WEEK-2

Task 1

• Implement the proposed algorithm for summarizing geospatial time series data using Google Earth Engine.

• Refer to GCBM tutorials on the Moja Global YouTube channel for guidance on working with Google Earth Engine.

• Write code to extract relevant data, perform necessary calculations, and generate visual outputs.

• Test the algorithm on sample datasets and ensure efficiency and accuracy.

• Document the algorithm's implementation details and share progress in the #18-outreachy channel.

Project 3:

Develop pipeline for high-throughput visualisation on Google Earth Engine

Time series processing using google earth engine

Time series processing using Google Earth Engine can be challenging, especially when dealing with Earth observation data. It's noisy by nature. In an ideal scenario, if the same satellite imaged the same object on two different days, you'd expect to see the same values, the same reflectances. However, reality is different. The presence of the atmosphere between the ground and the satellite introduces noise, including clouds and aerosols. Additionally, there are gaps in time series data due to cloud cover, obscuring what lies beneath.

To address this, we need to deal with these gaps and noise effectively. The goal is to clean up the data and extract the true signal from the noise. This technique enables us to achieve more accurate and reliable results in time series analysis.

We're considering implementing cloud masking techniques for the North India dataset to address issues arising from significant cloud cover. This involves removing areas with more than 30% cloud cover and adding masking for known cloud areas.

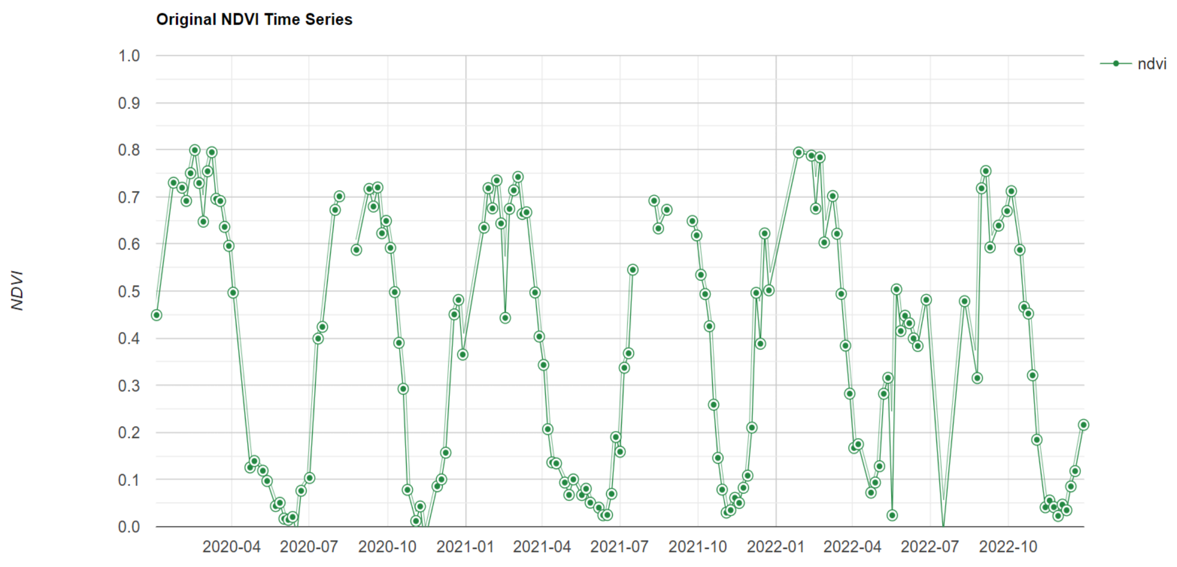
We aim to improve the quality of our data by addressing the imperfections in cloud masking. Specifically, we plan to remove known clouds and apply masking techniques. Once this is done, we'll proceed to map a function onto the dataset. At this stage, we'll have multiple images spanning the two-year period,

Using the given code

```
var s2 = ee.ImageCollection('COPERNICUS/S2_HARMONIZED');

var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
           qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

var originalCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2clouds)
  .map(addNDVI);


// Display a time-series chart
```

```
var chart = ui.Chart.image.series({
  imageCollection: originalCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Original NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },

    })
print(chart);

// Let's export the NDVI time-series as a video

var palette = ['#d73027','#f46d43','#fdae61','#fee08b',
  '#ffffbf','#d9ef8b','#a6d96a','#66bd63','#1a9850'];
var ndviVis = {min:-0.2, max: 0.8,  palette: palette}

Map.centerObject(geometry, 16);
var bbox = Map.getBounds({asGeoJSON: true});

var visualizeImage = function(image) {
  return image.visualize(ndviVis).clip(bbox).selfMask()
}

var visCollectionOriginal = originalCollection.select('ndvi')
  .map(visualizeImage)


Export.video.toDrive({
  collection: visCollectionOriginal,
  description: 'Original_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'original',
  framesPerSecond: 2,
  dimensions: 800,
  region: bbox})
```

Original NDVI time series



It seems like we're encountering significant noise and inconsistencies in our time series chart, which could be attributed to unmasked clouds and other data imperfections. The sharp drops and noisy patterns observed may not reflect the true behaviour of NDVI over time. Additionally, gaps in the data resulting from cloud masking further complicate the analysis.

