
               'id': '0',
               'properties': {'elevation': 317},
               'type': 'Feature'}],
 'properties': {'band_order': ['elevation']},
 'type': 'FeatureCollection'}
```

```
# Retrieve the elevation value: This step will get us even closer to the answer, but we ar
dem.sample(point, 1).first().getInfo()
```

```
{'type': 'Feature',
```

```
 'geometry': None,
 'id': '0',
 'properties': {'elevation': 317}}
```

```
# Retrieve the elevation value: This step will get us the correct value
elev = dem.sample(point, 1).first().getNumber('elevation').getInfo()
print(f'Elevation: {elev} m')
```

```
Elevation: 317 m
```

3.2 Example 2: Weather variables

Obtain the long-term mean annual air temperature and precipitation for a specific location

Product: For more information about bands and units visit: WorldClim BIO

```
# Define geographic coordinates
lat = 39.186512 # This is y
lon = -96.576844 # This is x

# Convert coordinates into a Point geometry following the x,y notation
point = ee.Geometry.Point(lon, lat)
```

```
# Load WorldClim BIO dataset
dataset = ee.Image('WORLDCLIM/V1/BIO')
```

```
# Access metadata of the product (output is long!)
dataset.getInfo()
```

```
# Get long-term mean annual air temperature
T_mean = dataset.select('bio01').sample(point,1).first().getNumber('bio01').multiply(0.1).
print(f'Mean annual air temperature: {round(T_mean,1)} Celsius')
```

```
Mean annual air temperature: 12.2 Celsius
```

```
# Get long-term annual precipitation
P_annual = dataset.select('bio12').sample(point,1).first().get('bio12').getInfo()
```

```
print(f'Mean annual precipitation: {round(P_annual,1)} mm')
```

```
Mean annual precipitation: 857 mm
```

3.3 Example 3: Reference ET

In this example we will retrieve daily values of reference evapotranspiration for a point.

Product: For more information visit the description in Google Earth Engine of GRIDMET

```
# Define point
point = ee.Geometry.Point([-96.576844, 39.186512])
```

```
# Obtain GRIDMET dataset for a specific period. End date is excluded from the call
start_date = ee.Date('2022-07-01')
end_date = ee.Date('2022-07-02')
day_of_interest = ee.Date('2022-03-15')
dataset = ee.ImageCollection('IDAHO_EPSCOR/GRIDMET').filterDate(start_date, end_date)
```

```
# Information about image collection
pprint(dataset.getInfo())
```

```
# Use the get methog to retrieve a specific property
dataset.get('description').getInfo()
```

```
# Information about first image within the collection (output is long!)
dataset.first().getInfo()
```

```
# Information about feature collection
dataset.first().sample(point,1).getInfo() # Sample at 1 meter resolution
```

```
{'type': 'FeatureCollection',
 'columns': {'bi': 'Float',
  'erc': 'Float',
  'eto': 'Float',
  'etr': 'Float',
  'fm100': 'Float',
  'fm1000': 'Float',
```

```
 'pr': 'Float',
 'rmax': 'Float',
 'rmin': 'Float',
 'sph': 'Float',
 'srad': 'Float',
 'th': 'Float',
 'tmmn': 'Float',
 'tmmx': 'Float',
 'vpd': 'Float',
 'vs': 'Float'},
'properties': {'band_order': ['pr',
  'rmax',
  'rmin',
  'sph',
  'srad',
  'th',
  'tmmn',
  'tmmx',
  'vs',
  'erc',
  'eto',
  'bi',
  'fm100',
  'fm1000',
  'etr',
  'vpd']},
'features': [{'type': 'Feature',
  'geometry': None,
  'id': '0',
  'properties': {'bi': 0,
   'erc': 20,
   'eto': 4.5,
   'etr': 5.300000190734863,
   'fm100': 14.899999618530273,
   'fm1000': 17.799999237060547,
   'pr': 43.70000076293945,
   'rmax': 90.9000015258789,
   'rmin': 60.20000076293945,
   'sph': 0.014340000227093697,
   'srad': 249.39999389648438,
   'th': 114,
   'tmmn': 293.70001220703125,
   'tmmx': 300.1000061035156,
```

```
    'vpd': 0.7300000190734863,
    'vs': 2.799999952316284}}]}
```

```
# Information about feature
eto = dataset.first().sample(point,1).first().getNumber('eto').getInfo()
print(f'The grass reference ET is {eto} mm')
```

```
The grass reference ET is 4.400000095367432 mm
```

Alternative solution: Access the image directly rather than the collection.

```
ee.Image('IDAHO_EPSCOR/GRIDMET/20220701').sample(point,1).first().getNumber('eto').getInfo
```

3.3.0.1 Get dataset timestamps

```
# Obtain the START time of the dataset to check that it matches our request.
# Response is in milliseconds since 1-Jan-1970
dataset.select('eto').first().getNumber('system:time_start').getInfo()
```

```
1656655200000
```

```
# Obtain the END time of the dataset to check that it matches our request.
# Response is in milliseconds since 1-Jan-1970
dataset.select('eto').first().getNumber('system:time_end').getInfo()
```

```
1656741600000
```

```
# Find the datetime of the serial date numbers
# Input in "fromtimestamp()" has to be in seconds, so we divide by 1000
print('Start time:', datetime.fromtimestamp(1656655200000/1000).strftime('%Y-%m-%d %H:%M:%
print('End time:', datetime.fromtimestamp(1656741600000/1000).strftime('%Y-%m-%d %H:%M:%S.
```

```
Start time: 2022-07-01 01:00:00.000000
End time: 2022-07-02 01:00:00.000000
```

3.4 Example 4: Retrieve soil properties for a given location

Product: SoilGrids

Source: https://samapriya.github.io/awesome-gee-community-datasets/projects/isric/

```
# Load the SoilGrids dataset from GEE
soil_grids = ee.Image("projects/soilgrids-isric/sand_mean")
```

```
# Define a point geometry
lat = 37.839154
lon = -99.101594
point = ee.Geometry.Point(lon,lat)
sand = soil_grids.sample(point,250).first().getNumber('sand_0-5cm_mean').multiply(0.1).get
```

```
print(f'The percentage of sand at ({lat},{lon}) is: {round(sand)} %')
```

```
The percentage of sand at (37.839154,-99.101594) is: 53 %
```

4 Value for multiple points

In the previous tutorial we learned how to retrieve a single value for a point, but often times we are interested in querying values for multiple points or locations in space. To illustrate this workflow, we will create a `for` loop based on the previous tutorial.

In this example we will retrieve soil physical properties, including sand and clay content for several stations of the Kansas Mesonet (https://mesonet.k-state.edu). Soil physical properties like sand and clay percentage are easy to measure and can be used to estimate other soil physical properties that are more difficult to measure, like water holding capacity. To compare different gridded products, we will use the SoilGrids and the Polaris datasets.

If you want to learn more about the network and the soil physical properties I recommend reading the following two peer-reviewed articles: Parker et al. (2022) and Patrignani et al. (2020)

```
# Import modules
import ee
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Authenticate
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

4.1 Load GEE datasets

```
# Load the SoilGrids dataset from GEE
soilgrids_sand = ee.Image("projects/soilgrids-isric/sand_mean")
soilgrids_clay = ee.Image("projects/soilgrids-isric/clay_mean")

# Load Polaris dataset
# Note: It's easier to load the image than the ImageCollection.
# Check the 'id' of each feature in the ImageCollection to get the link for the Image
```

```
# Code for ImageCollection: ee.ImageCollection('projects/sat-io/open-datasets/polaris/sand
polaris_sand = ee.Image('projects/sat-io/open-datasets/polaris/sand_mean/sand_0_5')
polaris_clay = ee.Image('projects/sat-io/open-datasets/polaris/clay_mean/clay_0_5')
```

Before requesting data using the loaded GEE products, it is necessary to understand the nature of these datasets:

4.1.0.1 Soil Grids

This dataset has a 250-meter spatial resolution (Poggio et al., 2021) and returns an image, where each band represents a depth for the given variable. After sampling the image, GEE returns a FeatureCollection with a single feature (we the `.first()` function to access this information). Run this line of code to inspect the output:

```
soilgrids_sand.getInfo()
```

4.1.0.2 Polaris

This dataset has a spatial resolution of 30-meters (Chaney et al., 2019). The way we are calling this dataset it returns an Image, where each band represents a depth for a given variable. Again, after sampling the image, GEE returns a FeatureCollection with a single feature (we the `.first()` function to access this information).Run this line of code to inspect the output:

```
polaris_clay.getInfo()
```

Different teams aggregate data using different structures, so before using these or any other products available in GEE is important to read the documentation and inspect the data with a few examples.

4.2 Station data

```
# Create dictionary with station metadata

stations = [
    {'name': 'Ashland Bottoms', 'latitude': 39.125773, 'longitude': -96.63653},
    {'name': 'Belleville 2W', 'latitude': 39.81409, 'longitude': -97.675093},
    {'name': 'Colby', 'latitude': 39.39247, 'longitude': -101.06864},
```

```
    {'name': 'Garden City', 'latitude': 37.99733, 'longitude': -100.81514},
    {'name': 'Gypsum', 'latitude': 38.72522, 'longitude': -97.44415},
    {'name': 'Hutchinson 10SW', 'latitude': 37.93097, 'longitude': -98.02},
    {'name': 'Manhattan', 'latitude': 39.20857, 'longitude': -96.59169},
    {'name': 'Ottawa 2SE', 'latitude': 38.54268, 'longitude': -95.24647},
    {'name': 'Parsons', 'latitude': 37.36875, 'longitude': -95.28771},
    {'name': 'Rossville 2SE', 'latitude': 39.11661, 'longitude': -95.91572},
    {'name': 'Silver Lake 4E', 'latitude': 39.09213, 'longitude': -95.78153},
    {'name': 'Tribune 6NE', 'latitude': 38.53041, 'longitude': -101.66434},
    {'name': 'Woodson', 'latitude': 37.8612, 'longitude': -95.7836}
]
```

4.3 Request soil data for each station

```
# We will retrieve both datasets at 250-meter resolution to match the
# coarser dataset.

# Iterate over each station metadata
for k,station in enumerate(stations):

    # Display current state of the loop
    print(f"Requesting data for {station['name']}")

    # Convert geographic coordinates into a Point geometry
    # following the x,y notation
    point = ee.Geometry.Point([station['longitude'], station['latitude']])

    # Soil Grids: Sample image and then select first and only feature with property value
    stations[k]['sand_soilgrids'] = soilgrids_sand.sample(point, 250).first().getNumber('s
    stations[k]['clay_soilgrids'] = soilgrids_clay.sample(point, 250).first().getNumber('c

    # Polaris: Sample image and then select first and only feature with property value
    stations[k]['sand_polaris'] = polaris_sand.sample(point, 250).first().getNumber('b1').
    stations[k]['clay_polaris'] = polaris_clay.sample(point, 250).first().getNumber('b1').
```

```
Requesting data for Ashland Bottoms
Requesting data for Belleville 2W
```

```
Requesting data for Colby
Requesting data for Garden City
Requesting data for Gypsum
Requesting data for Hutchinson 10SW
Requesting data for Manhattan
Requesting data for Ottawa 2SE
Requesting data for Parsons
Requesting data for Rossville 2SE
Requesting data for Silver Lake 4E
Requesting data for Tribune 6NE
Requesting data for Woodson
```

4.4 Create DataFrame

```
# Convert dictionary into a dataframe for easier visualization
df = pd.DataFrame(stations)
df.head(3)
```

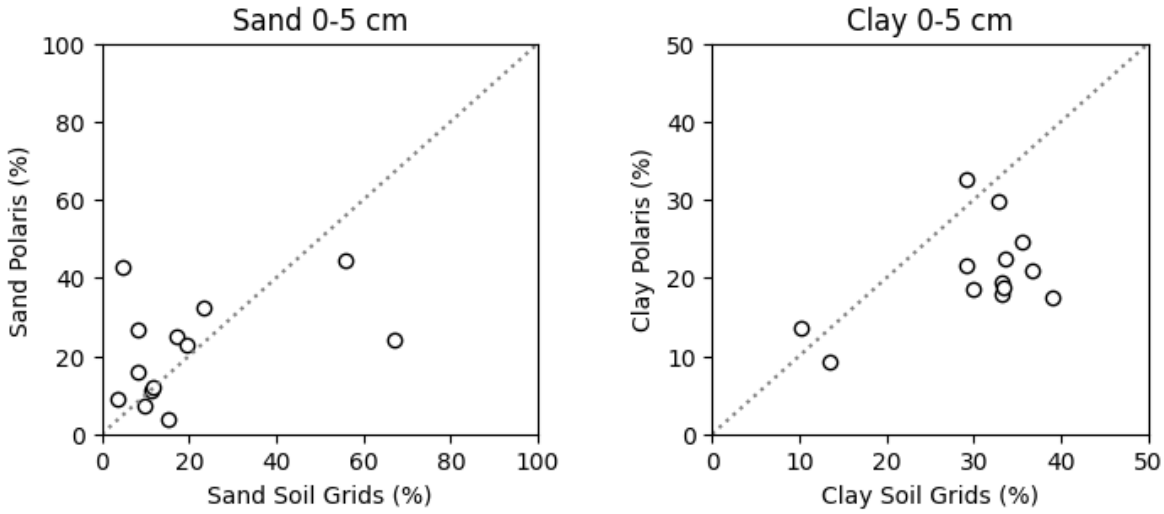
	name	latitude	longitude	sand	clay	sand_soilgrids	clay_soilgrids	sand_polaris
0	Ashland Bottoms	39.125773	-96.636530	15.4	29.2	15.4	29.2	3.831753
1	Belleville 2W	39.814090	-97.675093	9.9	35.6	9.9	35.6	7.110556
2	Colby	39.392470	-101.068640	17.1	33.2	17.1	33.2	25.102043

4.5 Create Figure

Now that we have the data from both datasets, let compare them

```
plt.figure(figsize=(8,3))

plt.subplot(1,2,1)
plt.title('Sand 0-5 cm')
plt.scatter(df['sand_soilgrids'], df['sand_polaris'],facecolor='w',edgecolor='k')
plt.xlabel('Sand Soil Grids (%)')
plt.ylabel('Sand Polaris (%)')
plt.xlim([0,100])
plt.ylim([0,100])
plt.axline((0,0), slope=1, color='grey', linestyle=':')
```

```
plt.subplot(1,2,2)
plt.title('Clay 0-5 cm')
plt.scatter(df['clay_soilgrids'], df['clay_polaris'],facecolor='w',edgecolor='k')
plt.xlabel('Clay Soil Grids (%)')
plt.ylabel('Clay Polaris (%)')
plt.xlim([0,50])
plt.ylim([0,50])
plt.axline((0,0), slope=1, color='grey', linestyle=':')

plt.subplots_adjust(wspace=0.4)
plt.show()
```



4.6 References

- Poggio, L., De Sousa, L. M., Batjes, N. H., Heuvelink, G. B., Kempen, B., Ribeiro, E., & Rossiter, D. (2021). SoilGrids 2.0: producing soil information for the globe with quantified spatial uncertainty. Soil, 7(1), 217-240.
- Parker, N., Kluitenberg, G. J., Redmond, C., & Patrignani, A. (2022). A database of soil physical properties for the Kansas Mesonet. Soil Science Society of America Journal, 86(6), 1495-1508. https://doi.org/10.1002/saj2.20465
- Patrignani, A., Knapp, M., Redmond, C., & Santos, E. (2020). Technical overview of the Kansas Mesonet. Journal of Atmospheric and Oceanic Technology, 37(12), 2167-2183. https://doi.org/10.1175/JTECH-D-19-0214.1

- Chaney, N. W., Minasny, B., Herman, J. D., Nauman, T. W., Brungard, C. W., Morgan, C. L., … & Yimam, Y. (2019). POLARIS soil properties: 30-m probabilistic maps of soil properties over the contiguous United States. Water Resources Research, 55(4), 2916-2938.

5 Time series at a point

```
# Import modules
import ee
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pprint import pprint

# Modules required for example 4
import folium
import clipboard # !pip install clipboard
```

```
# Trigger the authentication flow.
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

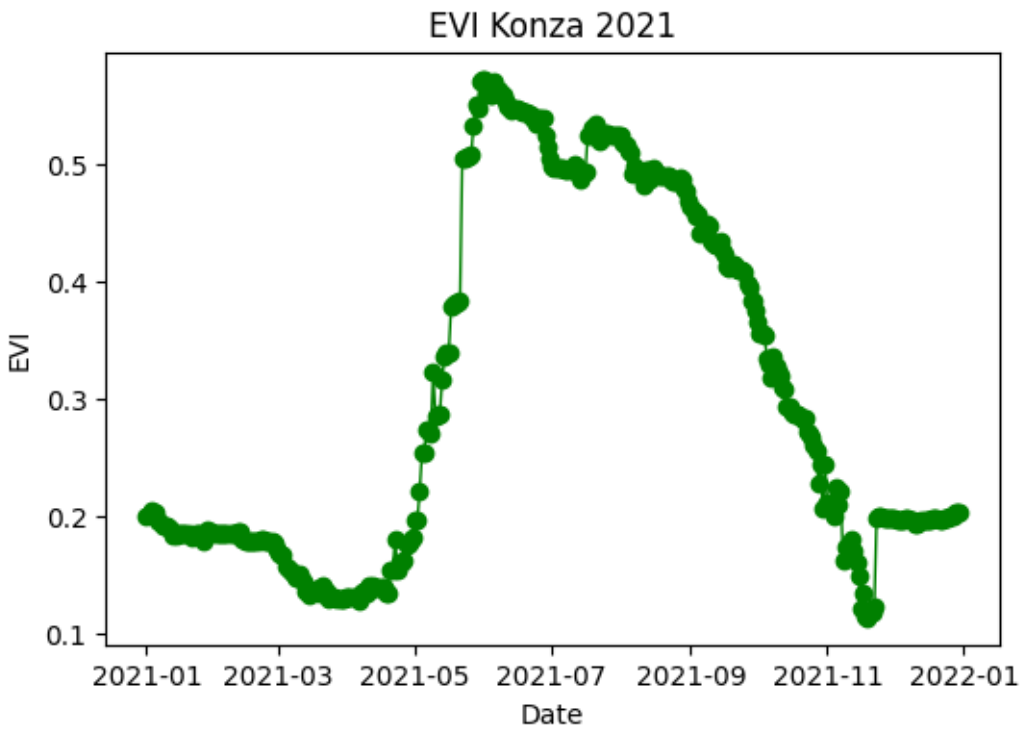
```
# Define function to create dataframe from GEE data
def array_to_df(arr):
    """Function to convert list into dataframe"""
    df = pd.DataFrame(arr[1:])
    df.columns = arr[0]
    df['time'] = pd.to_datetime(df['time'], unit='ms')
    return df
```

5.1 Example 1: Tallgrass prairie vegetation index

Retrive and plot the enhanced vegetation index (EVI) for a point under grassland vegetation at the Konza Prairie

Product: MODIS

```
# Get collection for Modis 16-day
MCD43A4 = ee.ImageCollection('MODIS/MCD43A4_006_EVI').filterDate('2021-01-01','2021-12-31'
EVI = MCD43A4.select('EVI')

# Run this line to explore dataset details (output is long!)
pprint(EVI.getInfo())

# Define point of interest
konza_point = ee.Geometry.Point([-96.556316, 39.084535])

# Get data for region
konza_evi = EVI.getRegion(konza_point, scale=1).getInfo()

# Run this line to inspect retrieved data (output is long!)
pprint(konza_evi)

# Convert array into dataframe
df_konza = array_to_df(konza_evi)

# Save dataframe as a .CSV file
# df.to_csv('modis_evi.csv', index=False)

# Create figure to visualize time series
plt.figure(figsize=(6,4))
plt.title('EVI Konza 2021')
plt.plot(df_konza['time'], df_konza['EVI'], linestyle='-',
         linewidth=1, marker='o', color='green')
plt.xlabel('Date')
plt.ylabel('EVI')
#plt.savefig('evi_figure.png', dpi=300)
plt.show()
```



5.2 Example 2: Drought index

Drought can be represented by a variety of indices, including soil moisture, potential atmospheric demand, days without measurable precipitation, and indices that combine one or more of these variables. The Evaporative Demand Drought Index (EDDI) is inteded to represent the potential for drought (rather than the actual occurrence of drought).

In this exercise we will compare drought conditions for eastern and western Kansas during 2021 and 2022.

Product: GRIDMET DROUGHT

```
# Define locations
eastern_ks = ee.Geometry.Point([-95.317201, 38.588548]) # Near Ottawa, KS
western_ks = ee.Geometry.Point([-101.721117, 38.517258]) # Near Tribune, KS
```

```
# Load EDDI product
gridmet_drought = ee.ImageCollection("GRIDMET/DROUGHT").filterDate('2021-01-01','2022-12-3
eddi = gridmet_drought.select('eddi14d')
```

```
# Get eddie for points
eastern_eddi = eddi.getRegion(eastern_ks, scale=1).getInfo()
western_eddi = eddi.getRegion(western_ks, scale=1).getInfo()
```

```
# Explore output
eastern_eddi[0:3]
```

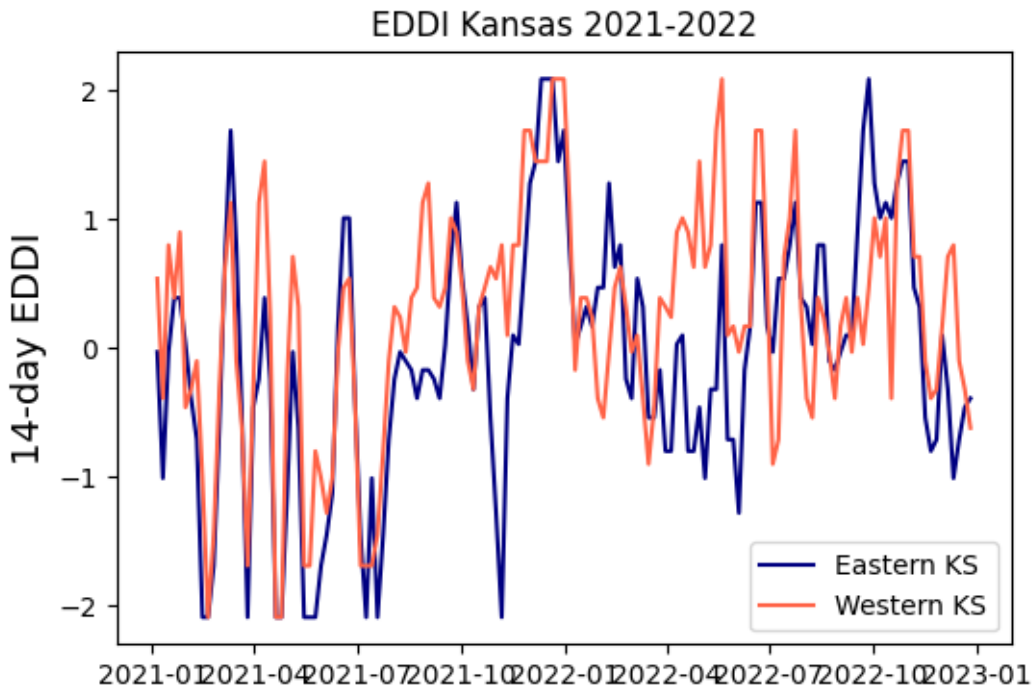
```
[['id', 'longitude', 'latitude', 'time', 'eddi14d'],
 ['20210105',
  -95.31720298383848,
  38.588547875926515,
  1609826400000,
  -0.029999999329447746],
 ['20210110',
  -95.31720298383848,
  38.588547875926515,
  1610258400000,
  -1.0099999904632568]]
```

```
# Create dataframe for each point
df_eastern = array_to_df(eastern_eddi)
df_western = array_to_df(western_eddi)

# Display a few rows
df_eastern.head()

# Save data to a comma-separated value file
# df.to_csv('eddi_14_day.csv', index=False)
```

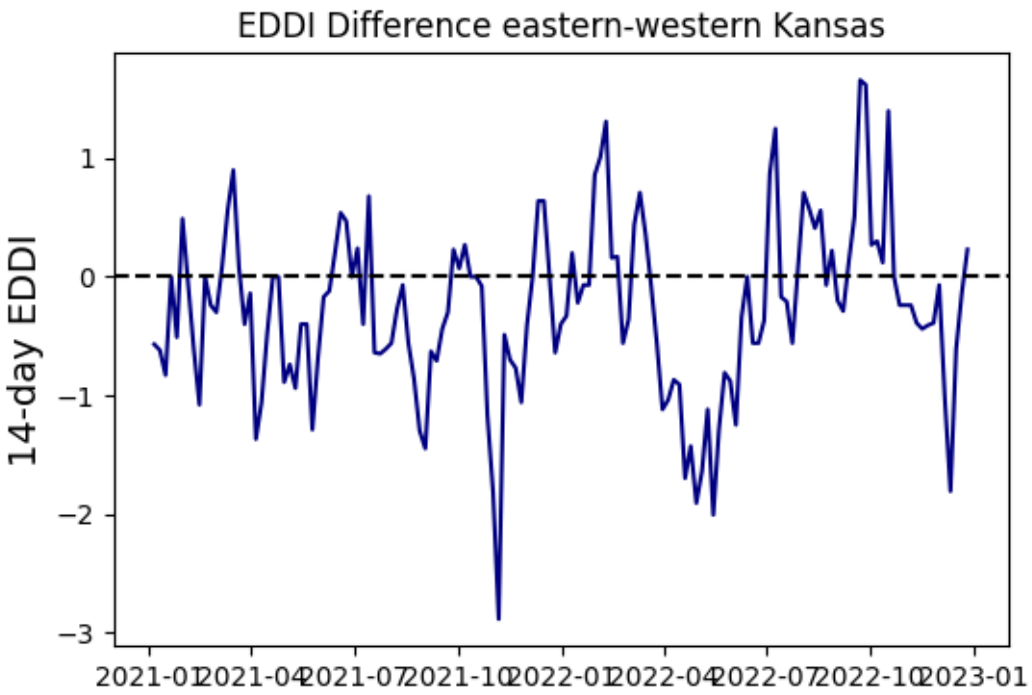
	id	longitude	latitude	time	eddi14d
0	20210105	-95.317203	38.588548	2021-01-05 06:00:00	-0.03
1	20210110	-95.317203	38.588548	2021-01-10 06:00:00	-1.01
2	20210115	-95.317203	38.588548	2021-01-15 06:00:00	-0.03
3	20210120	-95.317203	38.588548	2021-01-20 06:00:00	0.39
4	20210125	-95.317203	38.588548	2021-01-25 06:00:00	0.39

```
# Create figure to compare EDDI for both points
plt.figure(figsize=(6,4))
```

```
plt.title('EDDI Kansas 2021-2022')
plt.plot(df_eastern['time'], df_eastern['eddi14d'], linestyle='-', color='navy', label='Ea
plt.plot(df_western['time'], df_western['eddi14d'], linestyle='-', color='tomato', label='
plt.legend()
plt.ylabel('14-day EDDI', size=14)
plt.show()
```




```
# Compute difference. If negative, that means that drought potential is greater in
# western Kansas

plt.figure(figsize=(6,4))
plt.title('EDDI Difference eastern-western Kansas')
plt.plot(df_eastern['time'], df_eastern['eddi14d']-df_western['eddi14d'], linestyle='-', c
plt.axhline(0, linestyle='--', color='k')
plt.ylabel('14-day EDDI', size=14)
plt.show()
```



5.3 Example 3: Irrigated vs rainfed corn vegetation index

```
# Define points
data = {'latitude': [38.7640, 38.7628, 38.7787, 38.7642, 38.7162, 38.7783],
        'longitude':[-101.8946, -101.8069, -101.6937, -101.9922, -101.8284, -101.9919],
        'irrigated':[True, True, True, False, False, False]}

df = pd.DataFrame(data)
df.head()
```

	latitude	longitude	irrigated
0	38.7640	-101.8946	True
1	38.7628	-101.8069	True
2	38.7787	-101.6937	True
3	38.7642	-101.9922	False
4	38.7162	-101.8284	False

```
# Get product
MOD13Q1 = ee.ImageCollection("MODIS/061/MOD13Q1").filterDate(ee.Date("2022-04-01"),ee.Date
```

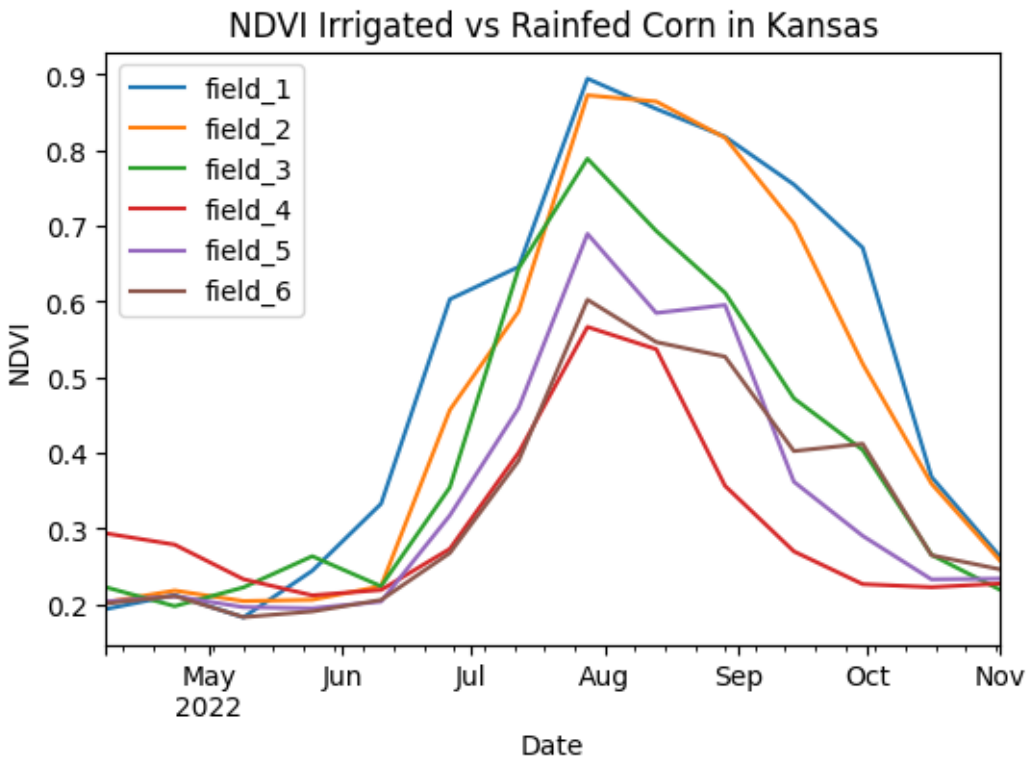
Since in this particular exercise we have multiple locations, one simple solution is to iterate over each point and request the time series of NDVI.