It's important to recognize that earth observational time series data are multidimensional, comprising images with potential missing data, holes, and aberrations. To address these challenges and improve our analysis, we'll explore techniques to enhance the quality of our time series data and extract the true signal from the noise.

Moving Window Smoothing

In moving window smoothing, we apply a time window to compute the average value within that window. This technique helps to reduce fine-grained variations in the data by replacing each value with the average value calculated within the window. For example, if we select a time window of 15 days, we would replace each observation with the average of the values from 15 days before and 15 days after that observation. This smoothing process results in a more consistent and smooth time series, helping to remove noise and highlight underlying trends in the data.

In order to compute a moving average for each image in a time series, we need to find the images within a specified time window for each image. Let's say we're interested in calculating the moving average at a particular point. First, we'll identify all the images before and after that point, which fall within the defined time window. Once we have these images, we'll have observations within the time window, including the point we want to smooth. We'll then compute the average value of these observations, including the point of interest, and replace the value of that point with this average. This process needs to be repeated for every single point at every pixel across all images in the series.

We'll begin by using joins to accomplish this task.

Joins in Earth Engine

Joins in Google Earth Engine enable us to merge information from one collection, such as images or features, with items from another collection. This process is valuable for data fusion, where we integrate information from different sources to enhance analysis.

For instance, suppose we want to predict Sentinel-2 images based on Sentinel-1 data. Since Sentinel-1 can penetrate cloud cover and provide insights into vegetation growth, we might wish to fill gaps in our Sentinel-2 data with corresponding Sentinel-1 imagery. To achieve this, we need to identify Sentinel-1 images collected on the same day at specific locations. This requires joining millions of images from one collection with millions from another.

Moreover, we can perform joins between different types of collections. For example, we can join a feature collection with an image collection. This is particularly useful for data extraction, where we want to associate vector layers (e.g., points or polygons) with corresponding image data.

Additionally, joining feature collections to feature collections is another application, facilitating geoprocessing tasks by combining vector data effectively.

ImageCollection to ImageCollection(Useful in Data fusion)
FeatureCollection to ImageCollection(Useful in Data extraction)

FeatureCollection to FeatureCollection(Useful in Geoprocessing)

Before using joins in Google Earth Engine, there are three key steps we need to take:

1. Select the type of join: We need to decide whether to use `saveAll()`, `saveBest()`, or `saveFirst()` based on our matching criteria. These methods work by taking items from the secondary collection (the one we're trying to join) and saving the matching images as properties on the primary collection.

2. Choose a filter: We'll select a filter to compare properties of images from the primary and secondary collections. In our case, we want to find all images where the timestamp of one image matches the timestamp of another image. This involves using a binary filter to compare two properties from different collections. Commonly used filters include geometry-based filters or filters based on maximum time difference, as they allow us to compare images based on their timestamps.

3. Apply the join: Once we've set up these steps, we'll apply the join operation. This involves executing the join function in Google Earth Engine, which will match items from the secondary collection with items from the primary collection based on the specified criteria. For example, if we're using the `saveAll()` join and comparing timestamps to find images within a time window, we'll apply the join operation to identify all images that match the filter criteria. These matching images will then be saved as properties on the primary collection under the designated property name, such as "images".