```
# Iterate over each point and retrieve NDVI
ndvi={}

for k, row in df.iterrows():
    point = ee.Geometry.Point(row['longitude'], row['latitude'])
    result = MOD13Q1.select('NDVI').getRegion(point, 0.01).getInfo()
    result_in_colums = np.transpose(result)
    ndvi[f"field_{k+1}"] = result_in_colums[4][1:]
    dates = result_in_colums[0][1:]
```

```
# Create Dataframe with the NDVI data for each field
df_ndvi = pd.DataFrame(ndvi,dtype=float)

# Add dates as index
df_ndvi.index = pd.to_datetime(dates, format='%Y_%m_%d')

# Apply conversion factor
df_ndvi = df_ndvi*0.0001

df_ndvi.head()
```

	field_1	field_2	field_3	field_4	field_5	field_6
2022-04-07	0.1933	0.2024	0.2222	0.2934	0.2032	0.2007
2022-04-23	0.2118	0.2178	0.1972	0.2785	0.2104	0.2107
2022-05-09	0.1824	0.2039	0.2220	0.2328	0.1959	0.1827
2022-05-25	0.2440	0.2056	0.2632	0.2115	0.1943	0.1902
2022-06-10	0.3325	0.2236	0.2235	0.2186	0.2033	0.2054

```
df_ndvi.plot(figsize=(6,4))
plt.title('NDVI Irrigated vs Rainfed Corn in Kansas')
plt.xlabel('Date')
plt.ylabel('NDVI')
plt.show()
```



5.4 Example 4: Interactive selection

In this example we will leverage the interactive functionality of the Folium library and the computer's clipboard to get geographic coordinates with a mouse click and then retrieve time series of a vegetation index using GEE.

```
# Define function to create raster map
# Declare a function (blueprint)
def create_raster(ee_object, vis_params, name):
    """Function that creates a folium raster layer"""
    raster = folium.raster_layers.TileLayer(ee_object.getMapId(vis_params)['tile_fetcher']
                                            name=name,
                                            overlay=True,
                                            control=True,
                                            attr='Map Data &copy; <a href="https://earthengine.
    return raster
```

```
# US Counties dataset
US_counties = ee.FeatureCollection("TIGER/2018/Counties")

# Select county of interest
state_FIP = '20'
county_name = 'Thomas'
county = US_counties.filter(ee.Filter.eq('STATEFP','20').And(ee.Filter.eq('NAME','Thomas')
county_meta = county.getInfo()
```

```
### Select cropland datalayer ###
start_date = '2018-01-01'
end_date = '2018-12-31'
CDL = ee.ImageCollection('USDA/NASS/CDL').filter(ee.Filter.date(start_date,end_date)).firs
cropland = CDL.select('cropland')

# Clip cropland layer to selected county
county_cropland = cropland.clip(county)


### Select vegetation index (vi) ###
band = 'EVI' # or 'NDVI'

# Define start and end of time series for vegetation index
start_date_vi = '2017-10-15'
end_date_vi = '2018-06-15'

# Request dataset and band
MCD43A4 = ee.ImageCollection('MODIS/MCD43A4_006_EVI').filterDate(start_date_vi,end_date_vi
vi = MCD43A4.select('EVI')
```

```
# Get county boundaries
location_lat = float(county_meta['features'][0]['properties']['INTPTLAT'])
location_lon = float(county_meta['features'][0]['properties']['INTPTLON'])

# Visualize county boundaries
m = folium.Map(location=[location_lat, location_lon], zoom_start=10)

# Add click event to paste coordinates into the clipboard
m.add_child(folium.ClickForLatLng(alert=False))
m.add_child(folium.LatLngPopup())
```

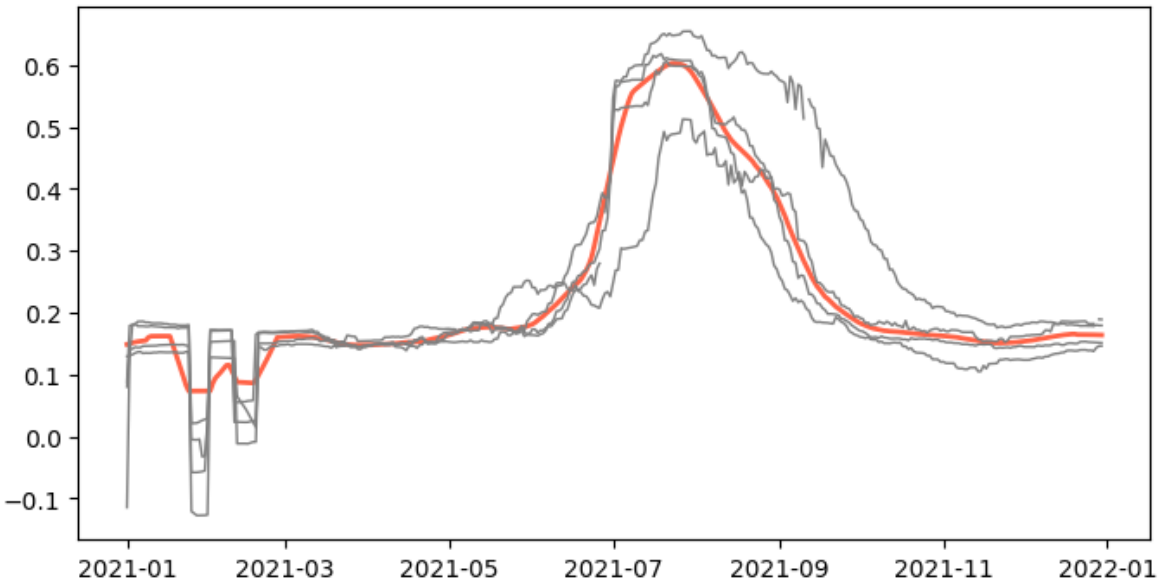
```
# Create raster using function defined earlier and add map
create_raster(county_cropland, {}, 'cropland').add_to(m)
folium.GeoJson(county.getInfo(),
               name='County boundary',
        style_function=lambda feature: {
        'fillColor': 'None',
        'color': 'black',
        'weight': 2,
        'dashArray': '5, 5'
    }).add_to(m)

# Add some controls
folium.LayerControl().add_to(m)

# Display map
m
```

```
<folium.folium.Map at 0x7fb070a34160>
```

```
lat, lon = eval(clipboard.paste())
print(lat, lon)

# Define selected coordinates as point geometry
field_point = ee.Geometry.Point([lon, lat])

point_evi = EVI.getRegion(field_point, scale=1).getInfo()
df_point = array_to_df(point_evi)

if 'df' not in locals():
    df = df_point
else:
    df = pd.concat([df, df_point])

evi_mean = df.groupby(by='time', as_index=False)['EVI'].median()
evi_mean['smoothed'] = evi_mean['EVI'].rolling(window=15, min_periods=1, center=True).mean

# Create figure
plt.figure(figsize=(8,4))
plt.plot(evi_mean['time'], evi_mean['smoothed'], color='tomato', linewidth=2)
for lat in df['latitude'].unique():
    idx = df['latitude'] == lat
```

```
    plt.plot(df.loc[idx,'time'], df.loc[idx,'EVI'], linestyle='-', linewidth=1, color='gra
plt.show()
```

```
39.402244 -100.990448
```



6 Image for an area

When dealing with geospatial data, analyzing single images can provide detailed insights into specific temporal snapshots of a region. Unlike animations that track changes over time, working with individual images allows for a focused examination of spatial characteristics of a region at a given moment. Google Earth Engine offers robust tools for accessing and processing these images, enabling researchers to extract valuable information from diverse datasets. This tutorial will guide you through the process of retrieving and analyzing single images from Google Earth Engine.

This tutorial focused on static properties that do not change substantially over time, like elevation above sea level and some soil properties.

```
# Import modules
import ee
import requests
import numpy as np
import pandas as pd
import xarray as xr
import matplotlib.pyplot as plt
from matplotlib import colors, colormaps
from pprint import pprint
import json
from datetime import datetime
import io
```

```
# Authenticate
# ee. Authernticate()

# Initialize the library.
ee.Initialize()
```

6.1 Define helper functions

```
# Define function to save images to the local drive
def save_geotiff(ee_image, filename, crs, scale, geom, bands=[]):
    """
    Function to save images from Google Earth Engine into local hard drive.
    """
    image_url = ee_image.getDownloadUrl({'region': geom,'scale':scale,
                                         'bands': bands,
                                         'crs': f'EPSG:{crs}',
                                         'format': 'GEO_TIFF'})

    # Request data using URL and save data as a new GeoTiff file
    response = requests.get(image_url)
    with open(filename, 'wb') as f:
        f.write(response.content)
        return print('Saved image')


# Define function to retrieve colormaps
def get_hex_cmap(name,n=10):
    """
    Function to get list of HEX colors from a Matplotlib colormap.
    """
    rgb_cmap = colormaps.get_cmap(name)
    rgb_index = np.linspace(0, rgb_cmap.N-1, n).astype(int)
    hex_cmap = [colors.rgb2hex(rgb_cmap(k)) for k in rgb_index]
    return hex_cmap
```

6.2 Example 1: State level elevation

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
region = US_states.filter(ee.Filter.eq('NAME','Kansas'))

# Create mask
mask = ee.Image.constant(1).clip(region).mask()
```

```
# Get image with elevation data from the Shuttle Radar Topography Mission (SRTM)
srtm = ee.Image("USGS/SRTMGL1_003")
elev_img = srtm.clip(region).mask(mask)
```

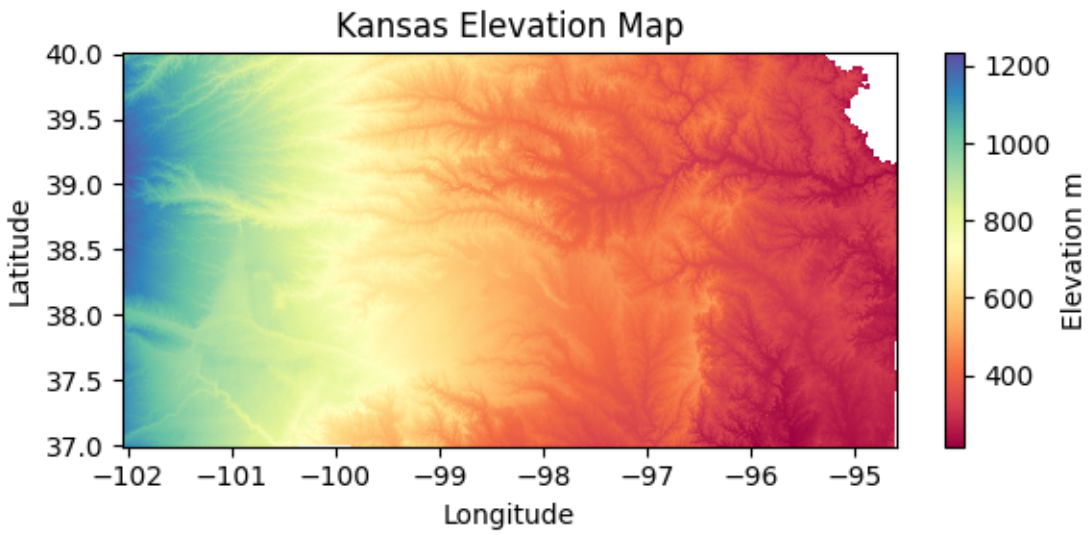
```
# Get colormap for both geotiff raster and folium raster amp
cmap = get_hex_cmap('Spectral', 12)
```

```
# Save geotiff
elev_filename = '../outputs/kansas_elevation_250m.tif'
save_geotiff(elev_img, elev_filename, crs=4326, scale=250, geom=region.geometry())
```

```
Saved image
```

```
# Read GeoTiff file using Xarray (and remove extra dimension)
elev_raster = xr.open_dataarray(elev_filename).squeeze()
```

```
# Create figure
elev_raster.plot.imshow(figsize=(6,3), cmap='Spectral', add_colorbar=True,
                        cbar_kwargs={'label':'Elevation m'});
plt.title('Kansas Elevation Map')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
#plt.axis('equal')
plt.tight_layout()
plt.show()
```



6.3 Example 2: State level soil textural class

In this example we will plot the 12 soil textural classes for the state of Kansas. We will also learn how to use Matplotlib's object-based syntax to define a colorbar with custom labels. The source for this example is from the 800-meter spatial resolution gridded soil product created by Walkinshaw et al. (2020) using the USDA-NRCS Soil Survey Geodatabase. Check out the link below for some cool maps:

- Walkinshaw, Mike, A.T. O'Geen, D.E. Beaudette. "Soil Properties." California Soil Resource Lab, 1 Oct. 2020, casoilresource.lawr.ucdavis.edu/soil-properties/.

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
region = US_states.filter(ee.Filter.eq('NAME','Kansas'))

# Create mask
mask = ee.Image.constant(1).clip(region).mask()
```

```
url_link = 'projects/earthengine-legacy/assets/projects/sat-io/open-datasets/CSRL_soil_pro
texture_img = ee.Image(url_link).clip(region).mask(mask)

# Palette
```

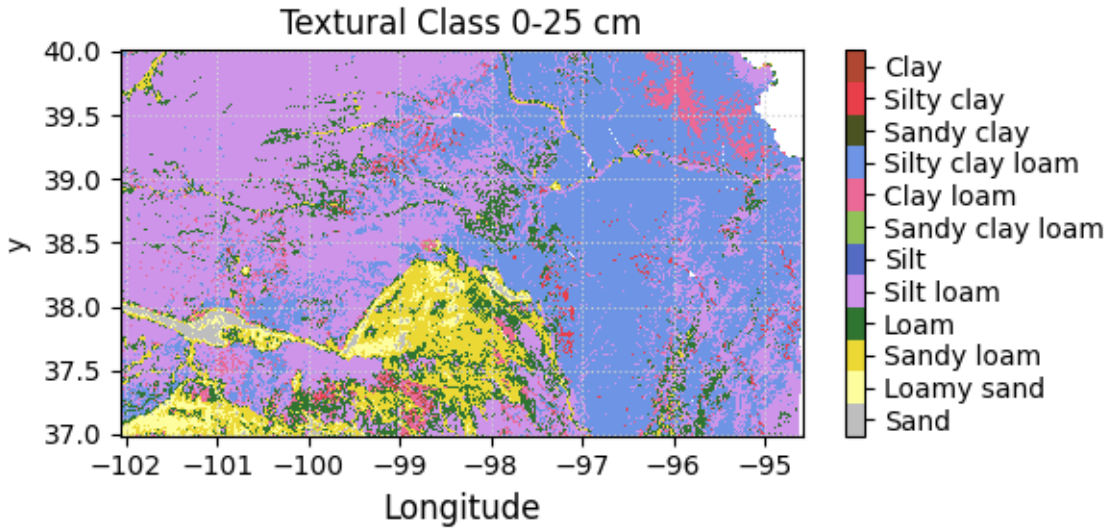
```
palette = ['#BEBEBE', #Sand
           '#FDFD9E', #Loamy Sand
           '#ebd834', #Sandy Loam
           '#307431', #Loam
           '#CD94EA', #Silt Loam
           '#546BC3', #Silt
           '#92C158', #Sandy Clay Loam
           '#EA6996', #Clay Loam
           '#6D94E5', #Silty Clay Loam
           '#4C5323', #Sandy Clay
           '#E93F4A', #Silty Clay
           '#AF4732', #Clay
          ]

# Visualization parameters
texture_cmap = colors.ListedColormap(palette)

# Save geotiff
texture_filename = '../outputs/kansas_texture_800m.tif'
save_geotiff(texture_img, texture_filename, crs=4326, scale=800, geom=region.geometry())
```

Saved image

```
# Read saved geotiff image
texture_raster = xr.open_dataarray(texture_filename).squeeze()
```

```
fig, ax = plt.subplots(figsize=(6,3))

raster = texture_raster.plot.imshow(ax=ax, cmap=texture_cmap,
                                    add_colorbar=False, vmin=1, vmax=13)

ax.set_title('Textural Class 0-25 cm')
ax.set_xlabel('Latitude', fontsize=12)
ax.set_xlabel('Longitude', fontsize=12)
ax.grid(which='major', color='lightgrey', linestyle=':')

# Add colorbar
cbar = fig.colorbar(raster, ax=ax)

# Customize the colorbar
```

```
cbar.set_ticks(ticks=np.linspace(1.5, 12.5, 12),
               labels=['Sand','Loamy sand','Sandy loam',
                       'Loam','Silt loam','Silt',
                       'Sandy clay loam','Clay loam',
                       'Silty clay loam','Sandy clay',
                       'Silty clay','Clay'])

plt.tight_layout()
#plt.savefig('flint_hills.jpg', dpi=300)
plt.show()
```



6.4 Example 3: Regional soil properties

```
# Ecoregions map
# https://developers.google.com/earth-engine/datasets/catalog/RESOLVE_ECOREGIONS_2017#desc
eco_regions = ee.FeatureCollection("RESOLVE/ECOREGIONS/2017")

# Select flint hills region
region = eco_regions.filter(ee.Filter.inList('ECO_ID',[392])) # Find ecoregion ID using th

# Define approximate region bounding box [E,S,W,N]
bbox = ee.Geometry.Rectangle([-95.6, 36, -97.4, 40])
```

```
# Create mask for the region
mask = ee.Image.constant(1).clip(region).mask()
```

6.4.0.1 Load maps of soil physical properties

```
# SAND

# Select layer, clip to region, and then mask
sand = ee.Image("projects/soilgrids-isric/sand_mean").clip(region).mask(mask)
sand_img = sand.select('sand_0-5cm_mean').multiply(0.1) # From g/kg to %


# Save geotiff
sand_filename = '../outputs/flint_hills_sand_800m.tif'
save_geotiff(sand_img, sand_filename, crs=4326, scale=250, geom=region.geometry())
```

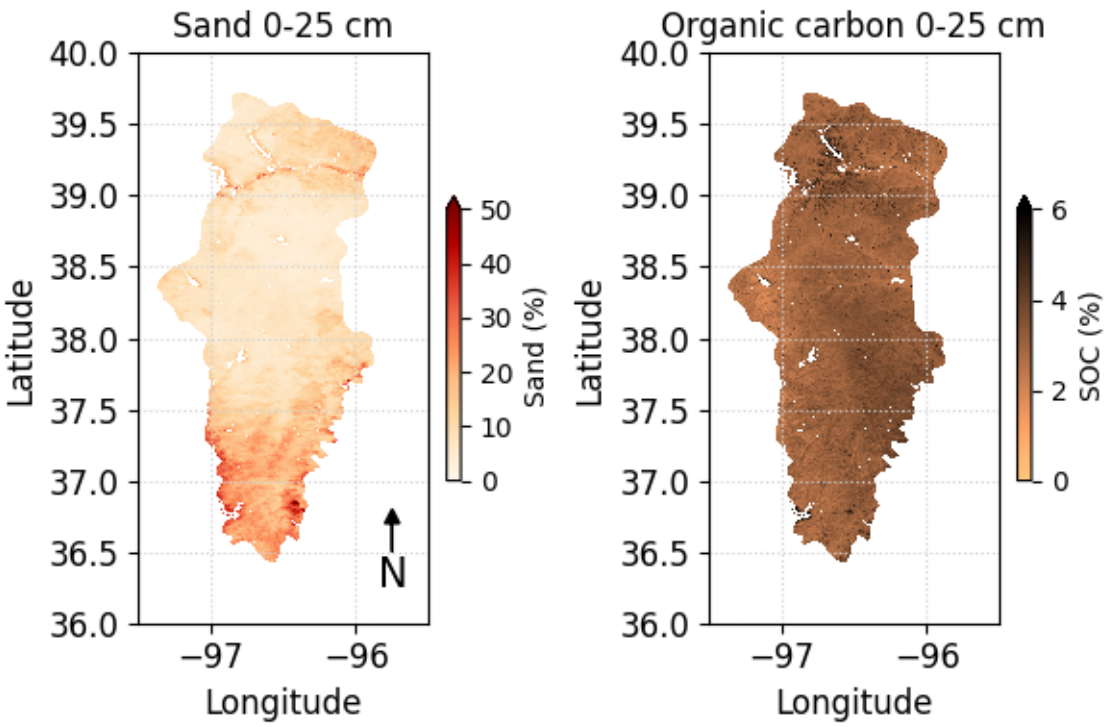
```
Saved image
```

```
# SOIL ORGANIC CARBON

# Select layer, clip to region, and then mask
soc = ee.Image("projects/soilgrids-isric/soc_mean").clip(region).mask(mask)

# Select surface sand layer
soc_img = soc.select('soc_0-5cm_mean').multiply(0.01) # From dg/kg to %


# Save geotiff
soc_filename = '../outputs/flint_hills_soc_800m.tif'
save_geotiff(soc_img, soc_filename, crs=4326, scale=250, geom=region.geometry())
```

```
Saved image
```

```
# Read GeoTiff images saved in our local drive
sand_raster = xr.open_dataarray(sand_filename).squeeze()
soc_raster = xr.open_dataarray(soc_filename).squeeze()
```

```
# Create figure
fs = 12 # Define font size variable

plt.figure(figsize=(6,4))

plt.subplot(1,2,1)
sand_raster.plot.imshow(cmap='OrRd', add_colorbar=True, vmin=0, vmax=50,
                    cbar_kwargs={'label':'Sand (%)', 'shrink':0.5});
plt.title('Sand 0-25 cm')
plt.xlabel('Longitude', fontsize=fs)
plt.ylabel('Latitude', fontsize=fs)
plt.xticks(fontsize=fs)
plt.yticks(fontsize=fs)
plt.grid(which='major', color='lightgrey', linestyle=':')
plt.arrow(-95.75, 36.5, 0, 0.2, head_width=0.1, head_length=0.1, fc='k', ec='k')
plt.text(-95.75, 36.25, 'N', fontsize=15, ha='center')
plt.xlim([-97.5, -95.5])
plt.ylim([36, 40])


plt.subplot(1,2,2)
soc_raster.plot.imshow(cmap='copper_r', add_colorbar=True, vmin=0, vmax=6,
                    cbar_kwargs={'label':'SOC (%)', 'shrink':0.5});
plt.title('Organic carbon 0-25 cm')
plt.xlabel('Longitude', fontsize=fs)
plt.ylabel('Latitude', fontsize=fs)
plt.xticks(fontsize=fs)
plt.yticks(fontsize=fs)
plt.grid(which='major', color='lightgrey', linestyle=':')
plt.xlim([-97.5, -95.5])
plt.ylim([36,40])

plt.tight_layout()
plt.show()
```



6.5 Example 4: Regional drought

In this example we will explore how to retrieve available dates for an Image Collection and then we will use that information to select a specific Image. More specifically, to illsutrate these concepts we will use the collection from the U.S. Drought Monitor, that releases weekly maps of drought conditions for the United States. The selected region for the map is the area covering the Ogallala aquifer, which is one of the largest aquifers in the world spanning multiple states in the U.S. Great Plains.

6.5.0.1 Read boundary

```
# Import boundary for Ogallala Aquifer from local drive
filename_bnd = '../datasets/ogallala_aquifer_bnd.geojson'

# Read the file
with open(filename_bnd) as file:
    roi_json = json.load(file)
```

```
# Get coordinates for plot
lon,lat = zip(*roi_json['features'][0]['geometry']['coordinates'][0])

# Define the ee.Geometry so that GEE can use it
roi_geom = ee.Geometry(roi_json['features'][0]['geometry'])

# Create mask
mask = ee.Image.constant(1).clip(roi_geom).mask()
```

6.5.0.2 Load USDM image collection

```
# Load U.S. Drought monitor Image Collection
usdm_collection = ee.ImageCollection("projects/sat-io/open-datasets/us-drought-monitor")
```

To access a specific number of items from a collection, we can use the `.toList()` method. For instance, to return the first three images of the U.S. Drought Monitor collection defined above we can run the following command: `usdm_collection.toList(3).getInfo()`

```
# Define function to get dates from each ee.Image object
get_date = lambda image: ee.Image(image).date().format('YYYY-MM-dd')

# Get the size of the image collection
N = usdm_collection.size()


# Apply function to collection
usdm_dates = usdm_collection.toList(N).map(get_date).getInfo()

# Iterate over all the dates and select images for August 2011
for k,date in enumerate(usdm_dates):
    date = datetime.strptime(date, '%Y-%m-%d')
    if date.year == 2012 and date.month == 10:
        print(k, date)
```

```
666 2012-10-02 00:00:00
667 2012-10-09 00:00:00
668 2012-10-16 00:00:00
669 2012-10-23 00:00:00
670 2012-10-30 00:00:00
```

Note that the following code will not work:

```
get_date = lambda image: image.date().format('YYYY-MM-dd')
usdm_dates = usdm_collection.map(get_date).getInfo()
```

The reason it does not work is because a mapping algorithm on a collection must return a Feature or Image, and the above code returns a date string.

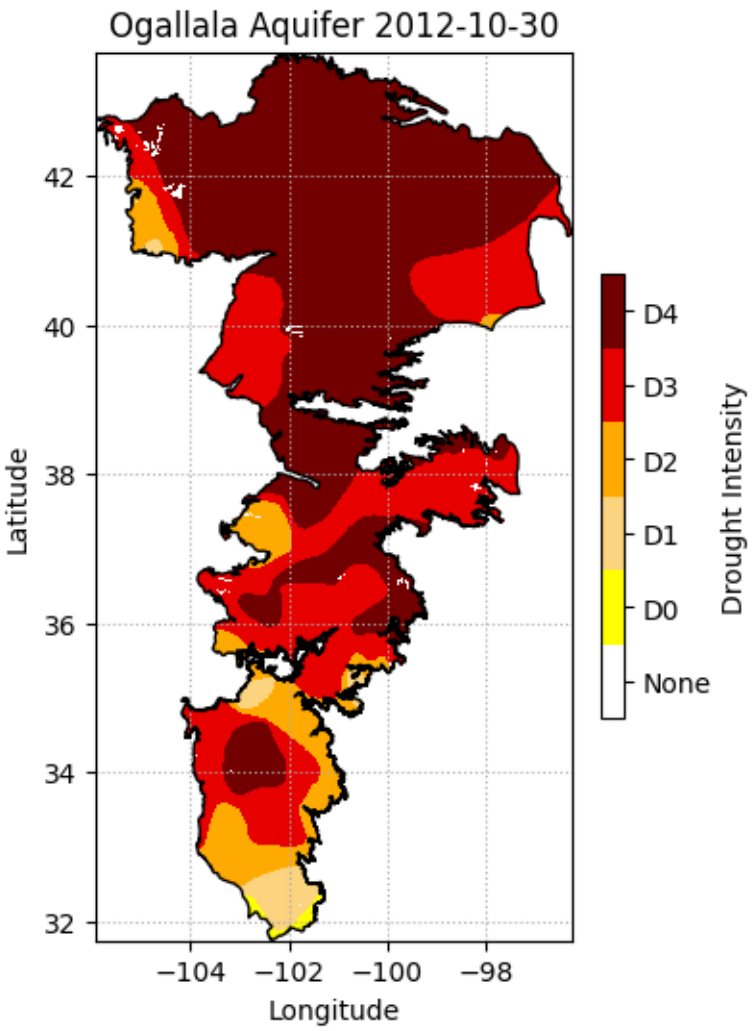
```
# Read specific image
image_number = 670
usdm_img = ee.Image(usdm_collection.toList(N).get(image_number))

# Clip and add 1 to the layer, so that we represent "None" as 0
usdm_img = usdm_img.clip(roi_geom).add(1).mask(mask)

# Get latest image of collection
# usdm_img = ee.Image(usdm_collection.toList(N).get(-1)).mask(mask)

# Get first image of collection
# usdm_img = ee.Image(usdm_collection.toList(N).get(0)).mask(mask) # or
# usdm_img = ee.Image(usdm_collection.toList(1)).mask(mask)
```

```
# Define colormap
usdm_hex_palette = ["#FFFFFF", "#FFFF00", "#FCD37F", "#FFAA00", "#E60000", "#730000"]

# Visualization parameters
usdm_cmap = colors.ListedColormap(usdm_hex_palette)
usdm_cmap
```

```
# Get map from url (it may take several seconds)
image_url = usdm_img.getDownloadUrl({
    'region': roi_geom,
    'scale':1_000,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF'})
```

```
# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)
```

```
# Check if the request was successful
if response.status_code == 200:

    # Read image data into a BytesIO object
    image_data = io.BytesIO(response.content)

    fig,ax = plt.subplots(figsize=(4,6))

    # Use Xarray to open the raster image directly from memory
    raster = xr.open_dataarray(image_data, engine='rasterio').squeeze()

    # Create raster map showing drought conditions
    usdm_map = raster.plot.imshow(ax=ax, cmap=usdm_cmap,
                                  vmin=-0.5, vmax=5.5, add_colorbar=False)

    # Add aquifer boundaries
    ax.plot(lon, lat, color='k', linewidth=1)

    # Add colorbar
    cbar = fig.colorbar(usdm_map, ax=ax, shrink=0.5, label='Drought Intensity')

    # Add labels in the center of each segment
    cbar.set_ticks(ticks=[0,1,2,3,4,5],
                   labels=['None','D0','D1','D2','D3','D4'])

    # Add labels
    ax.set_title(f'Ogallala Aquifer {usdm_dates[image_number]}')
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
    plt.grid(linestyle=':')
    plt.show()
```


Ogallala Aquifer 2012-10-30
Latitude
42
40
38
36
34
32
−104
−102
−100
−98
Longitude
D4
D3
D2
D1
D0
None
Drought Intensity

7 Reduce collection to image

Often times we are interested in summarizing a collection of images into a single image. Reducers in Google Earth Engine (GEE) are functions that aggregate data, enabling the computation of summary metrics over images, image collections, or features. Reducers can summarize data spatially, temporally, or across bands, making them powerful tools for analyzing and synthesizing large geospatial datasets.

```
# Import modules
import ee
import numpy as np
import pandas as pd
import requests
import xarray as xr
import matplotlib.pyplot as plt
from matplotlib import colors, colormaps
```

```
# Authenticate
#ee.Authenticate()

# Initialize GEE API
ee.Initialize()
```

7.1 Define helper functions

```
# Define function to save images to the local drive
def save_geotiff(ee_image, filename, crs, scale, geom, bands=[]):
    """
    Function to save images from Google Earth Engine into local hard drive.
    """
    image_url = ee_image.getDownloadUrl({'region': geom,'scale':scale,
                                         'bands': bands,
                                         'crs': f'EPSG:{crs}',
                                         'format': 'GEO_TIFF',
```

```
                                        'formatOptions': {'cloudOptimized': True,
                                                          'noDataValue': 0}})

    # Request data using URL and save data as a new GeoTiff file
    response = requests.get(image_url)
    with open(filename, 'wb') as f:
        f.write(response.content)
        return print('Saved image')
```

7.2 Example 1: Sum reducers

The sum reducer adds up all the values it encounters. This is useful for calculating total rainfall, snowfall, or any other cumulative measure over an area. IN this example we will compute the annual precipitation in 2023 across Oklahoma.

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
region = US_states.filter(ee.Filter.eq('NAME','Oklahoma'))

# Create mask
mask = ee.Image.constant(1).clip(region).mask()
```

> **Note**
>
> The `clip()` method only affects the visualization of the image in Google Earth Engine. To ensure the image is clipped in the exported GeoTIFF, we need to explicitly apply the mask.

```
prism = ee.ImageCollection('OREGONSTATE/PRISM/AN81d').filterDate('2023-01-01', '2023-12-31
precip_img = prism.select('ppt').reduce(ee.Reducer.sum()).mask(mask)
```

> **Note**
>
> The result of applying a reducer to an ImageCollection is an Image.

```
# Save geotiff
precip_filename = '../outputs/oklahoma_rainfall_2023.tif'
save_geotiff(precip_img, precip_filename, crs=4326, scale=4_000, geom=region.geometry())
```

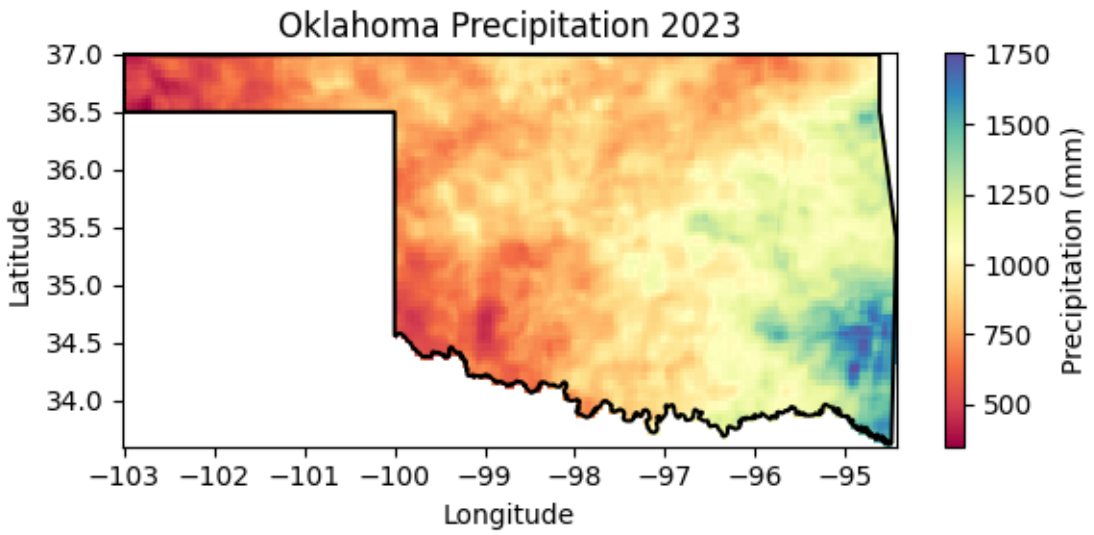
```
Saved image
```

```
# Read saved geotiff image
precip_raster = xr.open_dataarray(precip_filename).squeeze()
```

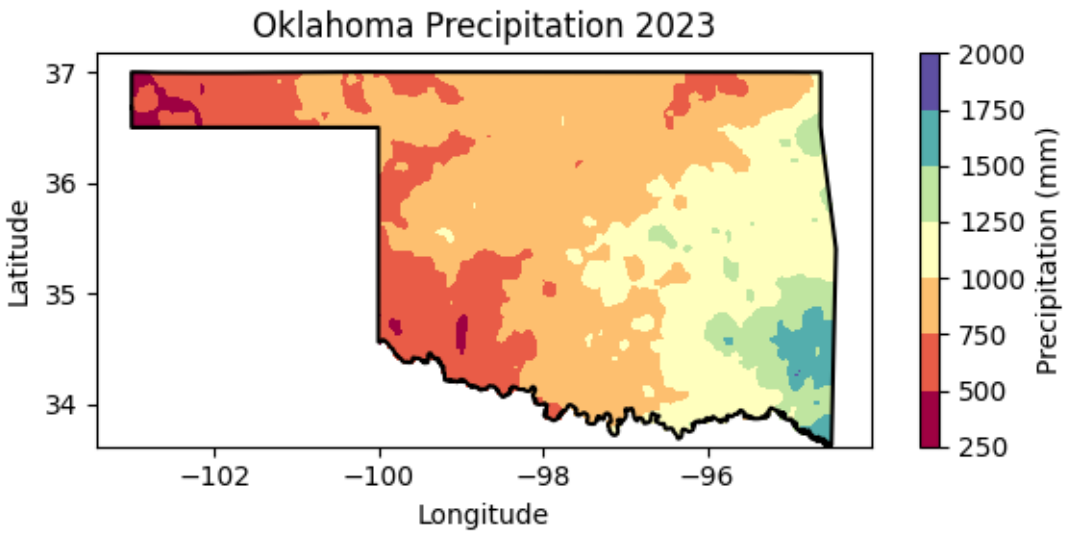
```
# Get coordinates of the county geometry
df = pd.DataFrame(region.first().getInfo()['geometry']['coordinates'][0])
df.columns = ['lon','lat']
df.head()
```

	lon	lat
0	-103.002435	36.675527
1	-103.002390	36.670994
2	-103.002390	36.668480
3	-103.002390	36.664934
4	-103.002390	36.661792

```
# Create figure
precip_raster.plot.imshow(figsize=(6,3), cmap='Spectral', add_colorbar=True,
                    cbar_kwargs={'label':'Precipitation (mm)'})
plt.plot(df['lon'], df['lat'],'-k')
plt.title('Oklahoma Precipitation 2023')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
#plt.axis('equal')
plt.tight_layout()
plt.show()
```




```
# Create contour figure
precip_raster.plot.contourf(figsize=(6,3), cmap='Spectral', add_colorbar=True,
                    cbar_kwargs={'label':'Precipitation (mm)'})
plt.plot(df['lon'], df['lat'],'-k')
plt.title('Oklahoma Precipitation 2023')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
#plt.axis('equal')
plt.tight_layout()
plt.show()
```




```
# Find min and max precipitation
precip_img.reduceRegion(reducer = ee.Reducer.minMax(),
                        geometry = region.geometry(),
                        scale = 4_000).getInfo()
```

```
{'ppt_sum_max': 1753.4329500616004, 'ppt_sum_min': 349.43359203080763}
```

> **Tip**
>
> The `.minMax()` reducer finds the minimum and maximum value. This particular reducer is useful for identifying extremes like the coldest and hottest temperatures or the lowest and highest annual precipitation amounts.

7.3 Example 2: Min Reducer

The `.min()` reducer finds the minimum value. In this example we will determine the minimum and maximum land surface teperatures.

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
```

```
region = US_states.filter(ee.Filter.eq('NAME','Kansas'))

# Create mask
mask = ee.Image.constant(1).clip(region).mask()
```

```
# Load an image collection from PRISM
prism = ee.ImageCollection('OREGONSTATE/PRISM/AN81d').filterDate('2010-01-01', '2020-12-31
tmin_img = prism.select('tmin').reduce(ee.Reducer.min()).mask(mask)
```

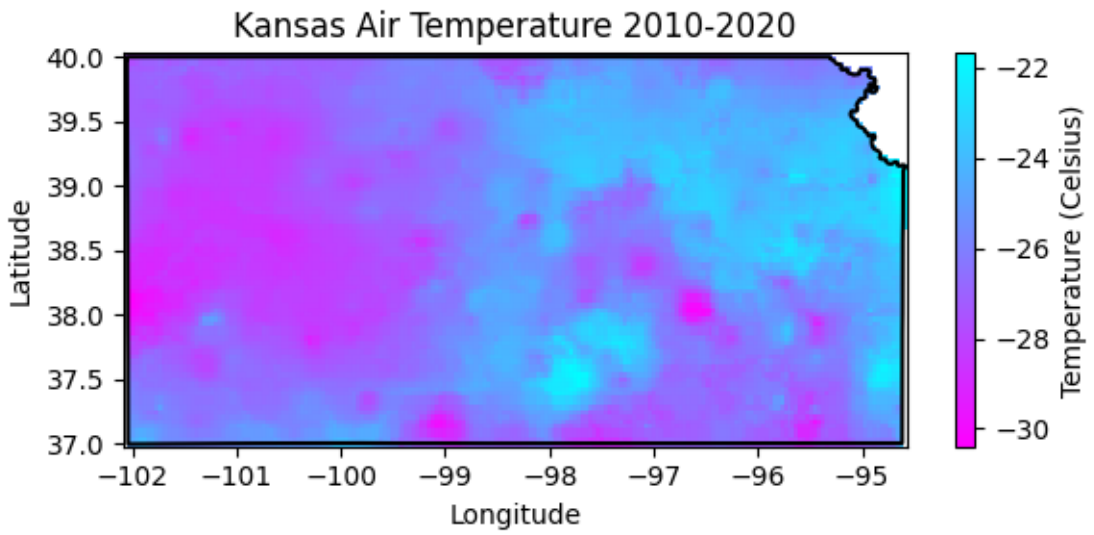
```
# Save geotiff
tmin_filename = '../outputs/kansas_tmin_2010_2020.tif'
save_geotiff(tmin_img, tmin_filename, crs=4326, scale=4000, geom=region.geometry())
```

```
Saved image
```

```
# Read saved geotiff image
tmin_raster = xr.open_dataarray(tmin_filename).squeeze()
```

```
# Get the dictionary with all the metadata into a variable
# Print this variable to see the details
region_metadata = region.first().getInfo()

# Get coordinates of the county geometry
df = pd.DataFrame(region_metadata['geometry']['geometries'][1]['coordinates'][0])
df.columns = ['lon','lat']
```

```
# Create figure
tmin_raster.plot.imshow(figsize=(6,3), cmap='cool_r', add_colorbar=True,
                    cbar_kwargs={'label':'Temperature (Celsius)'})
plt.plot(df['lon'], df['lat'],'-k')
plt.title('Kansas Air Temperature 2010-2020')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
#plt.axis('equal')
plt.tight_layout()
plt.show()
```




```
# Find lowest temperature
tmin_img.reduceRegion(reducer = ee.Reducer.min(),
                      geometry = region.geometry(),
                      scale = 4_000).getInfo()
```

```
{'tmin_min': -30.395000457763672}
```

> I still need to figure out how to obtain the coordinates of the location with the lowest temperature. If someone know the answer, please send me an e-mail or push the code to Github.

7.4 Example 4: Max reducer

The `.max()` reducer find the maximum value. In this example we will use it to find the total spatial extent burned by wildfires over multiple days in southwest Kansas and northwest Oklahoma.

The Starbuck wildfire occurred in March 2017 and stands as one of the largest wildfires in Kansas history. The Starbuck wildfire started in Beaver county in Oklahoma and then spread across multiple Kansas counties, including Meade, Clark, and Comanche counties, ravaging over 662,000 acres of land that included ranches and residential areas. The fire's magnitude was so extensive that it not only caused significant ecological damage but also resulted in the loss of numerous cattle, property destruction, and challenged the resilience of local communities. The Starbuck wildfire highlighted the importance of community solidarity, the challenges

of managing fire risk in rural regions, and the necessity for improved fire management, preparedness, and mitigation strategies.

```
# Load the US county boundaries.
counties = ee.FeatureCollection('TIGER/2016/Counties');

# Filter the counties by name and state (Kansas FIPS code is "20")
ks_counties = counties.filter(ee.Filter.Or(ee.Filter.eq('NAME', 'Clark'),
                                           ee.Filter.eq('NAME', 'Meade'),
                                           ee.Filter.eq('NAME', 'Comanche'))).filter(ee.Fi

# Filter the counties by name and state (Oklahoma FIPS code is "40")
ok_counties = counties.filter(ee.Filter.Or(ee.Filter.eq('NAME', 'Beaver'),
                                           ee.Filter.eq('NAME', 'Harper'))).filter(ee.Filt


## Combine the selected counties into a single geometry.
region = ks_counties.merge(ok_counties)

# Combined counties (not use in the tutorial)
region_union = region.union()

# Get bounding box of combined geometry
bbox = region.geometry().bounds()
```

> **Note**
>
> For combining FeatureCollections we use the `merge()` method. To combine geometries or features within a FeatureCollection we use the `.union()` method.

```
# Load modis product
modis = ee.ImageCollection('MODIS/061/MOD14A1').filterDate('2017-03-01', '2017-03-31')
max_fire = modis.select('FireMask').reduce(ee.Reducer.max())
```

```
# Save GeoTIFF
max_fire_filename = '../outputs/starbuck_wildfire_.tiff'
save_geotiff(max_fire, filename=max_fire_filename, crs=4326, scale=1000,
             geom=bbox, bands=[])
```

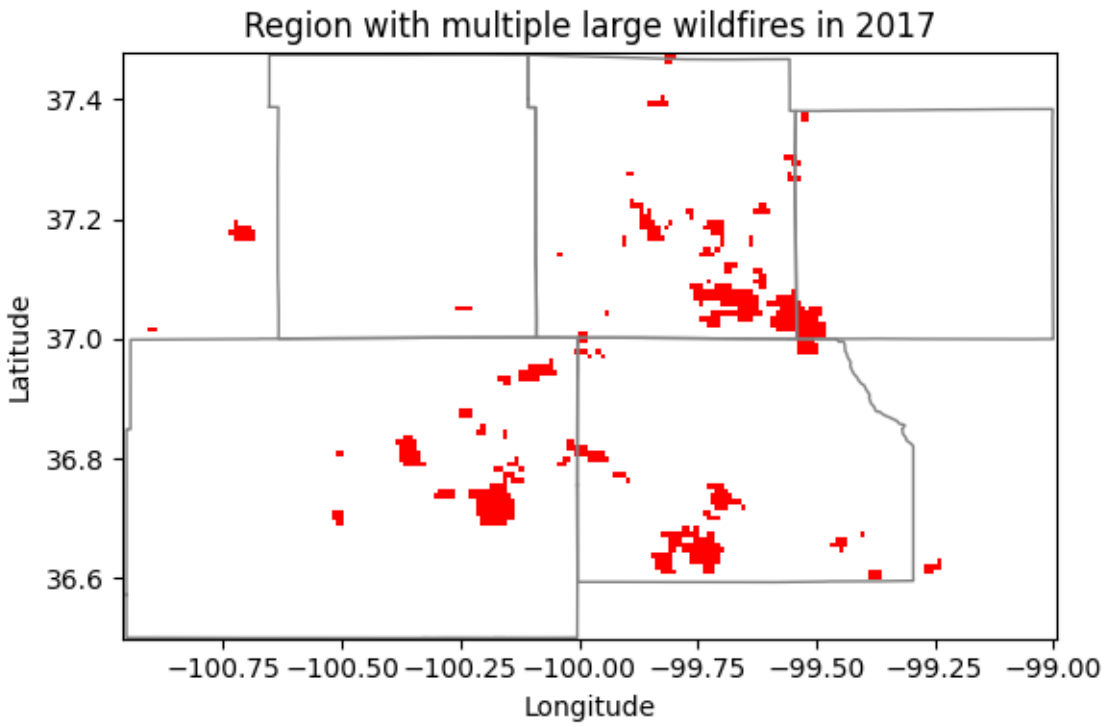
```
Saved image
```

```
# Get geometry coordinates for all counties (this is a MultiPolygon object)
region_geom = region.geometry().getInfo()
```

```
# Read saved geotiff image
max_fire_raster = xr.open_dataarray(max_fire_filename).squeeze()
```

```
# Paletter of colors for the Enhanced Vegetation Index
hex_palette = ['#ffffff','#ff0000']

# Use the built-in ListedColormap function to do the conversion
cmap = colors.ListedColormap(hex_palette)
```

```
# Create figure
max_fire_raster.plot.imshow(figsize=(6,4), cmap=cmap, add_colorbar=False)

for r in region_geom['coordinates']:
    lon,lat = zip(*r[0])
    plt.plot(list(lon), list(lat), linestyle='-', linewidth=1, color='grey')

plt.title('Region with multiple large wildfires in 2017')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
#plt.axis('equal')
plt.tight_layout()
plt.show()
```



7.5 Example 5: Median reducer

The median reducer computes the typicalvalue of a set of numbers. It's often used to aggregate pixel values across images or image collections.

In this example we will compute the median enhanced vegetation index for a county and we learn how to:

- filter a FeatureCollection,
- compute the median of an ImageCollection,
- create, apply, and update a mask,
- retrieve, save, and read a geotiff image

```
# US Counties dataset
US_counties = ee.FeatureCollection("TIGER/2018/Counties")

# Select county of interest
county = US_counties.filter(ee.Filter.eq('GEOID','20161'))

# Create mask
```

```
mask = ee.Image.constant(1).clip(county).mask()
```

```
# Modis EVI
modis_evi = ee.ImageCollection('MODIS/MCD43A4_006_EVI')

evi_img = modis_evi.filterDate('2021-04-01', '2021-09-30') \
          .select('EVI').filterBounds(county).median()

# Create mask for region
evi_img = evi_img.mask(mask)

# Update mask to avoid water bodies (wich typically have evi<0.2)
evi_img = evi_img.updateMask(evi_img.gt(0.2))
```

> **Note**
>
> `collection.median()` is a shorter way to compute the median than `collection.reduce(ee.Reducer.median())`. The latter offers more flexibility and allows for additional configurations not available through the shorter and more convenient method.

```
# Set visualization parameters.
# https://colorbrewer2.org/#type=sequential&scheme=YlOrBr&n=7
evi_palette = ['#CE7E45', '#DF923D', '#F1B555', '#FCD163', '#99B718', '#74A901',
               '#66A000', '#529400', '#3E8601', '#207401', '#056201', '#004C00', '#023B01',
               '#012E01', '#011D01', '#011301']
evi_cmap = colors.ListedColormap(evi_palette)
```

```
# Save geotiff image
evi_filename = '../outputs/evi_riley_county.tif'
save_geotiff(evi_img, evi_filename, crs=4326, scale=250,
             geom=county.geometry(), bands=['EVI'])
```

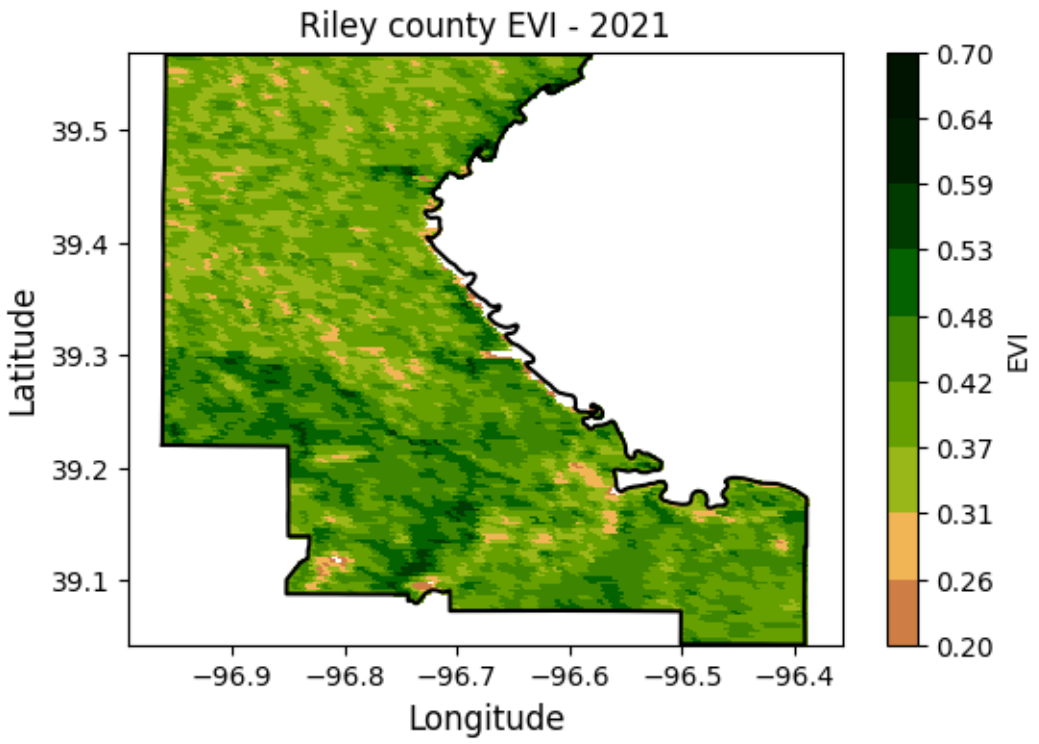
```
Saved image
```

```
# Read GeoTiff file using the Xarray package
evi_raster = xr.open_dataarray(evi_filename).squeeze()
```

```
# Get coordinates of the county geometry
df = pd.DataFrame(county.first().getInfo()['geometry']['coordinates'][0])
df.columns = ['lon','lat']
df.head()
```

	lon	lat
0	-96.961683	39.220095
1	-96.961369	39.220095
2	-96.956566	39.220005
3	-96.954188	39.220005
4	-96.952482	39.220005

```
# Create figure

plt.figure(figsize=(6,4))
evi_raster.plot.imshow(cmap=evi_cmap, vmin=0.2, vmax=0.7,
                       levels=10, cbar_kwargs={'label':'EVI', 'format':"{x:.2f}"}) # Can a
plt.plot(df['lon'], df['lat'],'-k')
plt.title('Riley county EVI - 2021', size=12)
plt.xlabel('Longitude', size=12)
plt.ylabel('Latitude', size=12)
plt.xlim([-97,-96.35])
plt.ylim([39, 39.6])
plt.axis('equal')
#plt.savefig('../outputs/riley_county_median_evi.jpg', dpi=300)
plt.show()
```



7.6 Example 6: Mean reducer

The mean reducer computes the pixel-wise average value of a set of images in a collection. It's often used to aggregate pixel values across images or image collections. In this exercise we will compute the mean NDVI for each pixel over a watershed.