CODE

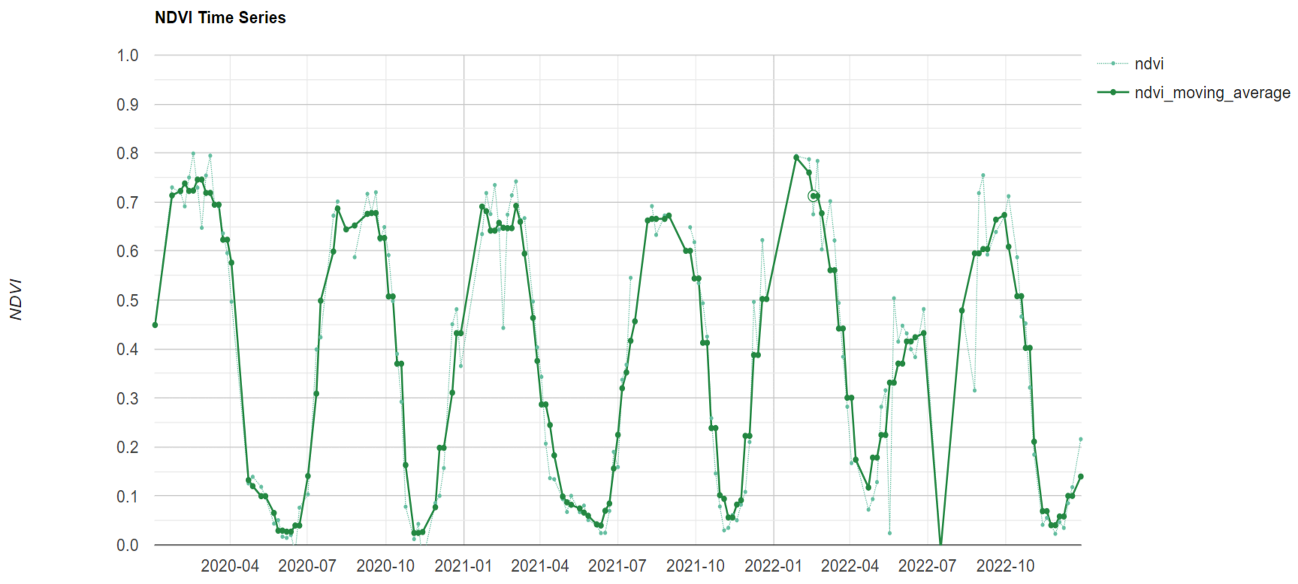
```
// Specify the time-window
var days = 15;

// Convert to milliseconds
var millis = ee.Number(days).multiply(1000*60*60*24)

var join = ee.Join.saveAll({
  matchesKey: 'images'
});

var diffFilter = ee.Filter.maxDifference({
  difference: millis,
  leftField: 'system:time_start',
  rightField: 'system:time_start'
});

var joinedCollection = join.apply({
  primary: originalCollection,
  secondary: originalCollection,
  condition: diffFilter
});

var extractAndComputeMean = function(image) {
  var matchingImages =
ee.ImageCollection.fromImages(image.get('images'));
  var meanImage =
matchingImages.reduce(ee.Reducer.mean())
  return ee.Image(image).addBands(meanImage)
}
var smoothedCollection =
joinedCollection.map(extractAndComputeMean);
```

Once the join operation is complete, the primary collection will have additional properties containing information from the secondary collection, allowing us to access and analyze the combined data for further processing.
Once the join operation is completed, each image in the primary collection will have a new property called "images", which contains the matched images based on the join criteria. In Google Earth Engine, properties serve as the storage mechanism, similar to an attribute table in desktop GIS software.

To further process the matched images, we need to extract them from the properties and perform computations. For example, we can write a function called `matchingImages.reduce(ee.Reducer.mean())`, which calculates the mean of the extracted images for each image in the collection. This function computes an average image based on the matched images for each image in the primary collection.

When applying this function, we want to preserve the original bands of the images and add the smoothed bands to them. This allows us to compare both the original and smoothed versions of the images. While we use the `ee.Reducer.mean()` here, you can also consider using `ee.Reducer.median()` if you prefer, as it is less sensitive to noise and outliers, such as cloudy pixels.

Once the function is applied to the collection, we obtain a smoothed collection. It's important to note that this process of multispectral smoothing isn't limited to just smoothing the NDVI band; it can be applied to multiple bands within the images.



Indeed, by applying the smoothing technique and filling the gaps with adjacent values, we have effectively reduced the sharp drops and smoothed out aberrations in the time series chart. The process of replacing the values with the average of surrounding images has helped to mitigate noise and inconsistencies in the data.

As a result, the time series chart appears much more visually appealing and provides a clearer representation of the underlying trends. With the aberrations smoothed and gaps filled, we can now confidently analyze the data and draw meaningful insights without being misled by noise or missing values.

Gap Filling technique

In this scenario, where we want to fill gaps caused by clouds in satellite observations without altering the original values, we can use an interpolation technique to replace those pixels. This method is particularly useful for creating gap-free time series, ensuring continuity in the data for analysis.

Here's how we can approach it:

1. Select a Time Window:Define a time window spanning before and after the gap where we'll search for replacement pixels.