```
# Read US watersheds using hydrologic unit codes (HUC)
watersheds = ee.FeatureCollection("USGS/WBD/2017/HUC12")
mcdowell_creek = watersheds.filter(ee.Filter.eq('huc12', '102701010204')).first()

# Get geometry so that we can plot the boundary
mcdowell_creek_geom = mcdowell_creek.geometry().getInfo()

# Create mask for the map
mask = ee.Image.constant(1).clip(mcdowell_creek).mask()
```

```
# Get watershed boundaries
df_mcdowell_creek_bnd = pd.DataFrame(np.squeeze(mcdowell_creek_geom['coordinates']),
                                     columns=['lon','lat'])
# Inspect dataframe
df_mcdowell_creek_bnd.head()
```

	lon	lat
0	-96.552986	39.093157
1	-96.553307	39.093741
2	-96.553592	39.094234
3	-96.553683	39.094391
4	-96.553988	39.094490

> **Note**
>
> Sometimes regions like watersheds are returned as a `feature` instead of a `geometry`. This will not work for reducing operations. To solve the problem we can use the `geometry()` method. To inspect whether the returned object is a `feature` or a `geometry` use the `getInfo()` method to inspect the object.

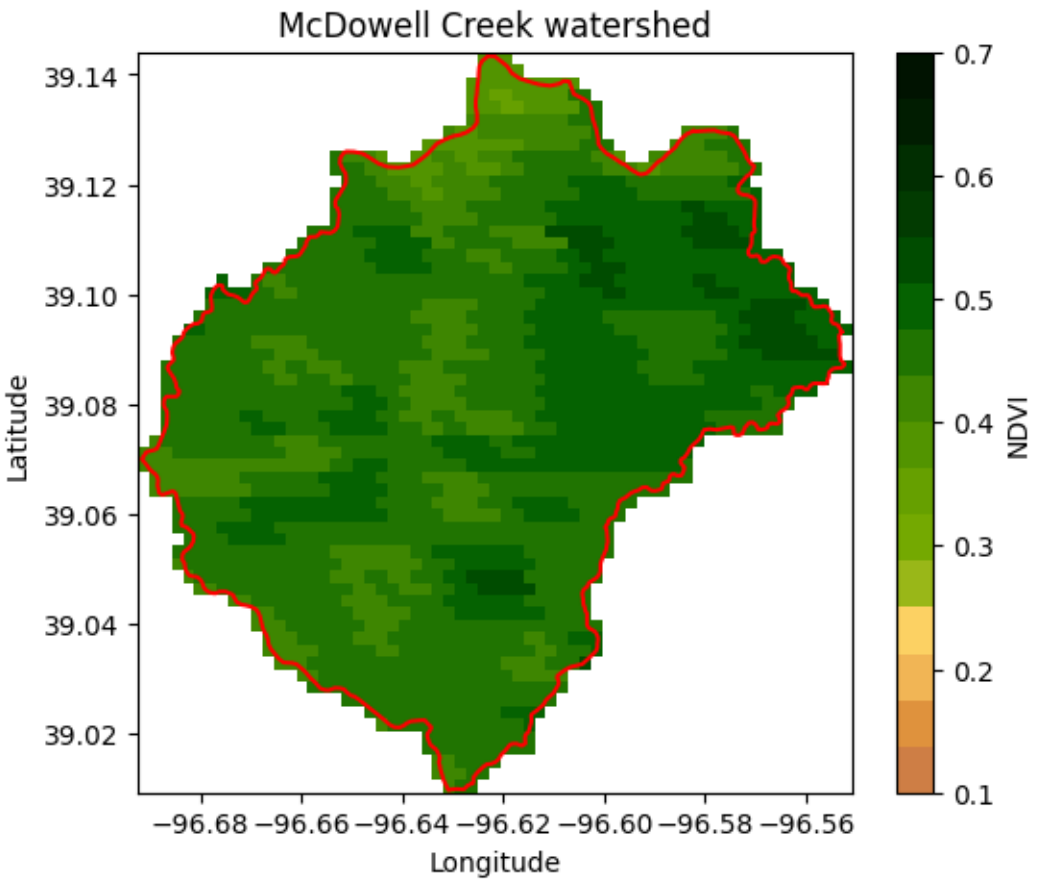
```
# Get collection for Terra Vegetation Indices 16-Day Global 500m
start_date = '2022-01-01'
end_date = '2022-12-31'
collection = ee.ImageCollection('MODIS/006/MOD13A1').filterDate(start_date, end_date)
```

```
# Reduce the collection to an image with a median reducer.
ndvi_mean = collection.mean().multiply(0.0001).clip(mcdowell_creek)

# Apply mask
ndvi_mean = ndvi_mean.mask(mask)

# Save geotiff image
ndvi_filename = '../outputs/ndvi_mean_mcdowell_creek.tif'
save_geotiff(ndvi_mean, ndvi_filename, crs=4326, scale=250,
             geom=mcdowell_creek.geometry(), bands=['NDVI'])
```

```
Saved image
```

```
# Define NDVI colormap
hex_palette = ['#CE7E45', '#DF923D', '#F1B555', '#FCD163', '#99B718', '#74A901',
               '#66A000', '#529400', '#3E8601', '#207401', '#056201', '#004C00', '#023B01',
               '#012E01', '#011D01', '#011301']

# Use the built-in ListedColormap function to do the conversion
ndvi_cmap = colors.ListedColormap(hex_palette)
```

```
# Read GeoTiff file using the Xarray package
raster = xr.open_dataarray(filename).squeeze()

plt.figure(figsize=(6,5))
raster.plot(cmap=ndvi_cmap, cbar_kwargs={'label':'NDVI'}, vmin=0.1, vmax=0.7)
plt.plot(df_mcdowell_creek_bnd['lon'], df_mcdowell_creek_bnd['lat'], color='r')
plt.title('McDowell Creek watershed')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.show()
```



7.7 Example 7: Count reducer over a region

The count reducer tallies the number of values, useful for counting the number of observations or pixels within a region that meet certain criteria. In this example we will find the area of Kansas covered by grasslands. If you thought that Kansas was mostly covered by cropland, then you are up for a surprise.

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
region = US_states.filter(ee.Filter.inList('NAME',['Kansas']))
```

```
# Land use for 2021
land_use = ee.ImageCollection('USDA/NASS/CDL') \
             .filter(ee.Filter.date('2021-01-01', '2021-12-31')).first().clip(region)

# Select cropland layer
cropland = land_use.select('cropland')
```

```
# Select pixels for a cover
land_cover_value = 176 # This is the code for grasslands
land_cover_img = cropland.eq(land_cover_value).selfMask()
```

```
# Save geotiff image
land_cover_filename = '../outputs/land_cover_kansas.tif'
save_geotiff(land_cover_img, land_cover_filename, crs=4326, scale=250,
             geom=region.geometry())
```

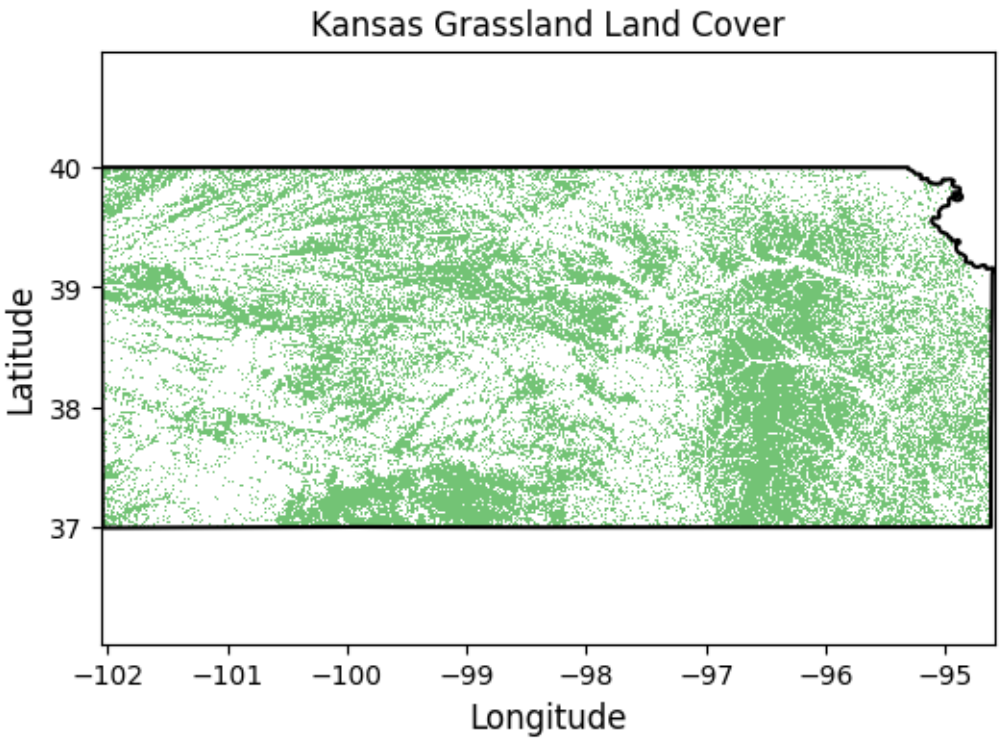
Saved image

```
# Read GeoTiff file using the Xarray package
land_cover_raster = xr.open_dataarray(land_cover_filename).squeeze()
```

```
# Get the dictionary with all the metadata into a variable
# Print this variable to see the details
region_metadata = region.first().getInfo()

# Get coordinates of the county geometry
df = pd.DataFrame(region_metadata['geometry']['geometries'][1]['coordinates'][0])
df.columns = ['lon','lat']
```

```
# Create figure

plt.figure(figsize=(6,4))
land_cover_raster.plot.imshow(cmap='Greens', add_colorbar=False)
plt.plot(df['lon'], df['lat'],'-k')
plt.title('Kansas Grassland Land Cover', size=12)
plt.xlabel('Longitude', size=12)
plt.ylabel('Latitude', size=12)
plt.xlim([-97,-96.35])
plt.ylim([39, 39.6])
```

```
plt.axis('equal')
#plt.savefig('../outputs/kansas_grasslands.jpg', dpi=300)
plt.show()
```




```
# Reduce the collection and compute area of land cover
# Kansas has an area of 213,100 square kilometers

# Apply reducer
land_cover_count = land_cover_img.reduceRegion(reducer=ee.Reducer.count(),
                                               geometry=region.geometry(),
                                               scale=250)

land_cover_pixels = land_cover_count.get('cropland').getInfo()
print(f'There are a total of {land_cover_pixels:,} pixels under grassland at 250 m spatial

state_area = 213_000 # km^2
state_fraction = (land_cover_pixels*250**2) / 10**6 / state_area
print(f'Kansas has a {round(state_fraction*100)} % of the area covered by grassland')
```

```
There are a total of 1,444,936 pixels under grassland at 250 m spatial resolution per pixel
Kansas has a 42 % of the area covered by grassland
```

The `reduceRegion()` function applies a reducer to all the pixels in a specified region. Altrnatively you can pass options as keyword arguments using a dictionary, like so:

```
land_cover_count = land_cover_img.reduceRegion(**{
    'reducer': ee.Reducer.count(),
    'geometry': region.geometry(),
    'scale': 250})
```

8 Reduce collection to time series

In geospatial analysis, collections of images are typically aggregated into images, but it is also possible to aggregate image collections in the form of a time series to visualize meaningful trends or patterns over a given area and time range.

This tutorial will guide you through the process of synthesizing each image in a collection of images into a single average value representing the entire region of interest, thus, creating a time series that reflects the mean value for a region over time.

```
# Import modules
import ee
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import xarray as xr
import requests
import json
```

8.1 Example 1: Mean vegetation dynamics for entire watershed

In this exercise we will reduce an image collection into a time series of average values of Enhanced Vegetation Index (EVI) for the region of interest over the specified date range.

```
# Trigger the authentication flow.
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

```
# Read US watersheds using hydrologic unit codes (HUC)
watersheds = ee.FeatureCollection("USGS/WBD/2017/HUC12")
mcdowell_creek = watersheds.filter(ee.Filter.eq('huc12', '102701010204')).first().geometry
mcdowell_creek_geom = mcdowell_creek.getInfo()
```

```
# Get collection for Terra Vegetation Indices 16-Day Global 500m
start_date = '2021-01-01'
end_date = '2022-12-31'
collection = ee.ImageCollection('MODIS/006/MOD13A1') \
                .filterDate(start_date, end_date) \
                .select('EVI')
```

The enxt step is to create a function that will be applied to each image of the ImageCollection. The function will aggregate all the pixels in the region of interest into a single average value. Since each image represent a day, the resulting average value will be associated with the date of the image, so that then we can create a time series.

```
# Function to apply reduceRegion to each image and extract the areal average EVI.
def compute_mean_evi(image):
    mean_evi = image.reduceRegion(
        reducer=ee.Reducer.mean(),
        geometry=mcdowell_creek,
        scale=500,
        maxPixels=1e9
    )

    # Get the date of the image.
    date = image.date().format('YYYY-MM-dd')

    # Return a Feature with properties for mean EVI and date.
    #return ee.Feature(None, {'meanEVI': mean_evi.get('EVI'), 'date': date})
    return ee.Feature(None, {'EVI_avg': mean_evi.get('EVI'), 'date': date})

# Map the function over the collection.
evi_means = collection.map(compute_mean_evi) # This is a feature collection
```

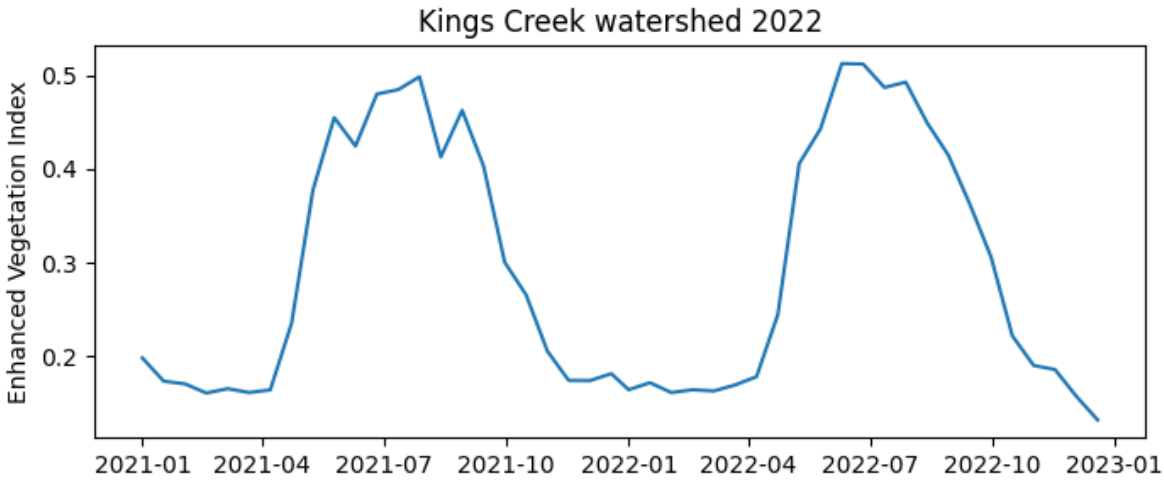
```
# Convert the resulting Feature Collection to a list of dictionaries.
reducer = ee.Reducer.toList(2)
selector = ['date','EVI_avg']
evi_data_list = evi_means.reduceColumns(reducer, selector).values().getInfo()
```

```
# Convert the list of data to a pandas DataFrame for easy handling and visualization.
df = pd.DataFrame(np.squeeze(evi_data_list), columns=['date','EVI_avg']).dropna()

# Apply factor to EVI
df['EVI_avg'] = df['EVI_avg'].astype(float) * 0.0001
```

```
# Convert dates to datetime format
df['date'] = pd.to_datetime(df['date'])

# Display a few rows
df.head(3)
```

	date	EVI_avg
0	2021-01-01	0.197894
1	2021-01-17	0.173088
2	2021-02-02	0.170202

```
# Create figure
plt.figure(figsize=(8,3))
plt.title('Kings Creek watershed 2022')
plt.plot(df['date'], df['EVI_avg'])
plt.ylabel('Enhanced Vegetation Index')
plt.show()
```



8.2 Example 2: Mean ground water anomaly for entire aquifer

The Gravity Recovery and Climate Experiment (GRACE) mission utilizes a pair of satellites in tandem orbit to measure small variations in Earth's gravitational pull. These variations are caused by changes in mass distribution, including water moving through the Earth's system.

As water accumulates or depletes in an area, such as in an aquifer, it alters the region's gravitational attraction, which GRACE detects.

The application of GRACE data has been crucial in monitoring the Ogallala Aquifer, one of the largest aquifers in the world, stretching across eight states in the United States. Over-reliance on this aquifer for agricultural irrigation, municipal, and industrial water has led to substantial depletion in water levels. By capturing the changes in gravitational pull over the Ogallala region, GRACE has provided researchers and policymakers with essential data on the aquifer's depletion rates.

```
# Read boundary of Ogalla Aquifer
with open('../datasets/ogallala_aquifer_bnd.geojson') as file:
    roi_json = json.load(file)

# Define the ee.Geometry
roi = ee.Geometry(roi_json['features'][0]['geometry'])

# Create mask
mask = ee.Image.constant(1).clip(roi).mask()
```

```
# Get GRACE equivalent water thickeness anomaly
grace = ee.ImageCollection('NASA/GRACE/MASS_GRIDS/LAND') \
                  .filterDate('2002-04-01', '2017-01-07') \
                  .filterBounds(roi) \
                  .select('lwe_thickness_csr')
```

```
# Create function to apply the reduceRegion() method to each image
def compute_mean_water(image):
    mean_water = image.reduceRegion(
        reducer=ee.Reducer.mean(),
        geometry=roi,
        scale=100_000,
        maxPixels=1e9
    )

    # Get the date of the image.
    date = image.date().format('YYYY-MM-dd')

    # Return a Feature with properties for mean water thickness and date.
    return ee.Feature(None, {'water_thickness_avg':mean_water.get('lwe_thickness_csr'),
                             'date': date})
```

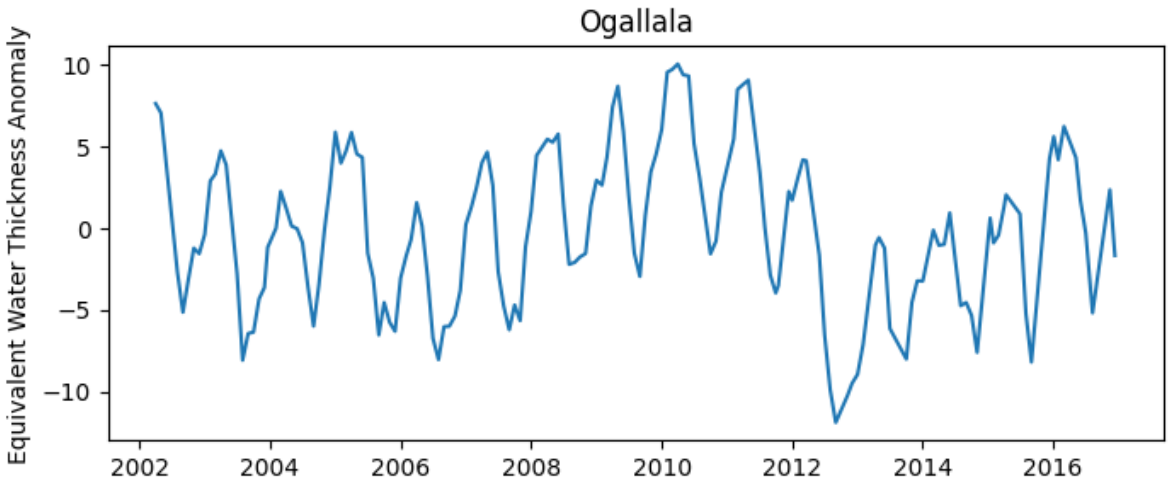
```
# Apply the created function to each image in the collection
# This operation results in a FeatureCollection
mean_water_thickness = grace.map(compute_mean_water)
```

```
# Convert the resulting Feature Collection to a list of dictionaries.
reducer = ee.Reducer.toList(2)
selector = ['date','water_thickness_avg']
data_list = mean_water_thickness.reduceColumns(reducer, selector).getInfo()
```

```
# Convert the list of data to a pandas DataFrame for easy handling and visualization.
df = pd.DataFrame(np.squeeze(data_list['list']),
                  columns=['date','water_thickness_avg']).dropna()

# Convert dates to datetime format
df['date'] = pd.to_datetime(df['date'])

df['water_thickness_avg'] = df['water_thickness_avg'].astype(float)

# Display a few rows
df.head(3)
```

	date	water_thickness_avg
0	2002-04-01	7.638767
1	2002-05-01	7.061499
2	2002-08-01	-2.686608

```
# Create figure
plt.figure(figsize=(8,3))
plt.title('Ogallala aquifer')
plt.plot(df['date'], df['water_thickness_avg'])
plt.ylabel('Equivalent Water Thickness Anomaly')
plt.show()
```


Ogallala
Equivalent Water Thickness Anomaly
10
5
0
−5
−10
2002
2004
2006
2008
2010
2012
2014
2016

9 Use my vector data

What if we want to use Google Earth Engine for our own region of interest? This could be our own field boundary, watershed, or perhaps some experimental plot. The nice part, is the we can turn any vector map into a GEE Geometry using a GeoJSON file. Let's dive into an example using a boundary layer for the Ogallala Aquifer.

You can download the file from the Github repository

```
# Import modules
import ee
import json
import requests
import matplotlib.pyplot as plt
import xarray as xr
import io
from pprint import pprint
```

```
# Authenticate
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

9.0.0.1 Load vector file from local drive

In this case we will load the boundary of the Ogallala Aquifer, spanning multiple states in the U.S. Great Plains. The map is in the `GeoJSON` format. You can also use the `GeoPandas` module to read shapefiles, and then convert them into GeJSON siles.

```
# Import boundary for Ogallala Aquifer

# From local drive
filename_bnd = '../datasets/ogallala_aquifer_bnd.geojson'
```

```
# Read the file
with open(filename_bnd) as file:
    roi_json = json.load(file)
```

9.0.0.2 Convert local geojson file into GEEO geometry

```
# Define the ee.Geometry so that GEE can use it
roi_geom = ee.Geometry(roi_json['features'][0]['geometry'])

# Create mask
mask = ee.Image.constant(1).clip(roi_geom).mask()
```

9.0.0.3 Learn how to unpack coordinates

For plotting purposes we will need to access the list of geographic coordinates from the `GeoJSON` structure. The coordinates are stored in `[lon, lat]` pairs. Here is an example:

```
[[-105.08669315719459, 41.81747353490203],
 [-105.12647954793552, 41.819996817840824],
 [-105.14733301403585, 41.821369666176125]]
```

The catch is that for plotting we need latitude and longitude in separate lists. Here is where Python shines. We typically use the `zip` function to create the above structure from individual columns. We can use the same `zip` function with the `*` operator to do the inverse. Sometimes we call this operation `unpacking`, because we turn a list packed with other lists, into separate lists.

First we need to learn how to access the raw data that we need. The code line belows shows how to access the nested coordinates, so that then we can use this line with the `zip` function. Remember to go step by step when breaking down the problem.

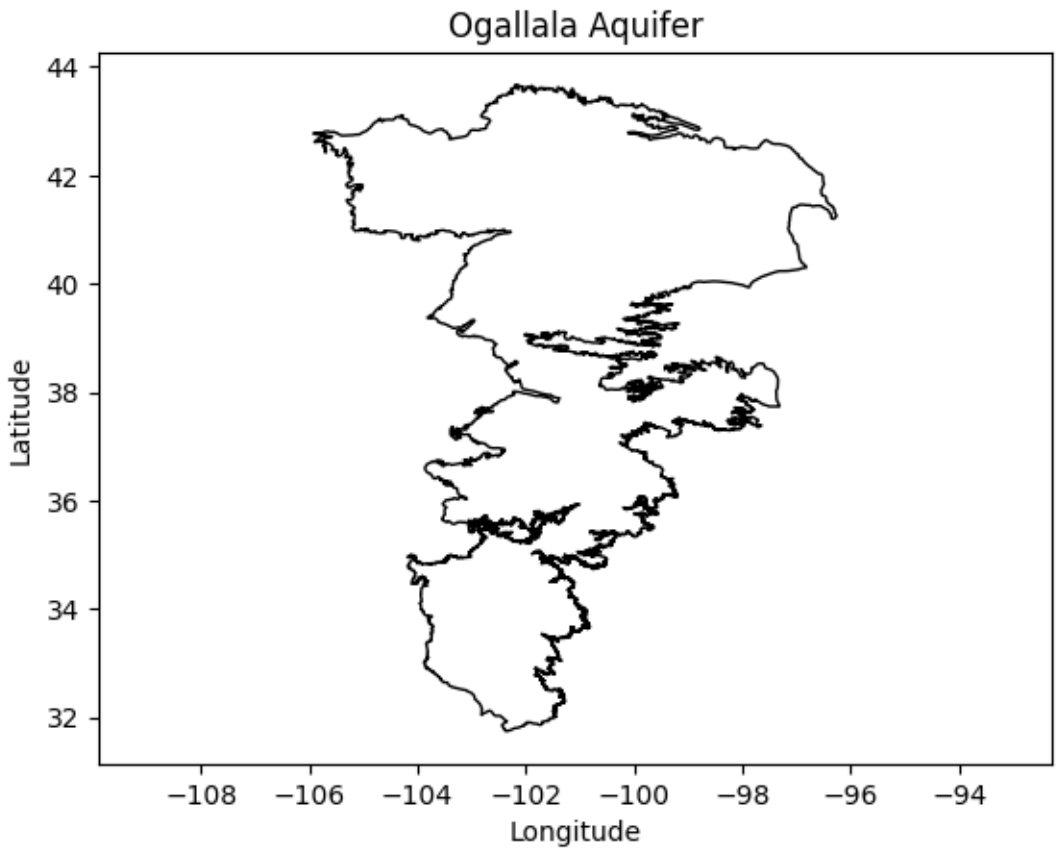
```
# Visualize the first 10 coordinate pairs inside of the geojson file (full output is long!
pprint(roi_json['features'][0]['geometry']['coordinates'][0][0:10])
```

```
[[-105.08669315719459, 41.81747353490203],
 [-105.12647954793552, 41.819996817840824],
 [-105.14733301403585, 41.821369666176125],
 [-105.18923395737376, 41.807798034351976],
```

```
[-105.24321303184018, 41.771571878769855],
[-105.26291012100836, 41.78519856399359],
[-105.28731840106028, 41.77917046505193],
[-105.29891733873183, 41.79071058205539],
[-105.30166992435787, 41.811543641132005],
[-105.30989321696639, 41.830431875672 55]]
```

```
# Apply the zip function (note the use of *
# Recall that lon is first and lat second in the geojson file
# this is because coordinates are stored as x,y pairs (lon is x, and lat is y)

lon,lat = zip(*roi_json['features'][0]['geometry']['coordinates'][0])
```

```
# Display vector map to ensure everything looks good.
plt.figure()
plt.plot(lon, lat, color='k', linewidth=1)
plt.title('Ogallala Aquifer')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.axis('equal')
plt.show()
```



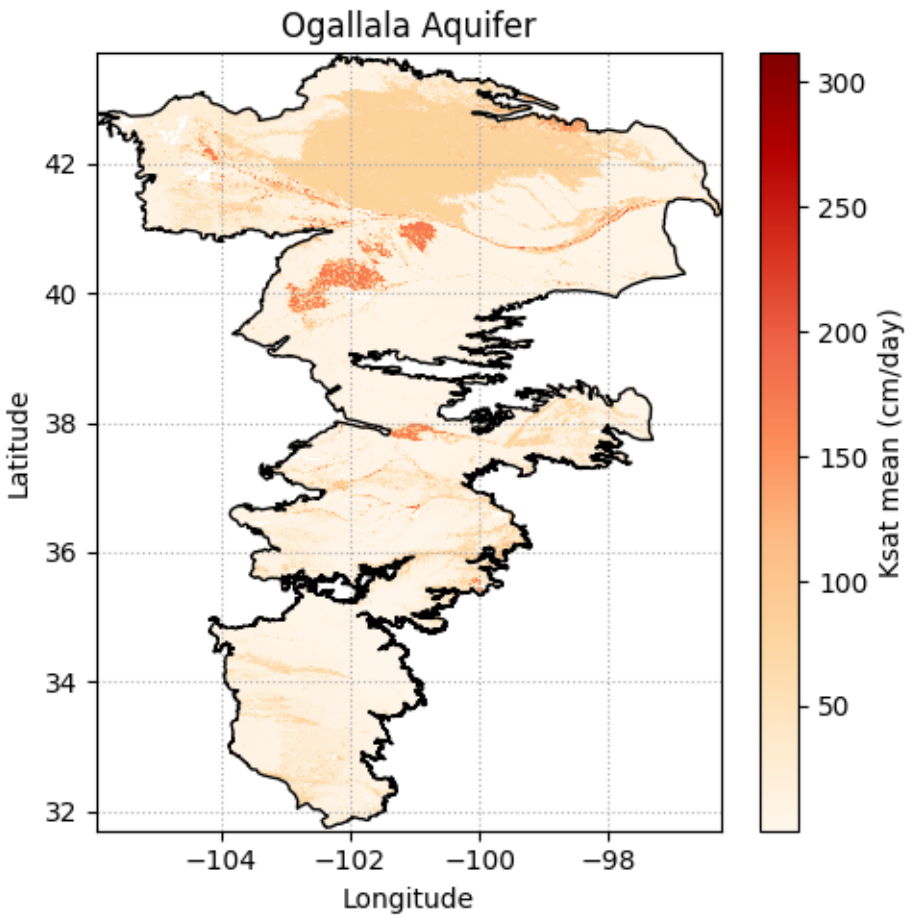
9.0.0.4 Load GEE datasets

For this tutorial we will load the mean saturated hydraulic conductivity layer from the USDA-NRCS Soil Survey GeoDatabase (SSURGO, see Walkinshaw et al. 2020). Since the saturated hydraulic conductivity represents the permeablity of the soil under saturated conditions, this layer will reveal places where the potential for aquifer recharge is greatest. Note the word "potential", since soils in this region are rarely saturated due to the low annual precipitation regime.

```
# Load GEE layers
ksat = ee.Image('projects/earthengine-legacy/assets/projects/sat-io/open-datasets/CSRL_soi

# Clip image to region of interest
factor = 1/(10e4) * 86400 # To convert micrometers/second to cm/day
ksat = ksat.select('b1').multiply(factor).clip(roi_geom).mask(mask)
```

```
# Note that this cell will take a few seconds, the area we are requesting
# is large and has a complex shape

# Get map from url
image_url = ksat.getDownloadUrl({
    'region': roi_geom,
    'scale':800,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF'})

# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)
```

```
# Check if the request was successful
if response.status_code == 200:

    # Read image data into a BytesIO object
    image_data = io.BytesIO(response.content)

    # Use Xarray to open the raster image directly from memory
    raster = xr.open_dataarray(image_data, engine='rasterio').squeeze()

    raster.plot.imshow(figsize=(5,5) , cmap='OrRd', add_colorbar=True,
                       cbar_kwargs={'label':'Ksat mean (cm/day)'})
    plt.plot(lon, lat, color='k', linewidth=1)

    plt.title('Ogallala Aquifer')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.axis('equal')
    plt.tight_layout()
    plt.grid(linestyle=':')
    plt.show()
```



9.0.0.5 Add state boundaries to map

This part requires using the GeoPandas library to clip and overlay the map of U.S. states on the aquifer map.

```
import geopandas as gpd
```

```
roi_geom.bounds().getInfo()
```

```
{'geodesic': False,
 'type': 'Polygon',
 'coordinates': [[[-105.92127815341775, 31.74341518646195],
   [-96.25860479316862, 31.74341518646195],
   [-96.25860479316862, 43.66377283318694],
```

```
[-105.92127815341775, 43.66377283318694],
[-105.92127815341775, 31.743415186461 95]]]}
```

```
# Read US states that overlap the region of interest
# Note that filterBounds() is not the same as clip()
states = ee.FeatureCollection("TIGER/2018/States").filterBounds(roi_geom)

## Get the bounding box of the region of interest
bbox = roi_geom.bounds()

## Create a rectangle geometry from the bounding box
rectangle = ee.Geometry(bbox)

# Define function to intersect and simplify each feature (to download less data)
clipping_fun = lambda feature: feature.intersection(rectangle, maxError=100).simplify(maxE

# Apply function to all features of the FeatureCollection
# This line can take several seconds
states_clipped = states.map(clipping_fun).getInfo()
```

Plotting the clipped layer of U.S. states can be accomplished using two methods: 1) using a regular for loop that iterates over each feature or 2) simply using the GeoPandas module to automatically convert all the features into a GeoDataFrame. The latter option makes things easier since GeoPandas will handle all the heavy lifting.

Iterating over each figure using a for loop would require extracting the coordinates of each geometry. However, because we clipped the features using the bounding box of the region, a complicating factor is that not all the resulting clipped state geometries consist of a single polygon. After clipping, a few states may consist of a Polygon and one or more LineStrings, which requires grouping and nesting using a GeometryCollection, and we would need to handle this during the loop.

Let's look at this problem quickly and then we will load the GeoPandas module to avoid handling this manually.

```
# Find out how many states we have
print('There are', len(states_clipped['features']), 'features in this layer')
print('') # Add an empty line for clarity

for f in states_clipped['features']:
    print(f['geometry']['type'])

# Remove comment from line below to examine one of the GeometryCollections
```

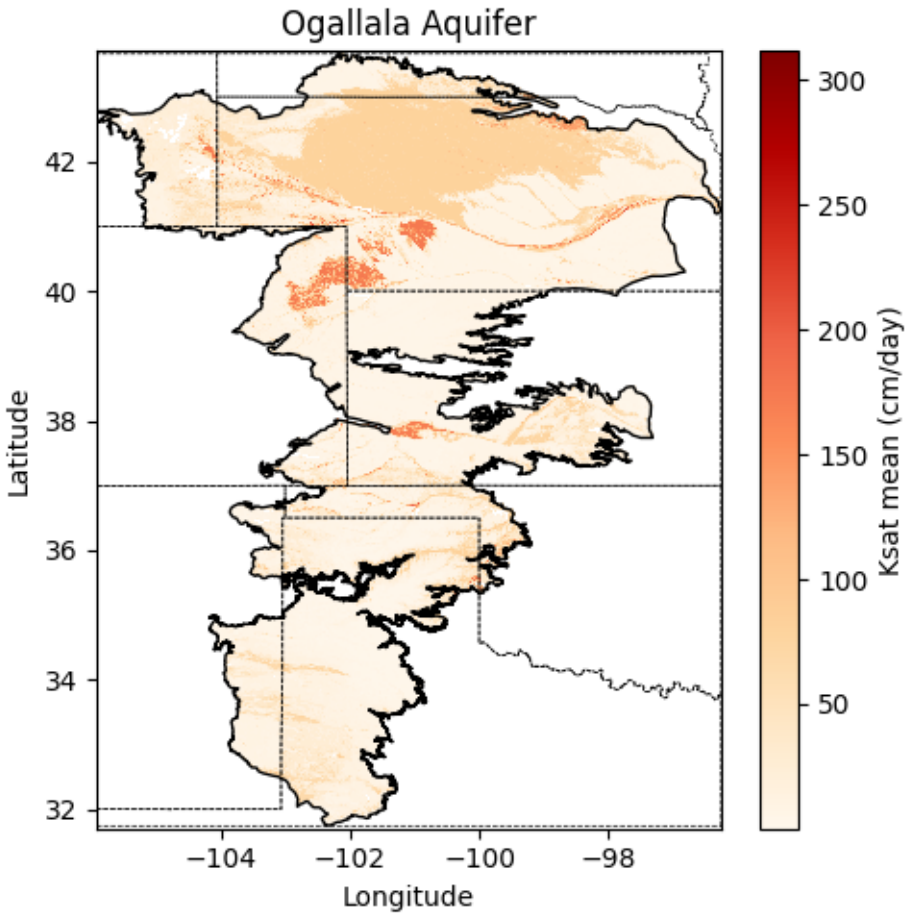
```
#print(states_clipped['features'][2]['geometry'])
```

```
There are 8 features in this layer

Polygon
Polygon
GeometryCollection
GeometryCollection
GeometryCollection
Polygon
Polygon
Polygon
```

```
# Convert region of interest from json to GeoDataframe
states_gdf = gpd.GeoDataFrame.from_features(states_clipped)
```

```
# Check if the request was successful
if response.status_code == 200:

    # Read image data into a BytesIO object
    image_data = io.BytesIO(response.content)

    # Use Xarray to open the raster image directly from memory
    raster = xr.open_dataarray(image_data, engine='rasterio').squeeze()

    plt.figure(figsize=(5,5))
    raster.plot.imshow(cmap='OrRd', add_colorbar=True,
                       cbar_kwargs={'label':'Ksat mean (cm/day)'})
    plt.plot(lon, lat, color='k', linewidth=1)

    # Mute this line to avoid
    states_gdf.plot(ax=plt.gca(),
                            facecolor='None',
                            edgecolor='k',
                           linewidth=0.5, linestyle='--')

    plt.title('Ogallala Aquifer')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.axis('equal')
```

```
plt.tight_layout()
plt.show()
```



9.1 References

- Walkinshaw, M., O'Geen, A., & Beaudette, D. (2021). Soil properties. California Soil Resource Lab.

10 Band computations

Active and passive remote sensors onboard orbitting satellites usually collect multiple spectral bands. The combination of several of these bands can often be used to characterize land surface features, like vegetation, water bodies, and wildfires.

In this example we will use the red and near infrared bands from Sentinel 2 satellite of the European Space Agency to compute the normalized difference vegetation index at 10-meter spatial resolution for a production field in Argentina.

```
# Import modules
import ee
import json
import requests
import matplotlib.pyplot as plt
from matplotlib import colors
import xarray as xr
import pandas as pd
```

```
# Authenticate
#ee.Authenticate()

# Initialize API
ee.Initialize()
```

```
# Import boundary for area of interest (aoi)
with open('../datasets/field_bnd_carmen.geojson') as file:
    aoi_json = json.load(file)

# Define the ee.Geometry
aoi = ee.Geometry(aoi_json['features'][0]['geometry'])

# Create mask for field
mask = ee.Image.constant(1).clip(aoi).mask()
```

```
# Define start and end dates
# Summer for southern hemisphere, matching the soybean growing season
start_date = '2023-01-10'
end_date = '2023-01-30'
```

```
# Load Sentinel-2 image collection
S2 = ee.ImageCollection('COPERNICUS/S2') \
                .filterDate(start_date, end_date) \
                .filterBounds(aoi) \
                .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 20)) \
                .select(['B8', 'B4'])  # B8 is NIR, B4 is Red
```

```
# Print the following line to explore the selected images
#S2.getInfo()['features']
```

```
def calculate_ndvi(image):
    """
    Function to calculate NDVI for a single image.
    """
    ndvi = image.normalizedDifference(['B8', 'B4']).rename('NDVI')
    return image.addBands(ndvi)

# Apply the NDVI function to each image in the collection
ndvi_collection = S2.map(calculate_ndvi)
```

```
# Apply the function to add mean NDVI as a property
ndvi_mean = ndvi_collection.reduce(ee.Reducer.mean())

# Mask ndvi image
ndvi_mean = ndvi_mean.mask(mask)
```

```
# Inspect resulting image (note the the new band is "NDVI_mean"
ndvi_mean.select('NDVI_mean').getInfo()
```

```
{'type': 'Image',
 'bands': [{'id': 'NDVI_mean',
   'data_type': {'type': 'PixelType',
    'precision': 'float',
    'min': -1,
    'max': 1},
```

```
 'crs': 'EPSG:4326',
 'crs_transform': [1, 0, 0, 0, 1, 0]}]}
```

```
# Get url link for image
image_url = ndvi_mean.getDownloadUrl({'region': aoi,'scale':10,
                                      'bands':['NDVI_mean'],
                                     'crs': 'EPSG:4326',
                                     'format': 'GEO_TIFF'})

# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)

# Save geotiff image to local drive
filename = '../outputs/field_carmen_ndvi.tif'
with open(filename, 'wb') as f:
    f.write(response.content)
```

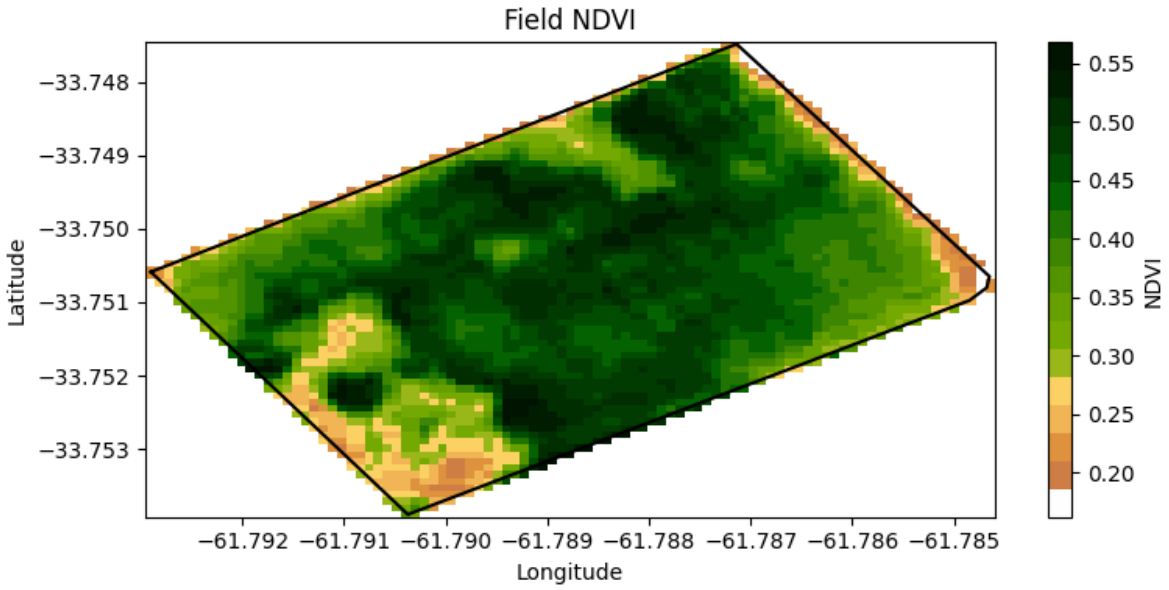
```
# Read saved geotiff image from local drive
ndvi_raster = xr.open_dataarray(filename).squeeze()
```

```
df = pd.DataFrame(aoi_json['features'][0]['geometry']['coordinates'][0])
df.columns = ['lon','lat']
df.head(3)
```

	lon	lat
0	-61.792900	-33.750588
1	-61.790372	-33.753893
2	-61.784847	-33.750976

```
# Create colormap
hex_palette = ['#FEFEFE','#CE7E45', '#DF923D', '#F1B555', '#FCD163', '#99B718', '#74A901',
               '#66A000', '#529400', '#3E8601', '#207401', '#056201', '#004C00', '#023B01',
               '#012E01', '#011D01', '#011301']

# Use the built-in ListedColormap function to do the conversion
rgb_cmap = colors.ListedColormap(hex_palette)
```

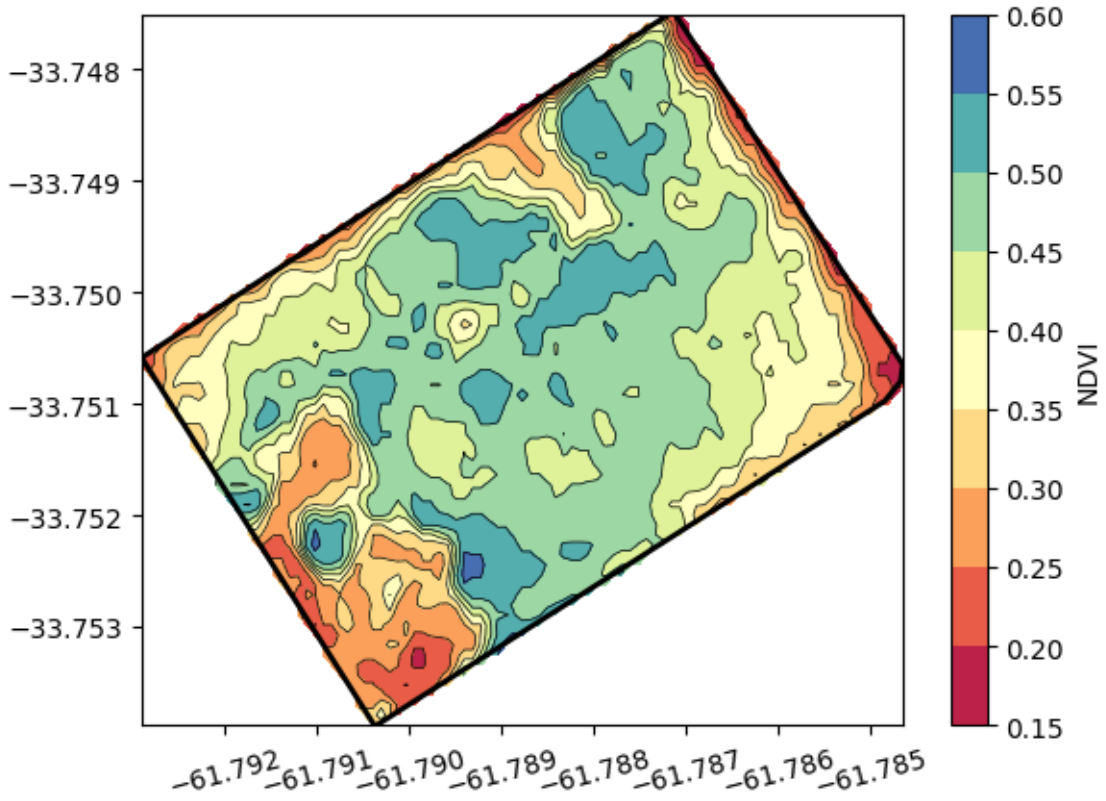
```
# Create figure
ndvi_raster.plot.imshow(figsize=(8,4), cmap=rgb_cmap, add_colorbar=True,
                    cbar_kwargs={'label':'NDVI'})

plt.plot(df['lon'], df['lat'],'-k')
plt.title('Field NDVI')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.tight_layout()
plt.show()
```



We can also get the NDVI values and corresponding coordinates to create a contour plot to delineate areas of higher and lower vegetation values.

```
# Give arrays a shorter name
X = ndvi_raster.coords['x']
Y = ndvi_raster.coords['y']
Z = ndvi_raster.values

plt.figure()
plt.plot(df['lon'], df['lat'],'-k', linewidth=2)
plt.contour(X, Y, Z, levels=8, linewidths=0.5, colors='k')
plt.contourf(X, Y, Z, levels=8, cmap='Spectral')
```

```
plt.colorbar(label='NDVI')
plt.xticks(rotation=15)
plt.show()
```



11 Animate image collection

When working with spatial data in the form of images spanning a period of time, animations are an effective way to showcase temporal and spatial changes across a region. Similar to a video, animations are composed of sequential frames and Google Earth Engine provides tools for creating and retrieving animations via URL. Alternatively, individual images can be downloaded for local processing, that enable the creation of animations using plotting libraries like Matplotlib. This tutorial focuses on the former approach, exploring how to leverage Google Earth Engine for creating an animation of changes in annual vegetation over the central U.S. Great Plains.

```
# Import and initialize GEE
import ee
import glob
import requests
import pandas as pd
from datetime import datetime, timedelta
```

```
# Authenticated
#ee.Authenticate()

# Initialize
ee.Initialize()
```

11.1 Example 1: Animation of vegetation dynamics in the cloud

11.1.0.1 Define animation area

```
# Define a rectangular region over the state of Kansas
# Rectangle coordinates: xMin, yMin, xMax, yMax
rect = ee.Geometry.Rectangle([[-102.5,36.5], [-94,40.5]]);
```

```
# Select state boundary
# For countries you can use: FAO/GAUL_SIMPLIFIED_500m/2015/level1
# A site with country codes: http://www.statoids.com/wab.html
region = ee.FeatureCollection("TIGER/2018/States").filter(ee.Filter.eq('NAME', 'Kansas'))
```

11.1.0.2 Retrieve vegetation product

```
# Select a collection from the available dataset
start_date = '2012-01-01'
end_date = '2022-01-01'
modis = ee.ImageCollection('MODIS/006/MOD13A2').filterDate(start_date, end_date)
collection = modis.select('NDVI').map(lambda img: img.clip(region))
```

11.1.0.3 Get day of the year for each image

In this step we define a function that is applied to each image of the collection using the `map()` method. The function computes the day of the year based on the date, and adds this variable to each image as a property. Since each image of the collection remains the same, except that each image now includes the day of the year (`doy`), we overwrite the collection with the updated version of itself.

```
# Define function
def get_doy(img):
    """
    Function that finds and adds the day of the year
    to each image in the collection.
    """
    doy = ee.Date(img.get('system:time_start')).getRelative('day', 'year')
    return img.set('doy', doy)

# Apply the function to each image of the collection using the .map() method
collection = collection.map(get_doy)

# The `doy` is added to the properties of each image
# Use the following line to see the added property
# collection.getInfo()
```

```
# Filter the complete collection to a single year of data e.g. 2021.
# We use one year as a dummy variable to compute the day of the year.
unique_doy = collection.filterDate('2021-01-01', '2022-01-01')
```

```
# Define a filter that identifies which images from the complete collection
# match the DOY of the unique DOY variable.
# leftField == rightField
filt = ee.Filter.equals(leftField='doy', rightField='doy')
```

```
# Define a join.
join = ee.Join.saveAll('doy_matches')

# Apply the join and convert the resulting FeatureCollection to an ImageCollection.
join_col = ee.ImageCollection(join.apply(unique_doy, collection, filt))
```

11.1.0.4 Reduce all images for a given DOY pixel-wise.

```
# Define median reducer for images of the same DOY
def apply_median(img):
    """
    Function that computes the pixel-wise median
    for all images matching a given DOY
    """
    doy_col = ee.ImageCollection.fromImages(img.get('doy_matches'))
    return doy_col.reduce(ee.Reducer.median()).multiply(0.0001)

# Apply function for each DOY and for all images matching a given DOY
composite = join_col.map(apply_median)
```

11.1.0.5 Create animation

```
# Define function that handles the visuals (paint adds the boundary,
# which is a assigned a value (-0.1) relative to the other pixels (so a low value means re
def animate(img):
    cmap = ['black','FFFFFF', 'CE7E45', 'DF923D', 'F1B555', 'FCD163', '99B718', '74A901',
        '66A000', '529400', '3E8601', '207401', '056201', '004C00', '023B01',
        '012E01', '011D01', '011301']
    frame = img.paint(region, -0.1, 2).visualize(min=-0.1, max=0.8, palette=cmap)
```

```
    return frame

# Map the function to each image
animation = composite.map(animate)
```

```
# Animation options
animationOptions = {'region': rect,     # Selected region on the map
                    'dimensions': 600,    # Size of the animation
                    'crs': 'EPSG:3857',   # Coordinate reference system
                    'framesPerSecond': 6  # Animation speed
}
```

```
# Render the GIF animation in the console.
print(animation.getVideoThumbURL(animationOptions))
```

```
https://earthengine.googleapis.com/v1alpha/projects/earthengine-legacy/videoThumbnails/dab15
```

```
# Right click on the generated GIF image in the browser and select "save image as" to down
```

11.2 Example 2: Animation of vegetation dynamics in local disk

This option requires downloading the images to the local drive and creating the animation ourselves, but it provides with the greatest flexibilty to edit the resulting animation.

```
# Import additional modules
import xarray as xr
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import colors
```

```
# Define function to save images to the local drive
def save_geotiff(ee_image, filename, crs, scale, geom, bands=[]):
    """
    Function to save images from Google Earth Engine into local hard drive.
    """
    image_url = ee_image.getDownloadUrl({'region': geom,'scale':scale,
                                         'bands': bands,
                                         'crs': f'EPSG:{crs}',
```

```
                                        'format': 'GEO_TIFF'})

    # Request data using URL and save data as a new GeoTiff file
    response = requests.get(image_url)
    with open(filename, 'wb') as f:
        f.write(response.content)
        return print(f"Saved image {filename}")
```

```
# Select
region = ee.FeatureCollection("TIGER/2018/States").filter(ee.Filter.eq('NAME', 'Kansas'))

# Create mask
mask = ee.Image.constant(1).clip(region).mask()
```

```
# Define the time range
start_date = '2022-01-01'
end_date = '2022-12-31'

# Select MODIS Terra Vegetation Indices 16-Day Global 1km
modis = ee.ImageCollection("MODIS/061/MOD13A2").filterDate(start_date, end_date)
collection = modis.select('NDVI')

# Get the list of available image dates
get_date = lambda image: ee.Image(image).date().format('YYYY-MM-dd')

dates = collection.toList(collection.size()).map(get_date).getInfo()
print(dates)
```

```
['2022-01-01', '2022-01-17', '2022-02-02', '2022-02-18', '2022-03-06', '2022-03-22', '2022-04
```

```
gif_folder = '../outputs/ndvi_gif_files'
if not glob.os.path.isdir(gif_folder):
    glob.os.mkdir(gif_folder)
```

```
for date in dates:
    start_date = date
    end_date = (datetime.strptime(start_date, '%Y-%m-%d') + timedelta(days=1)).strftime('%
    ndvi_img = ee.ImageCollection('MODIS/006/MOD13A2').filterDate(start_date, end_date).fi
    ndvi_img = ndvi_img.multiply(0.0001).clip(region).mask(mask)
```

```
    filename = f"{gif_folder}/ndvi_{date}.tiff"
    try:
        save_geotiff(ndvi_img, filename, crs=4326, scale=1000, geom=region.geometry(), ban
    except:
        print(f"Trouble loading image {filename}. Skipping this image.")
```

```
Saved image ../outputs/ndvi_gif_files/ndvi_2022-01-01.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-01-17.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-02-02.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-02-18.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-03-06.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-03-22.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-04-07.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-04-23.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-05-09.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-05-25.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-06-10.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-06-26.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-07-12.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-07-28.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-08-13.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-08-29.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-09-14.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-09-30.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-10-16.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-11-01.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-11-17.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-12-03.tiff
Saved image ../outputs/ndvi_gif_files/ndvi_2022-12-19.tiff
```

> **Note**
>
> In the method `.filterDate(start_date, end_date)` the start date is inclusive, but the end date is exclusive.

```
# Read the list of images
images = glob.glob(f"{gif_folder}/*.tiff")
images.sort()
```

```
# Paletter of colors for the Enhanced Vegetation Index
hex_palette = ['#FF69B4','#CE7E45', '#DF923D', '#F1B555', '#FCD163', '#99B718', '#74A901',
               '#66A000', '#529400', '#3E8601', '#207401', '#056201', '#004C00', '#023B01',
               '#012E01', '#011D01', '#011301']

# Use the built-in ListedColormap function to do the conversion
cmap = colors.ListedColormap(hex_palette)
```