2. Find Replacement Pixels: Traverse through the images within the time window after the gap. Look for the first unmasked (non-cloudy) pixels. If the first image is masked, continue searching until an unmasked pixel is found. This ensures that we select a pixel closest in time to the gap but still unmasked.

3. Apply Interpolation:Once replacement pixels are identified, use them to fill the gaps. Repeat this process for each pixel in the time series.

To implement this, we need to modify our join operation to include these additional steps. After identifying the time window and finding replacement pixels, we can use them to fill the gaps in our time series data. This approach ensures that we maintain the integrity of the original observations while filling in missing values caused by clouds or other factors.

To perform temporal interpolation for gap filling in our time series data, we begin by adding a band containing the timestamp to each image in our collection. This step is essential for identifying the time each pixel was collected, a crucial factor for later interpolation. Next, we define a suitable time window spanning before and after the gap, ensuring it's long enough to find replacement pixels but not too long to introduce significant interpolation errors. We apply a maximum difference filter to restrict the time window, ensuring that only images within a reasonable range from the current timestamp are considered for interpolation.

We then split the time window into two parts: before and after the current image's timestamp. By applying join operations, we add before and after images as properties to each image in our primary collection. This allows us to create a mosaic for each image, combining before and after images to create a flattened image where each pixel represents the first unmasked pixel in the stack.

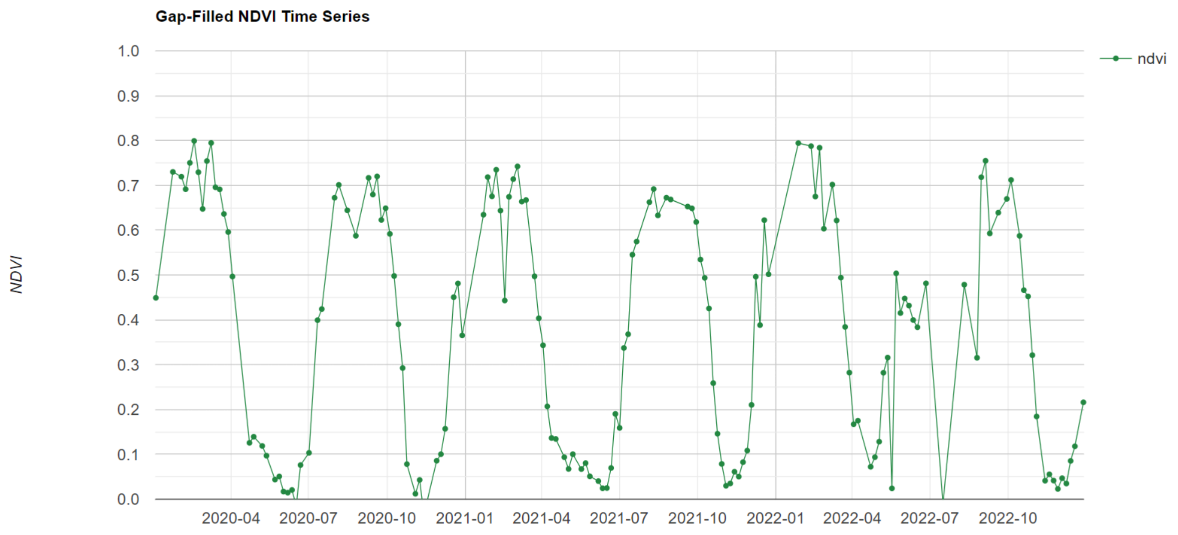
The next step involves temporal interpolation, where we calculate the weighted average of before and after images based on their timestamps and the current image's timestamp. This linear interpolation formula provides a suitable replacement value for the gap, ensuring smooth continuity in our time series data. Finally, we apply the interpolated values only to masked pixels, leaving the original values unchanged. This process, facilitated by the unmask() function, effectively fills gaps in our time series data, resulting in a continuous and gap-free dataset ready for analysis.