Colormap note

The first color of the palette is hot pink ('#FF69B4'). The color was added to represent the lowest NDVI values, which are typically caused by snow on the ground during winter months.

```
# Create figure
fig, ax = plt.subplots(figsize=(6,3))

# Leave a bit more room at the bottom to avoid cutting the xlabel
fig.subplots_adjust(bottom=0.15)


# Create figure with axes and colorbar, which will remain fixed.
raster = xr.open_dataarray(images[0]).squeeze()
raster.plot.imshow(ax=ax, cmap=cmap, add_colorbar=True,
                   cbar_kwargs={'label':'NDVI'},vmin=0, vmax=0.8)

def animate(index):
    """
    Function that creates each frame.
    """

    # Read geotiff image with xarray
    raster = xr.open_dataarray(images[index]).squeeze()

    # Clear axes and draw new objects (without colorbar)
    # Force vmin and vmax to keep the same range of values as the colorbar
    ax.clear()
    raster.plot.imshow(ax=ax, cmap=cmap, add_colorbar=False, vmin=0, vmax=0.8)
    ax.set_title(images[index][-15:-5])
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
```

```
    plt.tight_layout()

    return ax

# Avoid displaying the first figure
plt.close()

# Save animation as .gif
ani = animation.FuncAnimation(fig, animate, len(images),interval=250)
ani.save('../outputs/ndvi_animation.gif', writer='pillow')
```

```
<Figure size 640x480 with 0 Axes>
```

Here is the resulting gif. Note that during the winter the image occasionally shows some areas with snow on the ground (look for reddish patches). You can display it in your notebook using the following html code:

```
<img src="../outputs/ndvi_animation.gif" alt="drawing" width="650"/>
```

11.3 Example 3: Soil moisture dynamics

```
# Since the product is available every 3 hours, define one month only
# to avoid running hitting the GEE memory limit
start_date = '2023-01-01'
end_date = '2023-01-31'

# Select SMAP Level 3 layer at 9-km spatial resolution
smap = ee.ImageCollection('NASA/SMAP/SPL4SMGP/007')

# Get the list of available image dates
get_date = lambda image: ee.Image(image).date().format('YYYY-MM-dd HH:mm:SS')

collection = smap.filterDate(start_date, end_date)
dates = collection.toList(collection.size()).map(get_date).getInfo()
print(len(dates))
print(dates[0:10])
```

```
240
['2023-01-01 01:30:00', '2023-01-01 04:30:00', '2023-01-01 07:30:00', '2023-01-01 10:30:00',
```

```
smap_gif_folder = '../outputs/smap_gif_files'
if not glob.os.path.isdir(smap_gif_folder):
    glob.os.mkdir(smap_gif_folder)
```

```
# Use pandas to create range of dates
dates = pd.date_range('2023-01-01', '2023-12-31', freq='7D')
dates
```

```
DatetimeIndex(['2023-01-01', '2023-01-08', '2023-01-15', '2023-01-22',
               '2023-01-29', '2023-02-05', '2023-02-12', '2023-02-19',
               '2023-02-26', '2023-03-05', '2023-03-12', '2023-03-19',
               '2023-03-26', '2023-04-02', '2023-04-09', '2023-04-16',
               '2023-04-23', '2023-04-30', '2023-05-07', '2023-05-14',
               '2023-05-21', '2023-05-28', '2023-06-04', '2023-06-11',
               '2023-06-18', '2023-06-25', '2023-07-02', '2023-07-09',
               '2023-07-16', '2023-07-23', '2023-07-30', '2023-08-06',
               '2023-08-13', '2023-08-20', '2023-08-27', '2023-09-03',
               '2023-09-10', '2023-09-17', '2023-09-24', '2023-10-01',
               '2023-10-08', '2023-10-15', '2023-10-22', '2023-10-29',
               '2023-11-05', '2023-11-12', '2023-11-19', '2023-11-26',
               '2023-12-03', '2023-12-10', '2023-12-17', '2023-12-24',
               '2023-12-31'],
              dtype='datetime64[ns]', freq='7D')
```

```
# Only use weekly moisture levels to avoid retrieving tons of images

for date in dates:
    start_date = date.strftime('%Y-%m-%d')
    end_date = (date + pd.Timedelta('1D')).strftime('%Y-%m-%d')

    # Request data and create average of all images for that day
    smap_img = smap.filterDate(start_date, end_date) \
                   .reduce(ee.Reducer.mean()).multiply(100).clip(region).mask(mask)

    try:
        # Creaoutput file name
        filename = f"{smap_gif_folder}/smap_{start_date}.tiff"

        # Note that the band name has `mean` appended since that is the reducer we used
        # I also donwscaled the map to 1 km resolution from 9 km
```

```
        save_geotiff(smap_img, filename, crs=4326, scale=9000,
                     geom=region.geometry(), bands=['sm_profile_mean'])

    except:
        print(f"Trouble loading image {filename}. Skipping this image.")
```

> **Note**
>
> In the method `.filterDate(start_date, end_date)` the start date is inclusive, but the end date is exclusive.

```
# Read the list of images
images = glob.glob(f"{smap_gif_folder}/*.tiff")
images.sort()
```

```
# Define colormap
cmap = 'Spectral'

# Create figure
fig, ax = plt.subplots(figsize=(6,3))

# Leave a bit more room at the bottom to avoid cutting the xlabel
fig.subplots_adjust(bottom=0.15)

# Create figure with axes and colorbar, which will remain fixed.
raster = xr.open_dataarray(images[0]).squeeze()
raster.plot.imshow(ax=ax, cmap=cmap, add_colorbar=True,
                   cbar_kwargs={'label':'VWC (%)'}, vmin=0, vmax=40)

def animate(index):
    """
    Function that creates each frame.
    """

    # Read geotiff image with xarray
    raster = xr.open_dataarray(images[index]).squeeze()

    # Clear axes and draw new objects (without colorbar)
    # Force vmin and vmax to keep the same range of values as the colorbar
    ax.clear()
```

```
    raster.plot.imshow(ax=ax, cmap=cmap, add_colorbar=False, vmin=0, vmax=40)
    ax.set_title(images[index][-15:-5])
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
    plt.tight_layout()

    return ax

# Avoid displaying the first figure
plt.close()

# Save animation as .gif
ani = animation.FuncAnimation(fig, animate, len(images), interval=250)
ani.save('../outputs/smap_animation.gif', writer='pillow')
```

```
<Figure size 640x480 with 0 Axes>
```

12 Integrate with GeoPandas

Combining remote sensing data from Google Earth Engine (GEE) with the spatial data processing capabilities of GeoPandas opens up a new avenue for advanced geographic analyses. GEE provides access to a comprehensive archive of satellite imagery and geospatial datasets, while GeoPandas extends the popular pandas library to allow for efficient operations of vector data. By leveraging the strengths of both platforms, researchers can perform sophisticated spatial operations, from overlay analyses to spatial joins, enhancing the possibilities of geospatial analysis and depth of research questions. To illustrate the integration of GEE with GeoPandas, we will study the extend of major habitat fragmentation.

12.1 Habitat fragmentation problem

The Flint Hills is a region is one of the last remaining tallgrass prairies in North America. This unique ecosystem is home to a rich biodiversity, including numerous grassland bird species, diverse plant life, and a variety of other wildlife. The Flint Hills have resisted the plow thanks to their rocky terrain, preserving an expanse of native grasslands that once covered much of the Great Plains. This area not only offers a glimpse into America's natural heritage but also serves as a critical refuge for species that thrive in the prairie habitat.

Habitat fragmentation is a process where large continuous habitats are divided into smaller, isolated sections. Habitat fragmentation is a significant ecological concern, particularly in regions like the Flint Hills. Roads, one of the primary drivers of such fragmentation, can disrupt the interconnected landscapes, limiting animal movement, altering animal behavior, and increasing vulnerability to external threats like invasive species and climate change. In the Flint Hills, the expansion of road networks could potentially threaten the integrity of this pristine habitat.

In this exercise we explore some of the fragmentation caused by primary and secondary roads.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap, BoundaryNorm
import geopandas as gpd
import requests
```

```
import xarray as xr
import ee
```

```
# Authenticate GEE
#ee.Authenticate()

# Initialize GEE
ee.Initialize()
```

```
# Define CRS
utm14 = 32614 # UTM 14 projected coordinates in meters (good for Kansas)
wgs84 = 4326 # WGS84 geographic coordinates
```

12.2 Load boundary of Flint Hills ecological region

```
# Load eco regions
eco_regions = ee.FeatureCollection("RESOLVE/ECOREGIONS/2017")

# Use this line to inspect properties
#eco_regions.first().propertyNames().getInfo()
```

```
# Select flint hills region
# Find ecoregion ID using their website: https://ecoregions.appspot.com
region = eco_regions.filter(ee.Filter.inList('ECO_ID',[392]))
region_data = region.getInfo()
```

```
# Convert to GeoDataframe
gdf_region = gpd.GeoDataFrame.from_features(region_data['features'], crs=wgs84)
gdf_region['geometry'] = gdf_region['geometry'].simplify(0.001)
gdf_region.head()
```

	geometry	BIOME_NAME	BI
0	POLYGON ((-97.36772 38.37556, -97.36465 38.359...	Temperate Grasslands, Savannas & Shrublands	8

12.3 Load primary and secondary roads

```
# Load TIGER roads dataset
roads_dataset = ee.FeatureCollection('TIGER/2016/Roads').filterBounds(region)
selected_roads = roads_dataset.filter(ee.Filter.inList('mtfcc',['S1100','S1200']))
roads_data = selected_roads.getInfo()

# Use this line to see available properties
# https://www2.census.gov/geo/pdfs/reference/mtfccs2017.pdf
# roads_dataset.first().propertyNames().getInfo()
```

```
# Convert Geodataframe
gdf_roads = gpd.GeoDataFrame.from_features(roads_data['features'], crs=wgs84)
gdf_roads.head()
```

	geometry	fullname	linearid	mtfcc	rttyp
0	LINESTRING (-96.64559 39.84213, -96.64558 39.8...	US Hwy 77	1103942688564	S1200	U
1	LINESTRING (-96.66120 36.78194, -96.66182 36.7...	1st St	110788715400	S1200	M
2	LINESTRING (-96.65110 36.94044, -96.65108 36.9...	Hwy 18	1103942486000	S1200	M
3	LINESTRING (-96.65181 36.93269, -96.65178 36.9...	Broadway	110788721819	S1200	M
4	LINESTRING (-96.65110 36.94044, -96.65108 36.9...	Broadway	1103942485999	S1200	M

```
# Show map
fig, ax = plt.subplots(1,1, figsize=(3,6))
gdf_region.plot(ax=ax, facecolor='None')
gdf_roads.plot(ax=ax, color='tomato')
```

12.4 Unify roads into single Multiline feature

```
unified_roads = gdf_roads.unary_union
gdf_unified_roads = gpd.GeoDataFrame(geometry=[unified_roads], crs=gdf_roads.crs)
gdf_unified_roads['geometry'] = gdf_unified_roads['geometry'].simplify(0.001)
gdf_unified_roads.head()
```

	geometry
0	MULTILINESTRING ((-96.64559 39.84213, -96.6435...

12.5 Create buffer area around roads

This step is needed to be able to compute intersection and difference with the boundary map. We will assign a buffer of 1 km on each side of each road to create clearly distinct polygons. If you work in smaller areas, a smaller buffer would be better. In this dataset, a couple of kilometers represents a small magnitude compared to the extent of the region.

```
# Create buffer as a GeoSeries
buffer_distance = 1000  # Example buffer distance
buffered_unified_roads = gdf_unified_roads.to_crs(utm14).buffer(buffer_distance).to_crs(gd
```

```
# Create a new GeoDataFrame with the buffered unified roads geometry
gdf_buffered_unified_roads = gpd.GeoDataFrame(geometry=gpd.GeoSeries(buffered_unified_road
                                              crs=gdf_roads.crs)

gdf_buffered_unified_roads.head()
```

	geometry
0	MULTIPOLYGON (((-97.36906 38.91970, -97.37023 ...

12.6 Obtained fragmented areas

The key operation in this step is to compute the ‘difference between layers.

```
# Obtained fragmented areas
fragmented_regions = gpd.overlay(gdf_region, gdf_buffered_unified_roads, how='difference')
fragmented_regions.head()
```

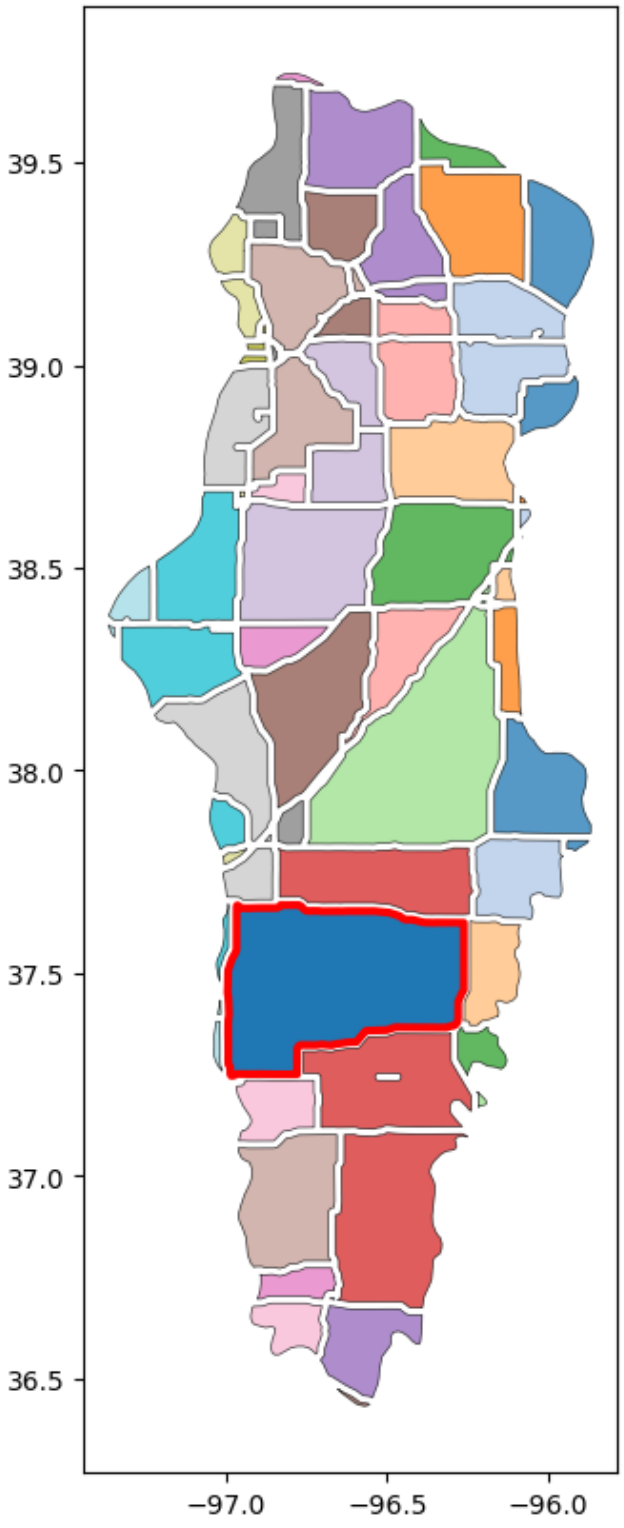
	geometry	BIOME_NAME
0	MULTIPOLYGON (((-95.99115 39.43608, -95.91121 ...	Temperate Grasslands, Savannas & Shrublands

```
# Explode the multipolygon feature
gdf_patches = fragmented_regions.explode(index_parts=True).reset_index(drop=True)
gdf_patches.head()
```

	BIOME_NAME	BIOME_NUM	COLOR	COLOR_BIO	COLOR_N
0	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
1	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
2	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
3	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
4	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23

```
# Compute area and find patch with largest area
gdf_patches['area'] = gdf_patches.to_crs(utm14).area
idx_largest_area = gdf_patches['area'].argmax()
```

```
# Plot fragmented areas
fig, ax = plt.subplots(figsize=(10, 10))
gdf_patches.plot(ax=ax, alpha=0.75, linewidth=0.5, cmap='tab20', edgecolor='black')
gdf_patches.loc[[idx_largest_area],'geometry'].plot(ax=ax, edgecolor='red', linewidth=3)

plt.show()
```


39.5
39.0
38.5
38.0
37.5
37.0
36.5
−97.0
−96.5
−96.0

12.7 Load cropland datalayer

This layer will help us understand which polygons are dominated by cropland and which polygons are still dominated by grassland vegetation

```
# Land use
cdl = ee.ImageCollection('USDA/NASS/CDL')
cdl_layer = cdl.filter(ee.Filter.date('2020-01-01', '2021-12-31')).first().clip(region)
cropland = cdl_layer.select('cropland')

# Single-band GeoTIFF files wrapped in a zip file.
url = cropland.getDownloadUrl({
    'region': region.geometry(),
    'scale':100,
    'format': 'GEO_TIFF'
})


# Request data using URL and save data as a new GeoTiff file
response = requests.get(url)
with open('../datasets/spatial/cropland.tif', 'wb') as f:
    f.write(response.content)
```

```
# Read GeoTiff file using the Xarray package
raster = xr.open_dataarray('../datasets/spatial/cropland.tif').squeeze()
raster = raster.rio.reproject(wgs84)
```

```
# Examine raster file
raster
```

```
<xarray.DataArray 'band_data' (y: 3560, x: 1658)>
array([[ nan,  nan,  nan, ...,  37.,  37., 176.],
       [ nan,  nan,  nan, ..., 111., 111.,   5.],
       [ nan,  nan,  nan, ...,  37.,  37.,   5.],
       ...,
       [ nan,  nan,  nan, ...,  nan,  nan,  nan],
       [ nan,  nan,  nan, ...,  nan,  nan,  nan],
       [ nan,  nan,  nan, ...,  nan,  nan,  nan]], dtype=float32)
Coordinates:
  * x            (x) float64 -97.39 -97.39 -97.39 ... -95.86 -95.86 -95.86
  * y            (y) float64 39.72 39.72 39.72 39.72 ... 36.43 36.43 36.43 36.43
```

```
    band         int64 1
    spatial_ref  int64 0
Attributes:
    AREA_OR_POINT:           Area
    TIFFTAG_RESOLUTIONUNIT:  1 (unitless)
    TIFFTAG_XRESOLUTION:     1
    TIFFTAG_YRESOLUTION:     1
```

12.8 Define custom colormap

```
# Define your labels and their corresponding colors
info = cropland.getInfo()

class_names = info['properties']['cropland_class_names']
class_values = info['properties']['cropland_class_values']
class_colors = info['properties']['cropland_class_palette']
class_colors = ['#'+c for c in class_colors]
```

```
# Create the colormap
cmap = ListedColormap(class_colors)

# Create a norm with boundaries
norm = BoundaryNorm(class_values, cmap.N)
```

12.9 Clip and save cropland map for each polygon

In this step we are clipping the cropland datalayer for each polygon and then saving that clipped and masked image into the GeoDataframe.

```
# Check that the CRS are the same before clipping the layer
gdf_patches.crs == raster.rio.crs
```

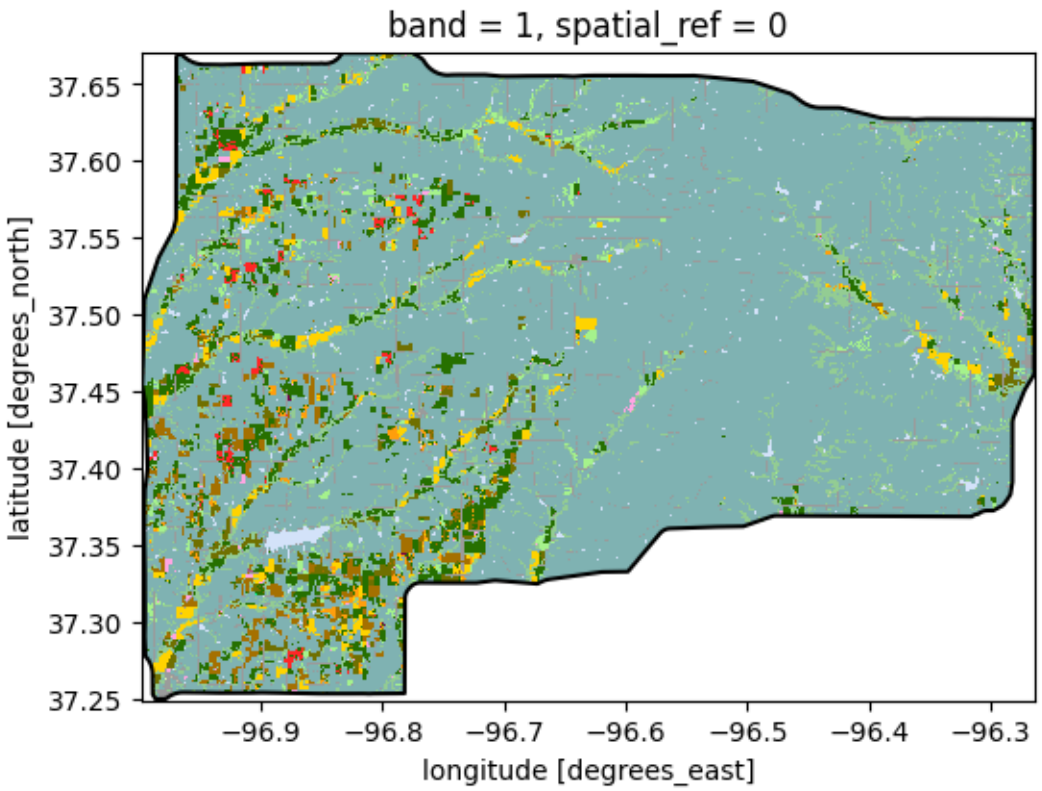
```
True
```

```
# Define a function to clip the raster with each polygon and return a numpy array
def clip_raster(polygon, raster):
```

```
    # Clip the raster with the polygon
    clipped = raster.rio.clip([polygon.geometry], crs=raster.rio.crs, all_touched=True)

    return clipped


# Apply the function to each row in the GeoDataFrame to create a new 'clipped_raster' colu
gdf_patches['clipped_raster'] = gdf_patches.apply(lambda row: clip_raster(row, raster), ax

# Inspect resulting GeoDataframe
gdf_patches.head(3)
```

	BIOME_NAME	BIOME_NUM	COLOR	COLOR_BIO	COLOR_N
0	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
1	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23
2	Temperate Grasslands, Savannas & Shrublands	8	#A87001	#FEFF73	#EE1E23

```
# Create figure showing the cropland data layer for a selected polygon
fig, ax = plt.subplots(figsize=(6, 6))

gdf_patches.loc[[idx_largest_area],'geometry'].boundary.plot(ax=ax, edgecolor='k')
gdf_patches.loc[idx_largest_area,'clipped_raster'].plot(ax=ax, cmap=cmap,
                                                        norm=norm, add_colorbar=False)
plt.show()
```



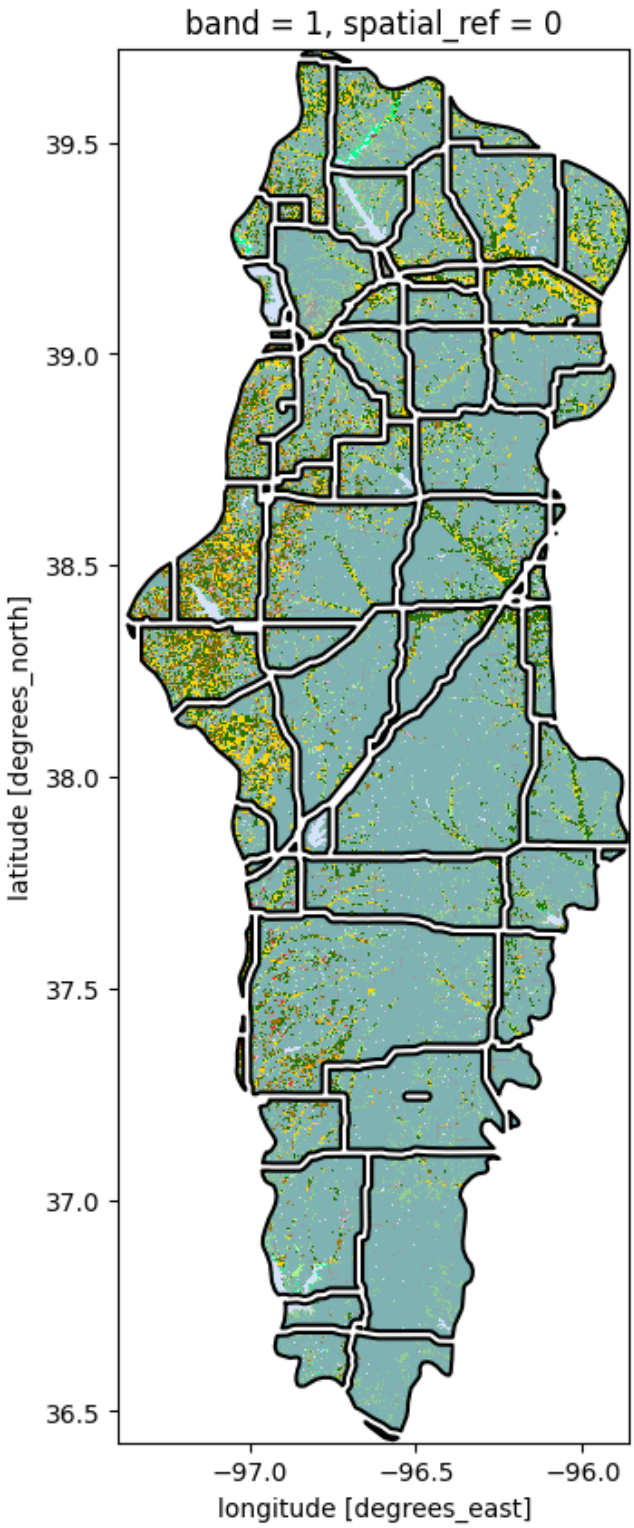
12.10 Clip all fragmentated areas to cropland data layer

This process involves rasterizing each polygon to match the spatial resolution of the cropland data layer.

```
# Define the transformation and shape from the raster metadata
transform = raster.rio.transform()
shape = (raster.rio.height, raster.rio.width)

# Rasterize the polygons
rasterized_image = rasterize(
    [(geometry, 1) for geometry in gdf_patches.geometry],
    out_shape=shape,
    fill=0,  # Background value for raster cells not covered by polygons
    transform=transform,
    all_touched=True  # Set to True if you want to include all pixels touched by geometrie
)
```

```
# Convert the rasterized image to an xarray DataArray
rasterized_da = xr.DataArray(rasterized_image, dims=("y", "x"), coords=raster.coords)

# Overlay the rasterized polygons on the raster by using the where method
# This will mask the raster outside the polygons
masked_raster = raster.where(rasterized_da == 1)
```

```
# Create figure showing the cropland data layer for all polygons
fig, ax = plt.subplots(figsize=(10, 10))
masked_raster.plot(ax=ax, cmap=cmap, norm=norm, add_colorbar=False)
gdf_patches.boundary.plot(ax=ax, edgecolor='k')
plt.show()
```


band = 1, spatial_ref = 0
39.5
39.0
38.5
38.0
37.5
37.0
36.5
latitude [degrees_north]
−97.0
−96.5
−96.0
longitude [degrees_east]

13 Unsupervised classification

Unsupervised clustering offers a powerful method for identifying patterns and categorizing data within satellite imagery, without prior labeling. Google Earth Engine (GEE) provides a robust platform for implementing these unsupervised clustering algorithms, enabling researchers to analyze and segment images based on natural groupings of spectral or spatial characteristics.

This tutorial demonstrates how to use unsupervised K-Means clustering to better understand landscapes and its inherent patterns. We will cover tutorials at the watershed level and state level using soil, climate, and landform datasets to generate regions of similar characteristics.

```
# Import modules and initialize
import ee
import json
import folium
import numpy as np
from pprint import pprint
from matplotlib import colormaps, colors
import requests
import xarray as xr
```

```
# Authenticate
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

```
# Create helper functions
def get_cmap(name,n=10):
    """
    Function to get list of colors from user-defined colormap in hex form.
    Example: get_cmap('viridis', 3)
    """
    c = colormaps[name]
    color_range = np.linspace(0,c.N-1,n).astype(int)
    cmap = [colors.rgb2hex(c(k)) for k in color_range]
```

```
    return cmap

# Get Color for different polygons
def get_polygon_color(polygon,cmap):
    color = cmap[polygon['properties']['label']]
    return {'fillColor':color, 'fillOpacity':0.8}
```

13.1 Example 1: Washed clustering

13.1.1 Load region from local file

In this tutorial we will use the Kings Creek watershed, which is fully contained withing the Konza Prairie Biological station. The watershed is dominated by a tallgrass vegetation and riparian areas surrounding the Kings Creek.

```
# Open geoJSON file and store it in a variable
with open("../datasets/kings_creek.geojson") as file:
    kings_creek = json.load(file)
```

```
# Define GEE Geometry using local file
geom = ee.Geometry(kings_creek['features'][0]['geometry'])
```

```
# Test GEE Geometry import printing the area in km^2
geom.area().getInfo()/1000000
```

```
11.353858144557279
```

13.1.2 Prepare clustering dataset

This step consists of reading images from different products, and then merging them into an image of multiple bands, where each band is one of the selected features for clustering.

```
# Import individual layers
weather = ee.Image("WORLDCLIM/V1/BIO").select(['bio01', 'bio12']).resample('bicubic')
soil = ee.Image("projects/soilgrids-isric/sand_mean").select('sand_0-5cm_mean').resample('
landforms = ee.Image('CSP/ERGo/1_0/Global/ALOS_landforms').select('constant').resample('bi
```

```
# Merge layers as bands
dataset = weather.addBands(soil).addBands(landforms)
```

13.1.3 Generate clusters

```
# Select random points across the image in the defined region for training
sample_points = dataset.sample(region=geom, scale=20, numPixels=1000)
```

```
# Define number of cluster to classify
cluster_number = 5
classificator = ee.Clusterer.wekaKMeans(cluster_number).train(sample_points)
```

13.1.4 Reduce image to vector

```
# Generate the clusters for the whole region
clusters = dataset.cluster(classificator).reduceToVectors(scale=20, geometry=geom).getInfo
```

13.1.5 Interactive plot

```
# Create Folium Map
m = folium.Map(location=[39.08648, -96.582], zoom_start=13)

cmap = get_cmap('Set1',n=cluster_number)
cluster_layer = folium.GeoJson(clusters, name="Kings Creek",
                               style_function=lambda x:get_polygon_color(x,cmap))
cluster_layer.add_to(m)

m
```

```
<folium.folium.Map at 0x7fb710d4fa00>
```

13.1.6 Export clusters as geoJSON

```
# Inspect resulting clusters
# pprint(clusters)
```

```
# Save result to a geoJSON file
with open('outputs/output.json', 'w') as file:
    json.dump(clusters, file)
```

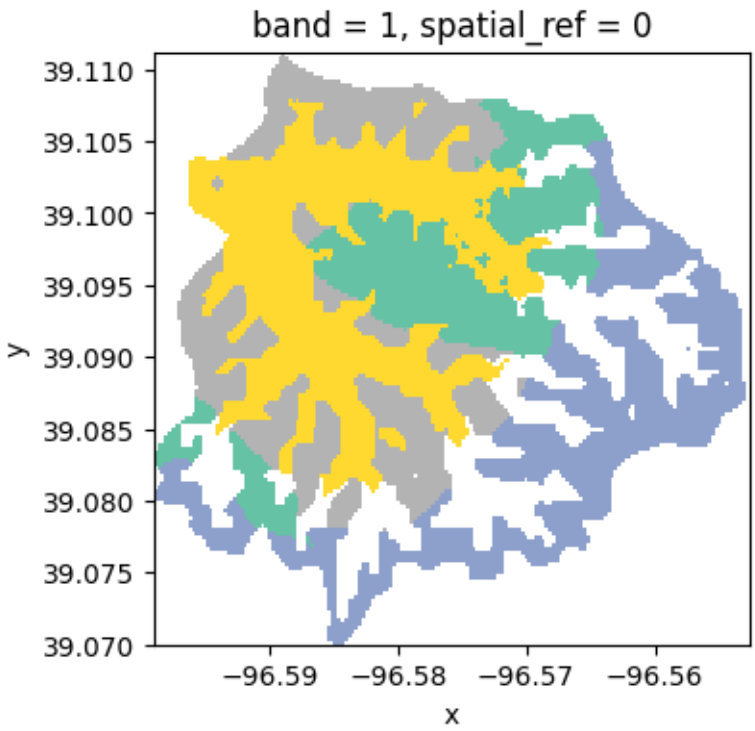
13.1.7 Export clusters as GeoTIFF

```
# Create mask
mask = ee.Image.constant(1).clip(geom).mask()
```

```
# Create image URL
image_url = dataset.cluster(classificator).mask(mask).getDownloadUrl({
    'region': geom,
    'scale':30,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF'
})
```

```
# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)
filename = '../outputs/kings_creek_clusters.tif'
with open(filename, 'wb') as f:
    f.write(response.content)
```

```
# Read GeoTiff file using the Xarray package
raster = xr.open_dataarray(filename).squeeze()

#fig, ax = plt.subplots(1,1,figsize=(6,5))
raster.plot(cmap='Set2', figsize=(4,4), add_colorbar=False);
```



13.2 Example 2: State-level clustering

In this tutorial we will identify macroregions across Kansas determined by climate and soil variables.

13.2.1 Load region

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
kansas = US_states.filter(ee.Filter.eq('NAME','Kansas'))
```

13.2.2 Prepare clustering dataset

```
# Import image layers
ppt = ee.ImageCollection('OREGONSTATE/PRISM/Norm91m').select('ppt').sum().resample('bicubi
tmean = ee.ImageCollection('OREGONSTATE/PRISM/Norm91m').select('tmean').mean().resample('b
vpdmax = ee.ImageCollection('OREGONSTATE/PRISM/Norm91m').select('vpdmax').mean().resample(
soil = ee.Image("projects/soilgrids-isric/sand_mean").select('sand_0-5cm_mean').resample('

# Merge layers into a single dataset
dataset = ppt.addBands(soil).addBands(tmean).addBands(vpdmax)
```

13.2.3 Generate clusters

```
# Select random points across the image in the defined region for training
sample_points = dataset.sample(region=kansas, scale=1000, numPixels=1000)
```

```
# Define number of cluster to classify
cluster_number = 9
classificator = ee.Clusterer.wekaKMeans(cluster_number).train(sample_points)
```

13.2.4 Reduce image to vectors

```
# Generate the clusters for the whole region
clusters = dataset.cluster(classificator).reduceToVectors(scale=1000, geometry=kansas).get
```

13.2.5 Interactive plot

```
# Create Folium Map
m = folium.Map(location=[38.5, -98.5], zoom_start=7)

cmap = get_cmap('Set1',n=cluster_number)
cluster_layer = folium.GeoJson(clusters, name="Kansas",
                               style_function=lambda x:get_polygon_color(x,cmap))
cluster_layer.add_to(m)

m
```

```
<folium.folium.Map at 0x7fb7102e5cc0>
```

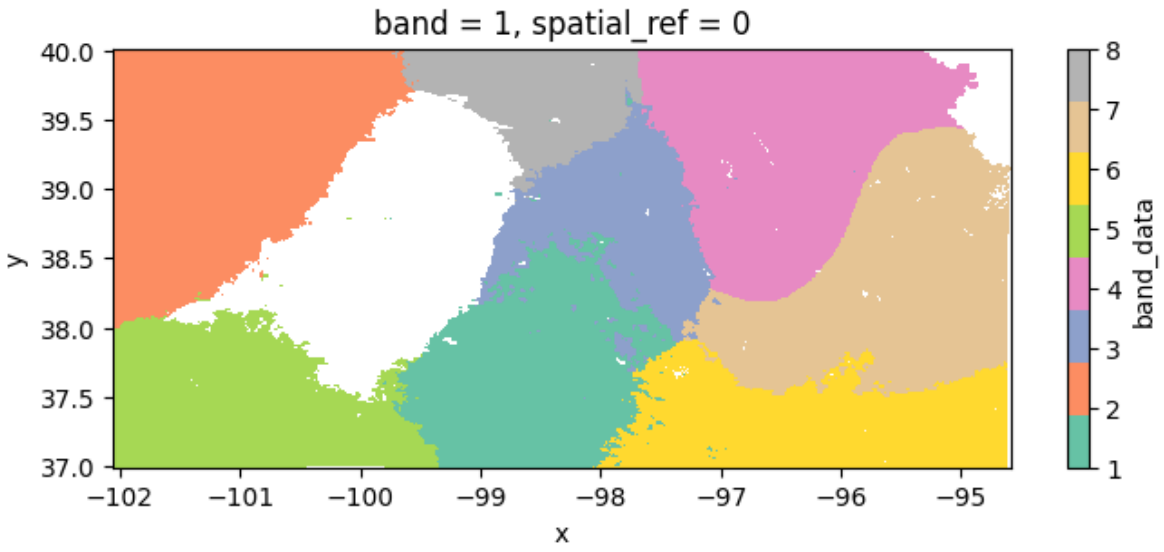
13.2.6 Export clusters as GeoTIFF

```
# Create mask
mask = ee.Image.constant(1).clip(kansas).mask()
```

```
# Create image URL
image_url = dataset.cluster(classificator).mask(mask).getDownloadUrl({
    'region': kansas.geometry(),
    'scale':1000,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF'
})
```

```
# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)
filename = '../outputs/kansas_clusters.tif'
with open(filename, 'wb') as f:
    f.write(response.content)
```

```
# Read GeoTiff file using the Xarray package
raster = xr.open_dataarray(filename).squeeze()

#fig, ax = plt.subplots(1,1,figsize=(6,5))
raster.plot(cmap='Set2', figsize=(8,3));
```


band = 1, spatial_ref = 0
40.0
39.5
39.0
38.5
38.0
37.5
37.0
y
−102
−101
−100
−99
−98
−97
−96
−95
x
8
7
6
5
4
3
2
1
band_data

14 Supervised classification

Supervised learning techniques, like the Random Forest algorithm, offer a robust framework for classifying complex landscapes, such as distinguishing between riparian and grassland areas within a watershed. This method relies on manually selected and labeled training areas to teach the model how to recognize these distinct environments. This tutorial focuses on using satellite imagery and Random Forest classification in Google Earth Engine (GEE) to accurately identify and classify riparian versus grassland regions in a watershed.

```
# Import modules and initialize
import ee
import json
import folium
import numpy as np
from pprint import pprint
from matplotlib import colormaps, colors
import requests
import xarray as xr
import matplotlib.pyplot as plt
import geopandas as gpd
```

```
# Authenticate
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

14.1 Define helper functions

```
# Create helper functions

# Get Color for different polygons
def get_polygon_color(polygon):
    color_values = ['#7b3294', '#008837']
```

```
    color = color_values[polygon['properties']['label']]
    return {'fillColor':color, 'fillOpacity':0.8}


def maskS2clouds(image):
    """
    Function to filter images by cloud percentage
    """
    qa = image.select('QA60')
    cloudBitMask = 1 << 10 # bits 10
    cirrusBitMask = 1 << 11 # bits 11

    # Both flags should be set to zero (clear sky conditions)
    mask = qa.bitwiseAnd(cloudBitMask).eq(0).And(qa.bitwiseAnd(cirrusBitMask).eq(0))
    return image.updateMask(mask)


# Define auxiliary functions
def create_raster(ee_object, vis_params, name):
    """
    Function that createst GEE map into raster for folium map
    """
    raster = folium.raster_layers.TileLayer(ee_object.getMapId(vis_params)['tile_fetcher']
                      name=name,
                      overlay=True,
                      control=True,
                      attr='Map Data &copy; <a href="https://earthengine.google.com/">Goo
    return raster
```

14.2 Load region boundaries

```
# Import boundary for region of interest (roi)
with open('../datasets/kings_creek.geojson') as file:
    roi_json = json.load(file)

# Define the ee.Geometry
roi_geom = ee.Geometry(roi_json['features'][0]['geometry'])
```

14.3 Load labeled regions for training

These polygons were created manually by inspecting a satellite image and drawing over small portions of the watershed that were clearly with riparian and grassland vegetation. Here is an excellent tool to get you started: https://geojson.io

```
# Import labeled regions for riparian and grassland areas.
with open('../datasets/riparian.geojson') as file:
    riparian_json = json.load(file)

with open('../datasets/grassland.geojson') as file:
    grassland_json = json.load(file)


# Define the ee.Geometry for each polygon
riparian_geom = ee.Feature(riparian_json['features'][0]['geometry'])
grassland_geom = ee.Feature(grassland_json['features'][0]['geometry'])
```

```
# Add class values to features
riparian_geom = riparian_geom.set('land_cover', 0)
grassland_geom = grassland_geom.set('land_cover', 1)
```

```
# Merge train dataset into a FeatureCollection
training_regions = ee.FeatureCollection([riparian_geom, grassland_geom])
```

14.4 Load image dataset

This image will be used to detect riparian and grassland vegetation beyond the provided training regions.

```
# Sentinel 2 multispectral instrument
S2 = ee.ImageCollection('COPERNICUS/S2_SR').filterDate('2022-06-01', '2022-07-31')
S2 = S2.filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE',10)).map(maskS2clouds)

# Mean multispectral image for entire period
img = S2.mean().divide(10000).clip(roi_geom)
```

```
# Select values from dataset for the training regions
training_data = img.sampleRegions(collection=training_regions,
                                  properties=['land_cover'], scale=30)
```

```
# Train Random Forest algorithm
ntrees = 10
trained_classifier = ee.Classifier.smileRandomForest(ntrees).train(features=training_data,
                                                                   classProperty='land_cover',
                                                                   inputProperties=img.bandNames())
```

```
# Classify the entire King's Creek Watershed
classified_img = img.classify(trained_classifier)
classified_vector = classified_img.reduceToVectors(scale=30,
                                                   geometry=roi_geom).getInfo()
```

14.5 Save static maps

Download vector and raster maps.

```
# Save vector map to GeoJSON file
filename_classified_vector = '../outputs/classified_vector_kings_creek.tif'
with open(filename_classified_vector, 'w') as file:
    # Convert dictionary to a GeoJSON string and save it
    file.write(json.dumps(classified_vector))


# Create raster of classified image in geoTIFF image
classified_img_url = classified_img.getDownloadUrl({
    'region': roi_geom,
    'scale':30,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF'
})

# Request and save geoTIFF map
response = requests.get(classified_img_url)
filename_classified_img = '../outputs/classified_kings_creek.tif'
with open(filename_classified_img, 'wb') as f:
    f.write(response.content)
```

```
# Create raster of classified image in geoTIFF image
truecolor_img_url = img.getDownloadUrl({
    'region': roi_geom,
    'scale':30,
    'crs': 'EPSG:4326',
    'format': 'GEO_TIFF',
    'bands':['B4', 'B3', 'B2']
})

# Request and save geoTIFF map
response = requests.get(truecolor_img_url)
filename_truecolor_img = '../outputs/truecolor_kings_creek.tif'
with open(filename_truecolor_img, 'wb') as f:
    f.write(response.content)
```

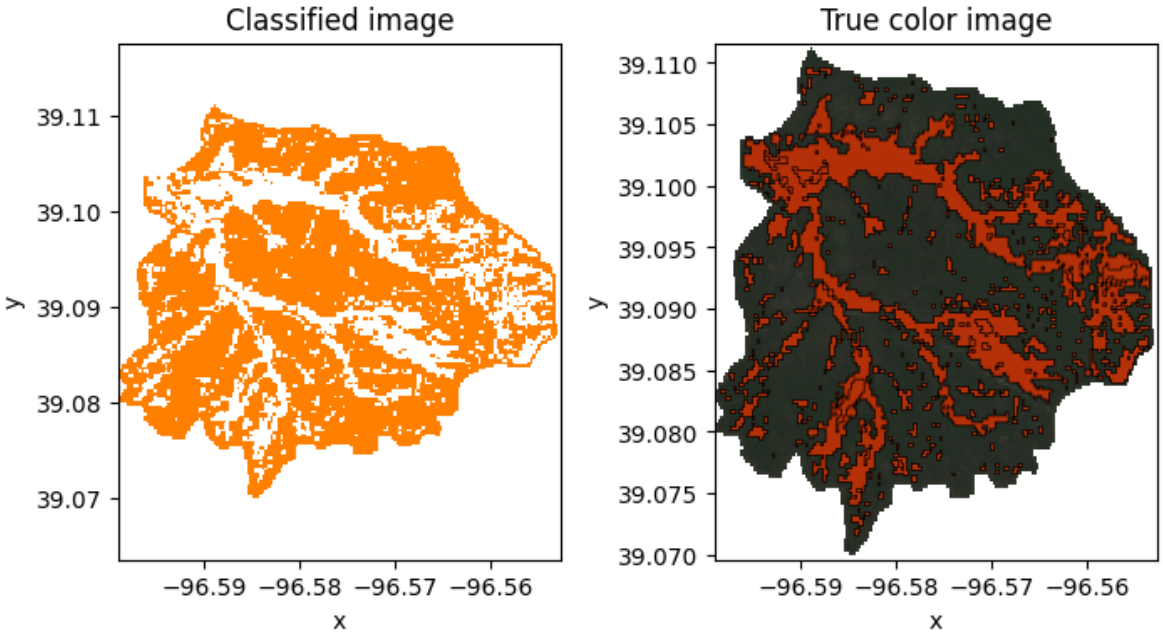
14.6 Create static figure

```
# Read GeoTiff file using the Xarray package
raster_classified = xr.open_dataarray(filename_classified_img).squeeze() # 2d image
raster_truecolor = xr.open_dataarray(filename_truecolor_img)

# COnvert vector layer to GeoDataframe
gdf = gpd.GeoDataFrame.from_features(classified_vector['features'], crs=4326)
gdf.head(3)
```

	geometry	count	label
0	POLYGON ((-96.55425 39.09019, -96.55344 39.090...	3	0
1	POLYGON ((-96.55398 39.08776, -96.55398 39.087...	77	0
2	POLYGON ((-96.55452 39.08965, -96.55452 39.089...	5	0

```
fig,(ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8,4))
fig.subplots_adjust(wspace=0.35)

raster_classified.plot.imshow(ax=ax1, cmap='Set1', add_colorbar=False);
ax1.set_title('Classified image')
ax1.axis('equal')
```

```
raster_truecolor.plot.imshow(ax=ax2);
ax2.set_title('True color image')
ax2.axis('equal')

idx_riparian = gdf['label'] == 0 # select rows for riparian areas
gdf.loc[idx_riparian].plot(ax=ax2, facecolor=(0.9,0.2,0,0.75),
                           edgecolor='k', linewidth=0.5)


plt.show()
```



14.7 Interactive map

```
# Create Folium Map
m = folium.Map(location=[39.09, -96.592], zoom_start=13)

# Create vector layer of classified polygons
cluster_layer = folium.GeoJson(classified_vector,
                               name="Classified image with RF",
                               style_function=get_polygon_color)
```

```
# Visualization parameters of true color image
# B4=Red, B3=Green, B2=Blue
vis_params = {'min': 0.0, 'max': 0.3, 'bands': ['B4', 'B3', 'B2'], }


# Create raster layer of true color image
true_color_layer = folium.raster_layers.TileLayer(img.getMapId(vis_params)['tile_fetcher']
                        name='True color',
                        overlay=True,
                        control=True,
                        attr='Map Data &copy; <a href="https://earthengine.google.com/">Goo

# Add layers to interactive map
true_color_layer.add_to(m)
cluster_layer.add_to(m)

# Add map controls to be able to compare
# the true color image with the clasified layer
folium.LayerControl().add_to(m)

# Render map
m
```

```
<folium.folium.Map at 0x7fb52e963bb0>
```

15 Export data

In many cases, it's necessary to store data from Google Earth Engine (GEE) locally for offline analysis and integration with other projects and tools. This notebook demonstrates techniques for saving time series data and images to a local hard drive, with similar examples found throughout the book.

A critical factor in downloading data from GEE is its limitation on data size per download request. Downloading and processing large datasets locally negates the advantage of GEE, which is designed to accelerate data processing through its advanced infrastructure.

GEE limits image downloads to a maximum size of 32 MB or a maximum grid dimension of 10,000 pixels.

```
# Import modules
import ee
import requests
import io
import json
import pandas as pd
import xarray as xr
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mpd
from matplotlib.colors import rgb2hex
```

```
# Authenticate GEE
#ee.Authenticate()

# Initialize GEE
ee.Initialize()
```

15.1 Select region for tutorial

```
# Read US watersheds using hydrologic unit codes (HUC)
watersheds = ee.FeatureCollection("USGS/WBD/2017/HUC12")
mcdowell_creek = watersheds.filter(ee.Filter.eq('huc12', '102701010204')).first()
watershed_point = ee.Geometry.Point([-96.556316, 39.084535])

# Create mask for the region
mask = ee.Image.constant(1).clip(mcdowell_creek).mask()
```

15.2 Raster data

15.2.1 Save geotiff image

In this example we will request a small georeferenced image and then save it in .geotiff format. We will also read the image back into the notebook.

```
# Import map from Digital Elevation Model (DEM)
srtm = ee.Image('CGIAR/SRTM90_V4')
```

```
# Clip elevation map to boundaries of McDowell Creek
# and mask points outside the watershed boundary
mcdowell_creek_elv = srtm.clip(mcdowell_creek).mask(mask)
```

```
# Define coordinate reference system
crs = 32614 # UTM14 Projected coordinates in meters
#crs = 4326 # WGS84 Geographic coordinates in degrees


# Get URL link to full image
image_url = mcdowell_creek_elv.getDownloadUrl({
    'region': mcdowell_creek.geometry(),
    'scale':30,
    'crs': f'EPSG:{crs}',
    'format': 'GEO_TIFF'
})

# Display clickable URL link to download TIFF image
```

```
print(image_url)
```

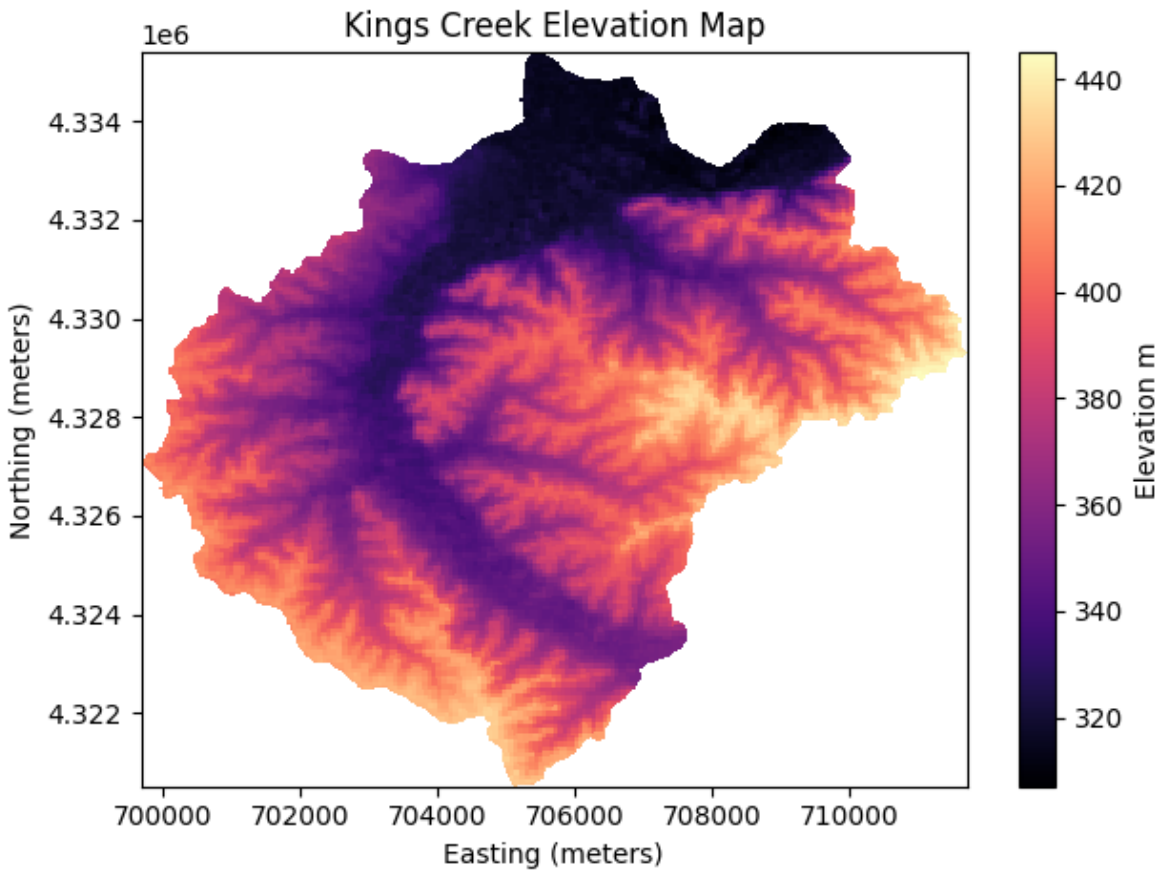
```
https://earthengine.googleapis.com/v1alpha/projects/earthengine-legacy/thumbnails/a3c4c75d3e
```

> **Tip**
>
> If we request the image using `"scale: 1"`, meaning that we want an image with each pixel representing 1 square meter, then for this particular watershed ww will get the following error message: `"Total request size (691287720 bytes) must be less than or equal to 50331648 bytes."`, which means that we are requesting too much data. To solve this issue we need to reduce the number of pixels by re-scaling the image so that we to meet the quota allowed by GEE.

```
# Request data using URL and save data as a new GeoTiff file
response = requests.get(image_url)

# Write file to disk
filename = '../outputs/mc_dowell_creek_elevation.tif'
with open(filename, 'wb') as f:
    f.write(response.content)
```

```
# Read GeoTiff file using the Xarray package
raster = xr.open_dataarray(filename).squeeze()
```

```
# Create a figure
raster.plot(cmap='magma', cbar_kwargs={'label':'Elevation m'});
plt.title('Kings Creek Elevation Map')
plt.xlabel('Easting (meters)')
plt.ylabel('Northing (meters)')
plt.tight_layout()
plt.show()
```



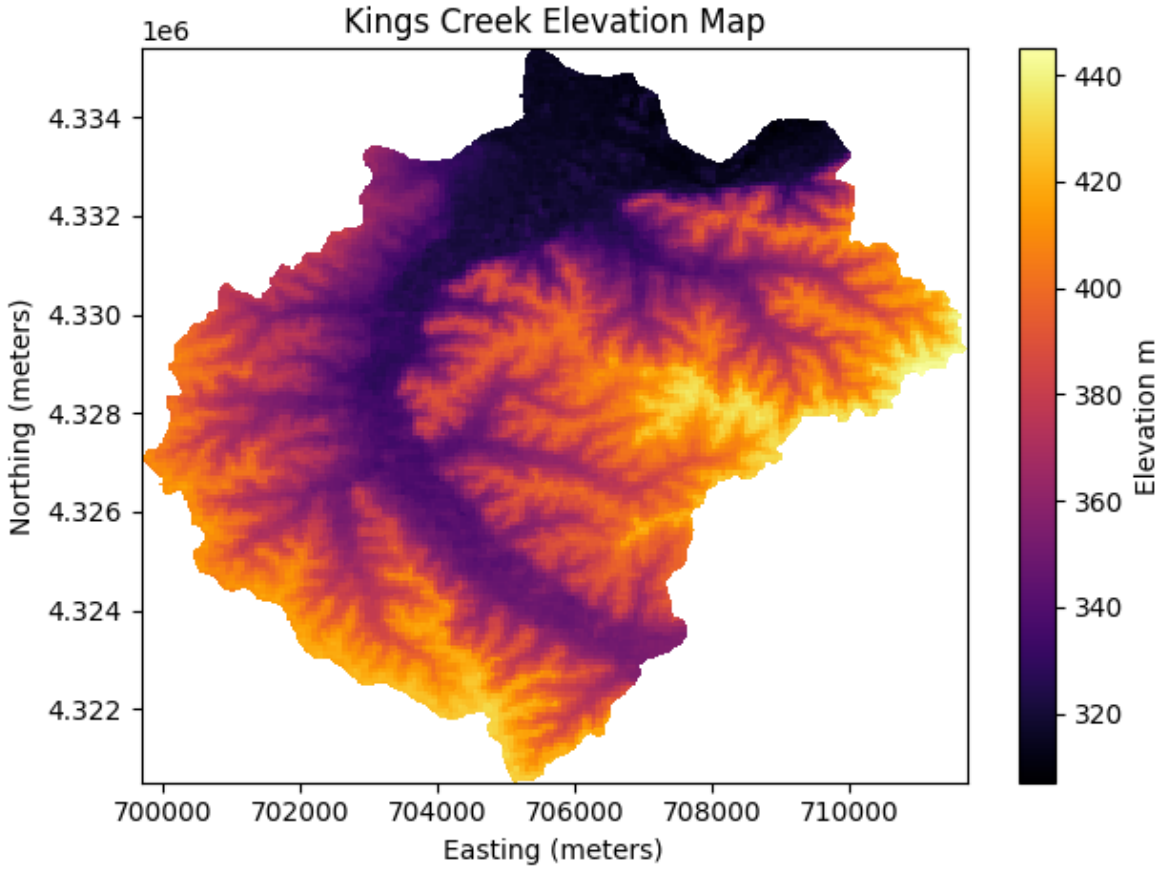
15.2.2 In-memory GeoTiff

Sometimes we simply want to retrieve an image and plot it, without necessarily saving it into our local drive.