Code

```
var s2 = ee.ImageCollection('COPERNICUS/S2_HARMONIZED');

var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
             qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

var originalCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2clouds)
  .map(addNDVI);


// Display a time-series chart
var chart = ui.Chart.image.series({
  imageCollection: originalCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Original NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
    lineWidth: 1,
    pointSize: 4,
    series: {
      0: {color: '#238b45'},
    },

  })
print(chart);

// Gap-filling

// Add a band containing timestamp to each image
// This will be used to do pixel-wise interpolation later
var originalCollection = originalCollection.map(function(image) {
  var timeImage = image.metadata('system:time_start').rename('timestamp')
  // The time image doesn't have a mask.
  // We set the mask of the time band to be the same as the first band of the image
  var timeImageMasked = timeImage.updateMask(image.mask().select(0))
  return image.addBands(timeImageMasked).toFloat();
})

// For each image in the collection, we need to find all images
// before and after the specified time-window

// This is accomplished using Joins
// We need to do 2 joins
// Join 1: Join the collection with itself to find all images before each image
// Join 2: Join the collection with itself to find all images after each image

// We first define the filters needed for the join

// Define a maxDifference filter to find all images within the specified days
// The filter needs the time difference in milliseconds
// Convert days to milliseconds

// Specify the time-window to look for unmasked pixel
var days = 45;
var millis = ee.Number(days).multiply(1000*60*60*24)

var maxDiffFilter = ee.Filter.maxDifference({
  difference: millis,
  leftField: 'system:time_start',
  rightField: 'system:time_start'
})

// We need a lessThanOrEquals filter to find all images after a given image
// This will compare the given image's timestamp against other images' timestamps
var lessEqFilter = ee.Filter.lessThanOrEquals({
  leftField: 'system:time_start',
  rightField: 'system:time_start'
```

```
})

// We need a greaterThanOrEquals filter to find all images before a given image
// This will compare the given image's timestamp against other images' timestamps
var greaterEqFilter = ee.Filter.greaterThanOrEquals({
  leftField: 'system:time_start',
  rightField: 'system:time_start'
})


// Apply the joins

// For the first join, we need to match all images that are after the given image.
// To do this we need to match 2 conditions
// 1. The resulting images must be within the specified time-window of target image
// 2. The target image's timestamp must be lesser than the timestamp of resulting images
// Combine two filters to match both these conditions
var filter1 = ee.Filter.and(maxDiffFilter, lessEqFilter)
// This join will find all images after, sorted in descending order
// This will gives us images so that closest is last
var join1 = ee.Join.saveAll({
  matchesKey: 'after',
  ordering: 'system:time_start',
  ascending: false})

var join1Result = join1.apply({
  primary: originalCollection,
  secondary: originalCollection,
  condition: filter1
})
// Each image now as a property called 'after' containing
// all images that come after it within the time-window
print(join1Result.first())

// Do the second join now to match all images within the time-window
// that come before each image
var filter2 = ee.Filter.and(maxDiffFilter, greaterEqFilter)
// This join will find all images before, sorted in ascending order
// This will gives us images so that closest is last
var join2 = ee.Join.saveAll({
  matchesKey: 'before',
  ordering: 'system:time_start',
  ascending: true})

var join2Result = join2.apply({
  primary: join1Result,
  secondary: join1Result,
  condition: filter2
})
```

```
var joinedCol = join2Result;

// Each image now as a property called 'before' containing
// all images that come after it within the time-window
print(joinedCol.first())
// Do the gap-filling

// We now write a function that will be used to interpolate all images
// This function takes an image and replaces the masked pixels
// with the interpolated value from before and after images.

var interpolateImages = function(image) {
  var image = ee.Image(image);
  // We get the list of before and after images from the image property
  // Mosaic the images so we a before and after image with the closest unmasked pixel
  var beforeImages = ee.List(image.get('before'))
  var beforeMosaic = ee.ImageCollection.fromImages(beforeImages).mosaic()
  var afterImages = ee.List(image.get('after'))
  var afterMosaic = ee.ImageCollection.fromImages(afterImages).mosaic()

  // Interpolation formula
  // y = y1 + (y2-y1)*((t – t1) / (t2 – t1))
  // y = interpolated image
  // y1 = before image
  // y2 = after image
  // t = interpolation timestamp
  // t1 = before image timestamp
  // t2 = after image timestamp

  // We first compute the ratio (t – t1) / (t2 – t1)

  // Get image with before and after times
  var t1 = beforeMosaic.select('timestamp').rename('t1')
  var t2 = afterMosaic.select('timestamp').rename('t2')

  var t = image.metadata('system:time_start').rename('t')

  var timeImage = ee.Image.cat([t1, t2, t])

  var timeRatio = timeImage.expression('(t - t1) / (t2 - t1)', {
    't': timeImage.select('t'),
    't1': timeImage.select('t1'),
    't2': timeImage.select('t2'),
  })
  // You can replace timeRatio with a constant value 0.5
  // if you wanted a simple average

  // Compute an image with the interpolated image y
  var interpolated = beforeMosaic
    .add((afterMosaic.subtract(beforeMosaic).multiply(timeRatio)))
  // Replace the masked pixels in the current image with the average value
  var result = image.unmask(interpolated)
  return result.copyProperties(image, ['system:time_start'])
}

// map() the function to gap-fill all images in the collection
var gapFilledCol = ee.ImageCollection(joinedCol.map(interpolateImages))

// Display a time-series chart
var chart = ui.Chart.image.series({
  imageCollection: gapFilledCol.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Gap-Filled NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },
    })
print(chart);

// Let's visualize the NDVI time-series
Map.centerObject(geometry, 16);
var bbox = Map.getBounds({asGeoJSON: true});

var palette = ['#d73027','#f46d43','#fdae61','#fee08b','#ffffbf','#d9ef8b','#a6d96a','#66bd63','#1a9850'];
var ndviVis = {min:-0.2, max: 0.8,  palette: palette}

var visualizeImage = function(image) {
  return image.visualize(ndviVis).clip(bbox).selfMask()
}

var visCollectionOriginal = originalCollection.select('ndvi')
  .map(visualizeImage)

var visualizeIGapFilled = gapFilledCol.select('ndvi')
  .map(visualizeImage)


Export.video.toDrive({
  collection: visCollectionOriginal,
  description: 'Original_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'original',
  framesPerSecond: 2,
  dimensions: 800,
  region: bbox})

Export.video.toDrive({
  collection: visualizeIGapFilled,
  description: 'Gap_Filled_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'gap_filled',
  framesPerSecond: 2,
  dimensions: 800,
  region: bbox})
```



Regular time series

Regular time series interpolation is essential for ensuring uniformly spaced observations, crucial for many applications like time series analysis, regression modeling, and data exporting. Various operations and analyses require data at regular intervals, and irregularly spaced data may lead to inaccuracies or inconsistencies in results. To address this, we employ interpolation techniques to estimate values between existing data points, creating a regular time series dataset.

Interpolation involves estimating missing values by considering neighboring observations. Techniques such as linear interpolation or spline interpolation are commonly used for this purpose. By filling in the gaps between observations, we generate a regularly spaced time series, facilitating smoother data analysis and modeling.

To create a regular time series from irregularly spaced observations, we first define the desired time interval, such as every 5 days. We then generate empty images or placeholders at each interval throughout the time period. These placeholders are populated by filling the gaps with data from adjacent observations, effectively interpolating the missing values.

We achieve this by selecting the time interval and creating a list of timestamps representing each interval within the time period. Using these timestamps, we generate empty images with the same number of bands as the original dataset, ensuring compatibility and preserving data integrity. We then combine these empty images with the observed dataset, filling in the missing values using neighboring observations.

The resulting regular time series enables more consistent analysis, allowing for smoother data visualization, trend identification, and model fitting. Despite challenges like cloud cover or data artifacts, regular time series interpolation provides a reliable framework for data processing and analysis, facilitating accurate insights and decision-making in various domains.

Savitzky-Golay filter

We went through the process of creating a regularly spaced, gap-free, and interpolated time series for a specific purpose: to apply array-based functions like the Savitzky-Golay filter. This filter, originating from signal processing, is highly effective in filtering out high-frequency noise, making it particularly useful for vegetation time series analysis.

Now that we have our data prepared, we can apply the Savitzky-Golay filter to our time series. This filter works by fitting a polynomial through successive images, allowing us to choose the order of the polynomial (e.g., third order, second order). It then utilizes this polynomial to estimate values between observations, effectively smoothing out the data.

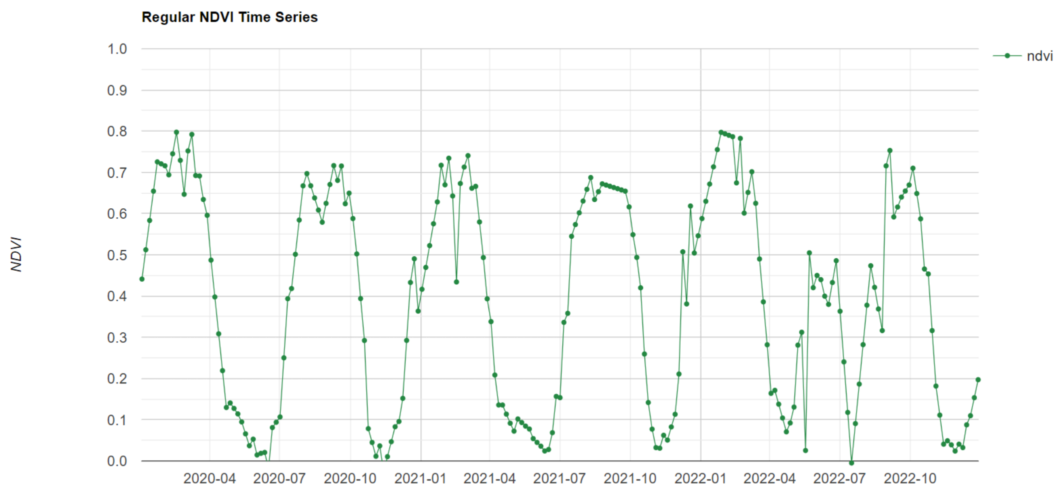
To implement this filter, we can leverage tools like the Open Earth Engine Library (OEEL), which provides functions and utilities for working with Google Earth Engine data. By applying the Savitzky-Golay filter to our regularly spaced, gap-free, and interpolated time series, we can enhance the quality of our data and extract meaningful insights with greater accuracy.