If the code below does not work, try installating the `rioxarray` package using `!pip install rioxarray`, which extends `xarray` and allows it to use the `rasterio engine` to read bytes file.

```
# Open temporary file in memory, save the content
# of the request (this is the geotiff image) in a temporary file,
# then read the temporary file and plot the map.
with io.BytesIO(response.content) as memory_file:
    raster = xr.open_dataarray(memory_file, engine='rasterio').squeeze()

    # Create plot while the file is open
```

```
    raster.plot(cmap='inferno', cbar_kwargs={'label':'Elevation m'});
    plt.title('Kings Creek Elevation Map')
    plt.xlabel('Easting (meters)')
    plt.ylabel('Northing (meters)')
    plt.tight_layout()
    plt.show()
```



15.2.3 Handy function to save GeoTiffs

Since this is a common operation, let's wrap the code we wrote above into a function.

```
def save_geotiff(ee_image, filename, crs, scale, geom):
    """
```

```
    Function to save images from Google Earth Engine into local hard drive.
    """
    image_url = ee_image.getDownloadUrl({
    'region': geom,
    'scale':scale,
    'crs': f'EPSG:{crs}',
    'format': 'GEO_TIFF'})

    # Request data using URL and save data as a new GeoTiff file
    response = requests.get(image_url)
    with open(filename, 'wb') as f:
        f.write(response.content)
        return print(f'Image {filename} saved.')
```

15.2.4 Get Numpy array

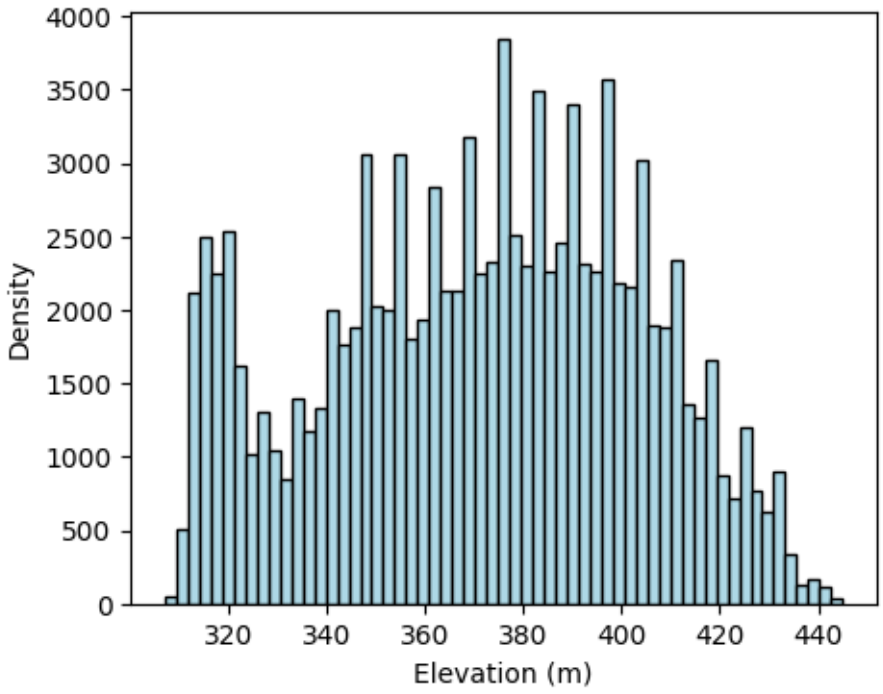
Sometimes instead of the georeferenced image we want the raw data in the form of a Numpy array, so that we can further process the data in Python.

```
# Use the requests method to read image
numpy_array_url = mcdowell_creek_elv.getDownloadUrl({'crs': f'EPSG:{crs}',
                                                     'scale':30,
                                                     'format': 'NPY'})
response = requests.get(numpy_array_url )
elev_array = np.load(io.BytesIO(response.content), encoding='bytes').astype(np.float64)
print(elev_array)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

```
# Convert zero values in the mask to NaN
idx_zero = elev_array == 0
elev_array[idx_zero] = np.nan
```

```
# Create histogram of elevation
plt.figure(figsize=(5,4))
plt.hist(elev_array.flatten(),
         bins='scott',
         facecolor='lightblue', edgecolor='k');
plt.xlabel('Elevation (m)')
plt.ylabel('Density')
plt.show()
```




```
print('Array dimensions:', elev_array.shape)
print('Total pixels:', elev_array.size)
```

```
Array dimensions: (496, 401)
Total pixels: 198896
```

15.2.5 Save thumbnail image

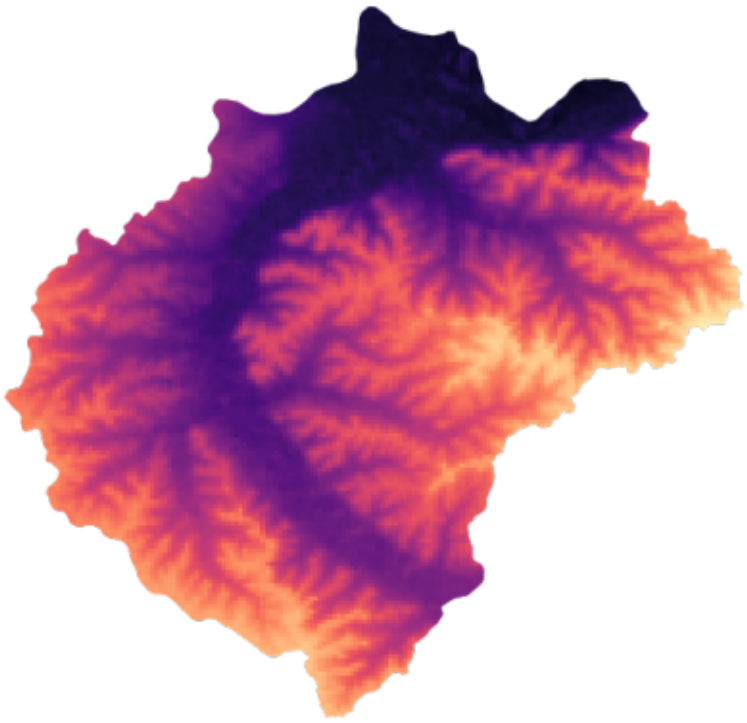
A thumbnail image is good for display purposes. This is not the actual, high-resolution image. This image does not have any elevation data or geographic coordinates. The color of each

pixel was assigned according to the colormap and range of values provided when requesting the thumbnail image.

```
# Create the url associated to the Image you want
cmap = [rgb2hex(c) for c in plt.get_cmap('magma').colors]
thumbnail_url = mcdowell_creek_elv.getThumbUrl({'min': 300,
                                        'max': 450,
                                        'dimensions': 512,
                                        'palette': cmap})

# Print URL
print(thumbnail_url)
```

```
https://earthengine.googleapis.com/v1alpha/projects/earthengine-legacy/thumbnails/f766af93b23
```

```
# Display a thumbnail of elevation map
response = requests.get(thumbnail_url )
img = plt.imread(io.BytesIO(response.content))
plt.imshow(img, cmap='magma')
plt.axis('off')
plt.savefig('../outputs/kings_creek_thumbnail.jpg')
plt.show()
```



15.3 Vector data

15.3.1 Get Feature coordinates into Pandas DataFrame

In this example we will simply access the latitude and longitude fields of a Kansas county.

```
# US Counties dataset
US_counties = ee.FeatureCollection("TIGER/2018/Counties")

# '20161' is the ID for Riley county in Kansas
county = US_counties.filter(ee.Filter.eq('GEOID','20161'))

# Get coordinates of the county geometryand put them in a dataframe for easy data handling
df = pd.DataFrame(county.first().getInfo()['geometry']['coordinates'][0])
df.columns = ['lon','lat']
df.head()
```

	lon	lat
0	-96.961683	39.220095
1	-96.961369	39.220095
2	-96.956566	39.220005
3	-96.954188	39.220005
4	-96.952482	39.220005

15.3.2 Save data in tabular format

In this example the goal is to request temporal soil moisture data, convert it into a Pandas DataFrame, and then save it as a comma-separated value file.

```
# Load SMAP product (3-hour resolution)
smap = ee.ImageCollection("NASA/SMAP/SPL4SMGP/007").filterDate('2018-01-01','2018-01-31')

# Select rootzone soil moisture band
sm_rootzone = smap.select('sm_rootzone')

# Get rootzone soil moisture for watershed point
watershed_sm = sm_rootzone.getRegion(watershed_point, scale=1).getInfo()
```

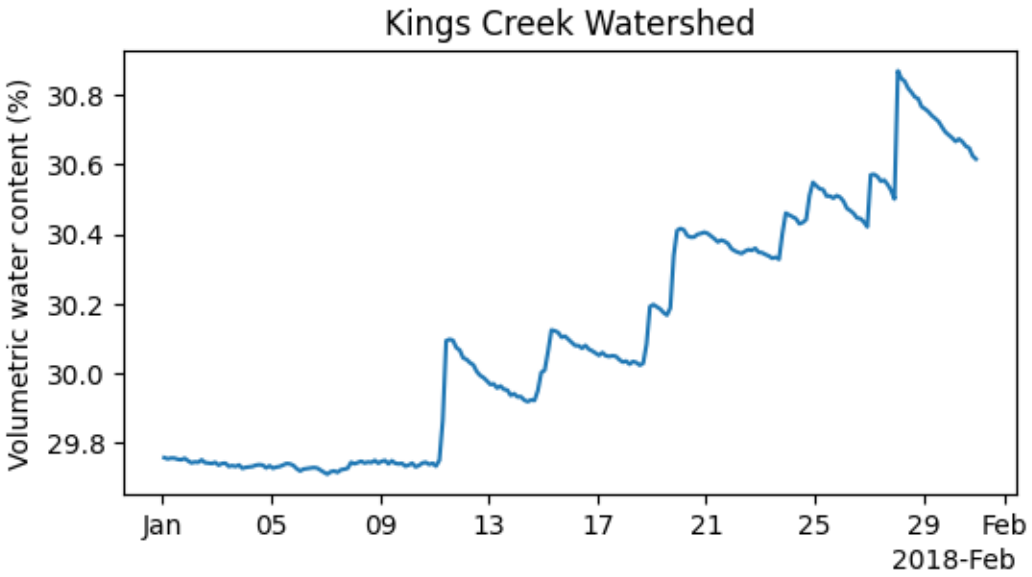
```
# Convert output into a Pandas Dataframe
df = pd.DataFrame(watershed_sm[1:])
df.columns = watershed_sm[0]
df['time'] = pd.to_datetime(df['time'], unit='ms')
df.head(3)
```

	id	longitude	latitude	time	sm_rootzone
0	20180101_0130	-96.556312	39.084535	2018-01-01 01:30:00	0.297576
1	20180101_0430	-96.556312	39.084535	2018-01-01 04:30:00	0.297541
2	20180101_0730	-96.556312	39.084535	2018-01-01 07:30:00	0.297562

```
# Save Dataframe to local drive
df.to_csv('../outputs/smap_rootzone_2018.csv', index=False)
```

```
# Create figure to visualize data
date_fmt = mpd.ConciseDateFormatter(mpd.AutoDateLocator())
plt.figure(figsize=(6,3))
```

```
plt.title('Kings Creek Watershed')
plt.plot(df['time'], df['sm_rootzone']*100)
plt.ylabel('Volumetric water content (%)')
plt.gca().xaxis.set_major_formatter(date_fmt)
plt.show()
```



15.3.3 Save Feature as geoJSON

We will save a polygon feature representing the boundary of the McDowell Creek small watershed in central Kansas as a geoJSON file.

```
# Convert the feature to a GeoJSON string
feature_info = mcdowell_creek.getInfo()  # Retrieves the information about the feature

# Save the GeoJSON string to a file
with open('../outputs/kings_creek_from_gee.geojson', 'w') as file:
    # Convert dictionary to a GeoJSON string and save it
    file.write(json.dumps(feature_info))
```

In this example we will save the boundary of the state of Kansas.

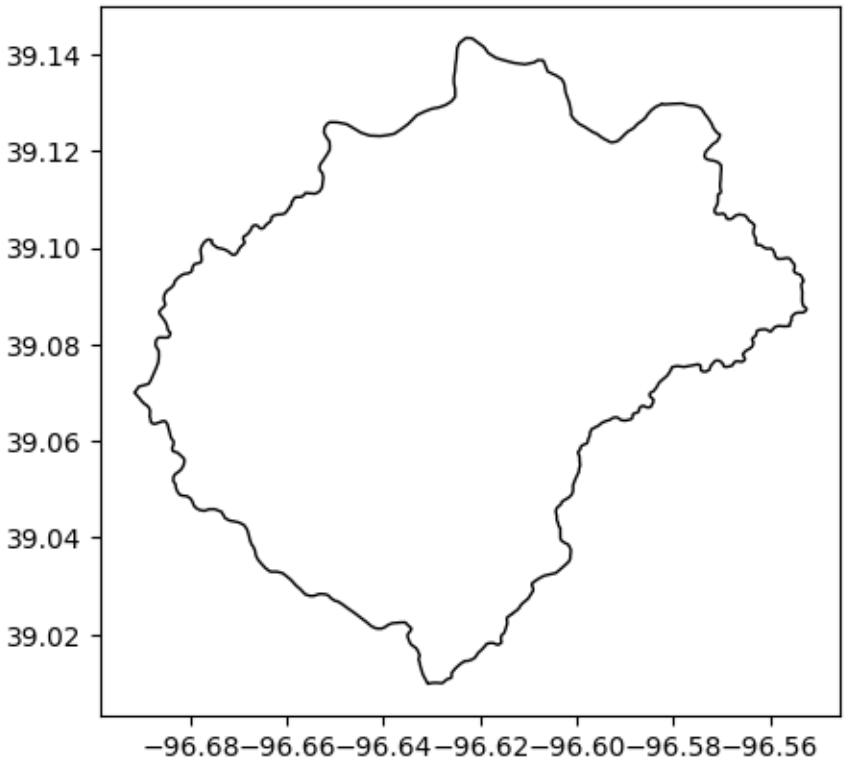
```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
state = US_states.filter(ee.Filter.inList('NAME',['Kansas']))

# Retrieves the information about the feature
state_info = state.getInfo()

# Save the GeoJSON string to a file
with open('../outputs/kansas_bnd_from_gee.geojson', 'w') as file:

    # Convert dictionary to a GeoJSON string and save it
    file.write(json.dumps(feature_info))
```

15.3.4 Save FeatureCollection as shapefile

This section requires the GeoPandas library.

```
# Import GeoPandas
import geopandas as gpd
```

```
# Convert FeatureCollection to GeoDataFrame
gdf = gpd.GeoDataFrame.from_features([feature_info])

# Create a figure to quickly inspect the dataset
gdf.plot(facecolor='None');
```




```
# Save as shapefile
gdf.to_file('../outputs/kings_creek_from_gee.shp', index=False)
```

```
# Save as geojson using GeoPandas
gdf.to_file('../outputs/kings_creek_from_gee.geojson', driver='GeoJSON', index=False)
```

15.3.5 Save FeatureCollection as shapefile (Advanced)

In this case we will save a FeatureCollection with Polygon features and with GeometryCollection features, which are not supported by shapefiles. So we will need to iterate over each geometry, and only select polygons for those geometries that contain other features, like Points or LineStrings. Using geojson is recommended in this case to avoid the extra processing to fit the specifications of ESRI Shapefiles.

Ideally all the counties should be Polygons, but sometimes geometries bordering rivers or other states can result in GeometryCollections.

```
# Read US counties and filter bounds for selected region
counties = ee.FeatureCollection("TIGER/2018/Counties")

# Get FeatureCollection data
# Kansas has STATEFP number equal to 20
counties_data = counties.filter(ee.Filter.eq('STATEFP','20')).getInfo()
```

```
# Define coordinate reference system
wgs84 = 4326 # World Geodetic System 1984

# Convert FeatureCollection to GeoDataFrame
gdf_counties = gpd.GeoDataFrame.from_features(counties_data, crs=wgs84)
gdf_counties.head(3)
```

	geometry	ALAND	AWATER	CBSAFP	CLASSFP	C
0	POLYGON ((-97.15333 37.47553, -97.15329 37.474...	2915648163	17322935	11680	H1	0
1	POLYGON ((-97.15347 37.82517, -97.15342 37.824...	3702816810	43671391	48620	H1	0
2	POLYGON ((-97.80760 37.47387, -97.80755 37.472...	3060429574	8589030	48620	H1	1

```
# Find if we have features that are not polygons
gdf_counties[gdf_counties.geometry.type != 'Polygon']
```

	geometry	ALAND	AWATER	CBSAFP	CL
46	GEOMETRYCOLLECTION (LINESTRING (-96.52551 36.9...	1654694134	15243776		H1
64	GEOMETRYCOLLECTION (LINESTRING (-98.34961 37.3...	2075290355	3899269		H1
99	GEOMETRYCOLLECTION (LINESTRING (-94.87632 39.7...	1019105692	12424173	41140	H1

```
# Let's inspect one of them to see what's inside
# Here we have a LineString and a Polygon.
list(gdf_counties.loc[46,'geometry'].geoms)

# The shapefile will not be able to save this. Let's retain the polygon only
```

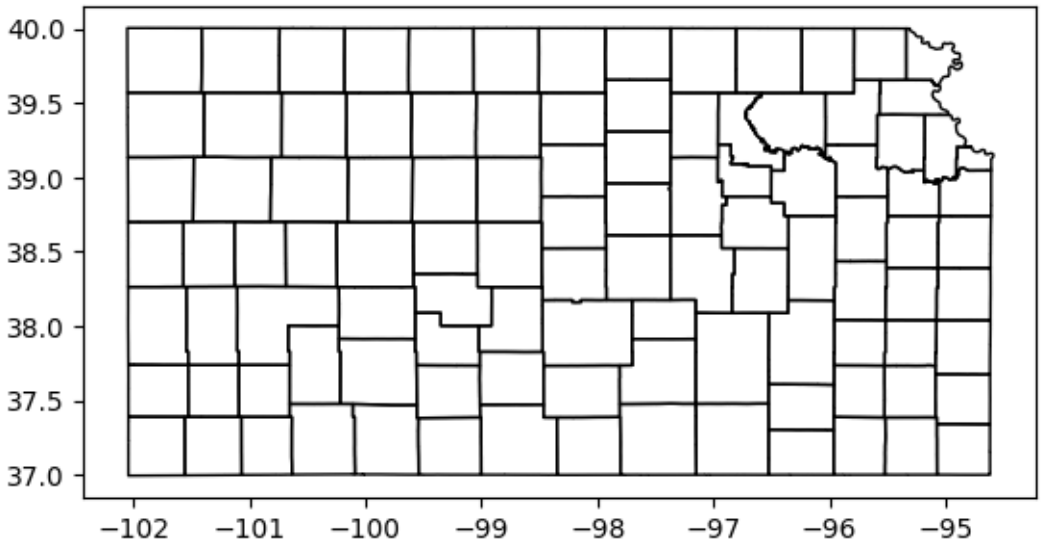
```
[<LINESTRING (-96.526 36.999, -96.526 36.999)>,
 <POLYGON ((-96.526 37.028, -96.526 37.016, -96.526 37.016, -96.526 37.008, -...>]
```

```
# Iterate over each row. Here we assume that there is
# only one polygon in each row that has a GeometryCollection.
for k,row in gdf_counties.iterrows():

    # Check if row geometry is 'GeometryCollection'
    if row['geometry'].geom_type == 'GeometryCollection':

        # Iterate over each component of the GeometryCollection
        for geom in list(row['geometry'].geoms):

            # Overwrite geometry of row with polygon
            if geom.geom_type == 'Polygon':
                gdf_counties.loc[k,'geometry'] = geom

# For multiple polygons you can keep the largest one using
# something like: max(polygons, key=lambda a: a.area)
```

```
# Check that we don't have any rows left that is not a polygon
# Returned GeoDataFrame should be empty
gdf_counties[gdf_counties.geometry.type != 'Polygon']
```

geometry	ALAND	AWATER	CBSAFP	CLASSFP	COUNTYFP	COUNTYNS	CSAFP	FUNC

```
# Plot counties to ensure we have all of them
gdf_counties.plot(facecolor='None');
```




```
# Now we can save the GeoDataframe as a Shapefile
gdf_counties.to_file('../outputs/kansas_counties_from_gee.shp', index=False)
```

16 Interactive maps

Interactive maps provide a dynamic user experience, allowing for real-time zooming, panning, and exploration of geospatial data layers. In contrast, Matplotlib's static figures, while useful for publication-quality visuals, lack the ability to explore data or to easily integrate multiple layers of information. Additionally, interactive maps can be easily integrated withing websites and are ideal for collaborative projects and public dissemination of geospatial research findings.

In this tutorial we will create an interactive map to explore different land covers across the state of Kansas. We will use the `folium` Python library, which is a wrapper around the popular Leaflet library implemented Javascript.

In this tutorial we will learn:

- How to add raster layers to an interactive map
- How to add vector layers to an interactive map

```
# Import modules
import ee
import folium
import numpy as np
from matplotlib import colors, colormaps
```

16.1 Define some helper functions

Let's define some functions to avoid repeating the following code everytime we want to create a raster image or every time we want to specify a given colormap for our map.

```
def get_raster_map(ee_object, vis, name):
    """
    Function that retrieves raster map from GEE and applies some style.
    """
    if isinstance(ee_object, ee.Image):
        layer = folium.TileLayer(ee_object.getMapId(vis)['tile_fetcher'].url_format,
                                 name=name,
```

```
                            overlay=True,
                            control=True,
                            attr='Map Data &copy; <a href="https://earthengine.google.com/"

    return layer



def get_vector_map(ee_object,vis,name):
    """
    Function that retrieves vector map from GEE and applies some style.
    """
    if isinstance(ee_object, ee.FeatureCollection):
        layer = folium.GeoJson(ee_object.getInfo(),
                               name=name,
                               style_function=lambda feature: {
                                   'fillColor': vis['face_color'],
                                   'color': vis['edge_color'],
                                   'weight': vis['line_width']})
    return layer
```

```
# Authenticate GEE (follow the link and copy-paste the token in the notebook)
#ee.Authenticate()

# Initialize the library.
ee.Initialize()
```

16.2 State land cover

```
# Read US states
US_states = ee.FeatureCollection("TIGER/2018/States")

# Select Kansas
region = US_states.filter(ee.Filter.inList('NAME',['Kansas']))
```

```
# Land use for 2021
land_use = ee.ImageCollection('USDA/NASS/CDL')\
             .filter(ee.Filter.date('2020-01-01', '2021-12-31')).first().clip(region)
```

```
# Select cropland layer
cropland = land_use.select('cropland')
```

```
# Get layer metadata
info = cropland.getInfo()

# Remove comment to print the entire information (output is long!)
# print(info)
print(info.keys())
```

```
dict_keys(['type', 'bands', 'id', 'version', 'properties'])
```

```
# Get land cover names, values, and colors from metadata

class_names = info['properties']['cropland_class_names']
class_values = info['properties']['cropland_class_values']
class_colors = info['properties']['cropland_class_palette']
```

```
# Create dictionary to easily access properties by land cover name
class_props = {}
for k,name in enumerate(class_names):
    class_props[name] = {'value':class_values[k], 'color':class_colors[k]}

# Print example
class_props['Corn']
```

```
{'value': 1, 'color': 'ffd300'}
```

```
# Create self masks for each land cover, so that only
# the pixels for this land cover visible

corn = cropland.eq(class_props['Corn']['value']).selfMask()
sorghum = cropland.eq(class_props['Sorghum']['value']).selfMask()
soybeans = cropland.eq(class_props['Soybeans']['value']).selfMask()
wheat = cropland.eq(class_props['Winter Wheat']['value']).selfMask()
grassland = cropland.eq(class_props['Grassland/Pasture']['value']).selfMask()
```

> **Tip**
>
> Since we created a dictionary using the name of the land cover class as the key, in the previous cell you can use Tab-autocompletion.

```
# Initialize map centered at a specific location and zoom level
m = folium.Map(location=[38, -98], zoom_start=7)

get_raster_map(corn, {'palette':[class_props['Corn']['color']]}, 'Corn').add_to(m)
get_raster_map(sorghum, {'palette':[class_props['Sorghum']['color']]}, 'Sorghum').add_to(m
get_raster_map(soybeans, {'palette':[class_props['Soybeans']['color']]}, 'Soybeans').add_t
get_raster_map(wheat, {'palette':[class_props['Winter Wheat']['color']]}, 'Wheat').add_to(
get_raster_map(grassland, {'palette':[class_props['Grassland/Pasture']['color']]}, 'Grassl

# Add vector layers
get_vector_map(region, vis={'face_color':'#00FFFF00',
                     'edge_color':'black',
                     'line_width':2}, name='State boundary').add_to(m)

get_vector_map(counties, vis={'face_color':'#00FFFF00',
                     'edge_color':'grey',
                     'line_width':1}, name='County boundary').add_to(m)


# Add a layer control panel to the map.
m.add_child(folium.LayerControl())

# Display the map.
display(m)
```

```
<folium.folium.Map at 0x7fb75ed804c0>
```