Code

```
var s2 = ee.ImageCollection('COPERNICUS/S2_HARMONIZED');

var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
            qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

var originalCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2clouds)
  .map(addNDVI);


// Display a time-series chart
var chart = ui.Chart.image.series({
  imageCollection: originalCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Original NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },

    })
print(chart);

// Prepare a regularly-spaced Time-Series

// Generate an empty multi-band image matching the bands
// in the original collection
var bandNames = ee.Image(originalCollection.first()).bandNames();
var numBands = bandNames.size();
var initBands = ee.List.repeat(ee.Image(), numBands);
var initImage = ee.ImageCollection(initBands).toBands().rename(bandNames)

// Select the interval. We will have 1 image every n days
var n = 5;
var firstImage = ee.Image(originalCollection.sort('system:time_start').first())
var lastImage = ee.Image(originalCollection.sort('system:time_start', false).first())
var timeStart = ee.Date(firstImage.get('system:time_start'))
var timeEnd = ee.Date(lastImage.get('system:time_start'))

var totalDays = timeEnd.difference(timeStart, 'day');
var daysToInterpolate = ee.List.sequence(0, totalDays, n)

var initImages = daysToInterpolate.map(function(day) {
  var image = initImage.set({
    'system:index': ee.Number(day).format('%d'),
    'system:time_start': timeStart.advance(day, 'day').millis(),
    // Set a property so we can identify interpolated images
    'type': 'interpolated'
  })
  return image
})
```

```
var initCol = ee.ImageCollection.fromImages(initImages)
print('Empty Collection', initCol)

// Merge original and empty collections
var originalCollection = originalCollection.merge(initCol)

// Interpolation

// Add a band containing timestamp to each image
// This will be used to do pixel-wise interpolation later
var originalCollection = originalCollection.map(function(image) {
  var timeImage = image.metadata('system:time_start').rename('timestamp')
  // The time image doesn't have a mask.
  // We set the mask of the time band to be the same as the first band of the image
  var timeImageMasked = timeImage.updateMask(image.mask().select(0))
  return image.addBands(timeImageMasked).toFloat();
})

// For each image in the collection, we need to find all images
// before and after the specified time-window

// This is accomplished using Joins
// We need to do 2 joins
// Join 1: Join the collection with itself to find all images before each image
// Join 2: Join the collection with itself to find all images after each image

// We first define the filters needed for the join

// Define a maxDifference filter to find all images within the specified days
// The filter needs the time difference in milliseconds
// Convert days to milliseconds

// Specify the time-window to look for unmasked pixel
var days = 45;
var millis = ee.Number(days).multiply(1000*60*60*24)

var maxDiffFilter = ee.Filter.maxDifference({
  difference: millis,
  leftField: 'system:time_start',
  rightField: 'system:time_start'
})
```

```
// We need a lessThanOrEquals filter to find all images after a given image
// This will compare the given image's timestamp against other images' timestamps
var lessEqFilter = ee.Filter.lessThanOrEquals({
  leftField: 'system:time_start',
  rightField: 'system:time_start'
})

// We need a greaterThanOrEquals filter to find all images before a given image
// This will compare the given image's timestamp against other images' timestamps
var greaterEqFilter = ee.Filter.greaterThanOrEquals({
  leftField: 'system:time_start',
  rightField: 'system:time_start'
})


// Apply the joins

// For the first join, we need to match all images that are after the given image.
// To do this we need to match 2 conditions
// 1. The resulting images must be within the specified time-window of target image
// 2. The target image's timestamp must be lesser than the timestamp of resulting
images
// Combine two filters to match both these conditions
var filter1 = ee.Filter.and(maxDiffFilter, lessEqFilter)
// This join will find all images after, sorted in descending order
// This will gives us images so that closest is last
var join1 = ee.Join.saveAll({
  matchesKey: 'after',
  ordering: 'system:time_start',
  ascending: false})

var join1Result = join1.apply({
  primary: originalCollection,
  secondary: originalCollection,
  condition: filter1
})
// Each image now as a property called 'after' containing
// all images that come after it within the time-window
print(join1Result.first())

// Do the second join now to match all images within the time-window
// that come before each image
var filter2 = ee.Filter.and(maxDiffFilter, greaterEqFilter)
// This join will find all images before, sorted in ascending order
// This will gives us images so that closest is last
var join2 = ee.Join.saveAll({
  matchesKey: 'before',
  ordering: 'system:time_start',
  ascending: true})

var join2Result = join2.apply({
  primary: join1Result,
  secondary: join1Result,
  condition: filter2
})

// Each image now as a property called 'before' containing
// all images that come after it within the time-window
print(join2Result.first())

var joinedCol = join2Result;

// Do the interpolation

// We now write a function that will be used to interpolate all images
// This function takes an image and replaces the masked pixels
// with the interpolated value from before and after images.

var interpolateImages = function(image) {
  var image = ee.Image(image);
  // We get the list of before and after images from the image property
  // Mosaic the images so we a before and after image with the closest unmasked
pixel
  var beforeImages = ee.List(image.get('before'))
  var beforeMosaic = ee.ImageCollection.fromImages(beforeImages).mosaic()
  var afterImages = ee.List(image.get('after'))
  var afterMosaic = ee.ImageCollection.fromImages(afterImages).mosaic()

  // Interpolation formula
  // y = y1 + (y2-y1)*((t – t1) / (t2 – t1))
  // y = interpolated image
  // y1 = before image
  // y2 = after image
  // t = interpolation timestamp
  // t1 = before image timestamp
  // t2 = after image timestamp

  // We first compute the ratio (t – t1) / (t2 – t1)

  // Get image with before and after times
  var t1 = beforeMosaic.select('timestamp').rename('t1')
  var t2 = afterMosaic.select('timestamp').rename('t2')

  var t = image.metadata('system:time_start').rename('t')

  var timeImage = ee.Image.cat([t1, t2, t])

  var timeRatio = timeImage.expression('(t - t1) / (t2 - t1)', {
    't': timeImage.select('t'),
    't1': timeImage.select('t1'),
    't2': timeImage.select('t2'),
  })
  // You can replace timeRatio with a constant value 0.5
  // if you wanted a simple average

  // Compute an image with the interpolated image y
  var interpolated = beforeMosaic
    .add((afterMosaic.subtract(beforeMosaic).multiply(timeRatio)))
  // Replace the masked pixels in the current image with the average value
  var result = image.unmask(interpolated)
  return result.copyProperties(image, ['system:time_start'])
}

// map() the function to interpolate all images in the collection
var interpolatedCol = ee.ImageCollection(joinedCol.map(interpolateImages))

// Once the interpolation are done, remove original images
// We keep only the generated interpolated images
var regularCol = interpolatedCol.filter(ee.Filter.eq('type', 'interpolated'))


// Display a time-series chart
var chart = ui.Chart.image.series({
  imageCollection: regularCol.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Regular NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },
    })
print(chart);

// Let's visualize the NDVI time-series
Map.centerObject(geometry, 16);
var bbox = Map.getBounds({asGeoJSON: true});

var palette =
['#d73027','#f46d43','#fdae61','#fee08b','#ffffbf','#d9ef8b','#a6d96a','#66bd63','#1a985
0'];
var ndviVis = {min:-0.2, max: 0.8,  palette: palette}

var visualizeImage = function(image) {
  return image.visualize(ndviVis).clip(bbox).selfMask()
}

var visCollectionOriginal = originalCollection.select('ndvi')
  .map(visualizeImage)

var visualizedRegular = regularCol.select('ndvi')
  .map(visualizeImage)


Export.video.toDrive({
  collection: visCollectionOriginal,
  description: 'Original_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'original',
  framesPerSecond: 2,
  dimensions: 800,
  region: bbox})

Export.video.toDrive({
  collection: visualizedRegular,
  description: 'Regular_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'regular',
  framesPerSecond: 5,
  dimensions: 800,
  region: bbox})
```



As you can observe, the NDVI signal now closely reflects the actual growth pattern without the interference of noise. The impact of clouds is virtually eliminated due to the sophisticated interpolation techniques employed, going beyond simple linear interpolation. The resulting time series closely mirrors what one would expect to observe on the ground.

With the data now in an image collection format, it can be exported for further analysis. This refined dataset is well-suited for tasks such as identifying harmonic parameters and conducting trend analysis. By implementing techniques like moving window smoothing, gap filling, interpolation, and advanced time series filtering, we have significantly improved the quality of the data, enabling more accurate and insightful analyses.