`ee.Image` constructor

Images can be loaded by pasting an Earth Engine asset ID into the `ee.Image` constructor. You can find image IDs in the data catalog. For example, to a digital elevation model (NASADEM):

```
loaded_image = ee.Image('NASA/NASADEM_HGT/001')
```

Note that finding an image through the Code Editor search tool is equivalent. When you import the asset, the image construction code is written for you in the imports section of the Code Editor. You can also use a personal asset ID as the argument to the `ee.Image` constructor.

Get an `ee.Image` from an `ee.ImageCollection`

The standard way to get an image out of a collection is to filter the collection, with filters in order of decreasing specificity. For example, to get an image out of the Sentinel-2 surface reflectance collection:

```
first = (
    ee.ImageCollection('COPERNICUS/S2_SR')
    .filterBounds(ee.Geometry.Point(-70.48, 43.3631))
    .filterDate('2019-01-01', '2019-12-31')
    .sort('CLOUDY_PIXEL_PERCENTAGE')
    .first()
)

# Define a map centered on southern Maine.
m = geemap.Map(center=[43.7516, -70.8155], zoom=11)

# Add the image layer to the map and display it.
m.add_layer(
    first, {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 2000},
'first'
)
display(m)
```

Images from Cloud GeoTIFFs

You can use `ee.Image.loadGeoTIFF()` to load images from Cloud Optimized GeoTIFFs in Google Cloud Storage. For example, the public Landsat dataset hosted in Google Cloud contains this GeoTIFF, corresponding to band 5 from a Landsat 8 scene. You can load this image from Cloud Storage using `ee.Image.loadGeoTIFF()`:

```
uri = (
    'gs://gcp-public-data-landsat/LC08/01/001/002/'
    + 'LC08_L1GT_001002_20160817_20170322_01_T2/'
    + 'LC08_L1GT_001002_20160817_20170322_01_T2_B5.TIF'
)
cloud_image = ee.Image.loadGeoTIFF(uri)
display(cloud_image)
```

Note that if you want to reload a Cloud Optimized GeoTIFF that you export from Earth Engine to Cloud Storage, when you do the export, set `cloudOptimized` to **true**

Images from Zarr v2 arrays

You can use `ee.Image.loadZarrV2Array()` to load an image from a Zarr v2 array in Google Cloud Storage. For example, the public ERA5 dataset hosted in Google Cloud contains this Zarr v2 array, corresponding to meters of water that has evaporated from the Earth's surface. You can load this array from Cloud Storage using `ee.Image.loadZarrV2Array()`:

```
time_start = 1000000
time_end = 1000010
zarr_v2_array_image = ee.Image.loadZarrV2Array(

uri='gs://gcp-public-data-arco-era5/ar/full_37-1h-0p25deg-chunk-1.zarr-v3/evaporation/.zarray',
    proj='EPSG:4326',
    starts=[time_start],
    ends=[time_end],
```

```
)

display(zarr_v2_array_image)

m.add_layer(
    zarr_v2_array_image, {'min': -0.0001, 'max': 0.00005},
'Evaporation'
)
m
```

Constant images

In addition to loading images by ID, you can also create images from constants, lists or other suitable Earth Engine objects. The following illustrates methods for creating images, getting band subsets, and manipulating bands:

```
# Create a constant image.
image_1 = ee.Image(1)
display(image_1)

# Concatenate two images into one multi-band image.
image_2 = ee.Image(2)
image_3 = ee.Image.cat([image_1, image_2])
display(image_3)

# Create a multi-band image from a list of constants.
multiband = ee.Image([1, 2, 3])
display(multiband)

# Select and (optionally) rename bands.
renamed = multiband.select(
    ['constant', 'constant_1', 'constant_2'],  # old names
    ['band1', 'band2', 'band3'],  # new names
)
display(renamed)

# Add bands to an image.
```

```
image_4 = image_3.addBands(ee.Image(42))
display(image_4)
```

Image Visualization

There are a number of `ee.Image` methods that produce RGB visual representations of image data, for example: `visualize()`, `getThumbURL()`, `getMap()`, `getMapId()` (used in Colab Folium map display) and, `Map.addLayer()` (used in Code Editor map display, not available for Python). By default these methods assign the first three bands to red, green and blue, respectively. The default stretch is based on the type of data in the bands (e.g. floats are stretched in [0, 1], 16-bit data are stretched to the full range of possible values), which may or may not be suitable. To achieve desired visualization effects, you can provide visualization parameters:

Parameter	Description	Type
bands	Comma-delimited list of three band names to be mapped to RGB	list
min	Value(s) to map to 0	number or list of three numbers, one for each band
max	Value(s) to map to 255	number or list of three numbers, one for each band

gain	Value(s) by which to multiply each pixel value	number or list of three numbers, one for each band
bias	Value(s) to add to each DN	number or list of three numbers, one for each band
gamma	Gamma correction factor(s)	number or list of three numbers, one for each band
palette	List of CSS-style color strings (single-band images only)	comma-separated list of hex strings
opacity	The opacity of the layer (0.0 is fully transparent and 1.0 is fully opaque)	number
format	Either "jpg" or "png"	string

RGB composites

The following illustrates the use of parameters to style a Landsat 8 image as a false-color composite:

```
# Load an image.
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Define the visualization parameters.
image_viz_params = {
    'bands': ['B5', 'B4', 'B3'],
```

```
    'min': 0,
    'max': 0.5,
    'gamma': [0.95, 1.1, 1],
}

# Define a map centered on San Francisco Bay.
map_l8 = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layer to the map and display it.
map_l8.add_layer(image, image_viz_params, 'false color composite')
display(map_l8)
```

Color palettes

To display a single band of an image in color, set the `palette` parameter with a color ramp represented by a list of CSS-style color strings. (See this reference for more information). The following example illustrates how to use colors from cyan (`'00FFFF'`) to blue (`'0000FF'`) to render a Normalized Difference Water Index (NDWI) image:

```
# Load an image.
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Create an NDWI image, define visualization parameters and display.
ndwi = image.normalizedDifference(['B3', 'B5'])
ndwi_viz = {'min': 0.5, 'max': 1, 'palette': ['00FFFF', '0000FF']}

# Define a map centered on San Francisco Bay.
map_ndwi = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layer to the map and display it.
map_ndwi.add_layer(ndwi, ndwi_viz, 'NDWI')
display(map_ndwi)
```

In this example, note that the `min` and `max` parameters indicate the range of pixel values to which the palette should be applied. Intermediate values are linearly stretched.

Also note that the `show` parameter is set to `false` in the Code Editor example. This results in the visibility of the layer being off when it is added to the map. It can always be turned on again using the Layer Manager in the upper right corner of the Code Editor map.

Saving default color palettes

To save color palettes on a classification image so that there is no need to remember to apply them, you can set two specially-named string image properties for each classification band.

For example, if your image has a band named `'landcover'` with three values 0, 1, and 2 corresponding to classes 'water', 'forest', and 'other', you can set the following properties to make the default visualization show a specified color for each class (the values used in the analysis will not be affected):

- `landcover_class_values="0,1,2"`
- `landcover_class_palette="0000FF,00FF00,AABBCD"`

See the managing assets page to learn how to set asset metadata.

Masking

You can use `image.updateMask()` to set the opacity of individual pixels based on where pixels in a mask image are non-zero. Pixels equal to zero in the mask are excluded from computations and the opacity is set to 0 for display. The following example uses an NDWI threshold (see the Relational Operations section for information on thresholds) to update the mask on the NDWI layer created previously:

```
# Mask the non-watery parts of the image, where NDWI < 0.4.
ndwi_masked = ndwi.updateMask(ndwi.gte(0.4))

# Define a map centered on San Francisco Bay.
map_ndwi_masked = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layer to the map and display it.
map_ndwi_masked.add_layer(ndwi_masked, ndwi_viz, 'NDWI masked')
display(map_ndwi_masked)
```

Visualization images

Use the `image.visualize()` method to convert an image into an 8-bit RGB image for display or export. For example, to convert the false-color composite and NDWI to 3-band display images, use:

```
image_rgb = image.visualize(bands=['B5', 'B4', 'B3'], max=0.5)
ndwi_rgb = ndwi_masked.visualize(min=0.5, max=1, palette=['00FFFF',
'0000FF'])
```

Mosaicking

You can use masking and `imageCollection.mosaic()` (see the Mosaicking section for information on mosaicking) to achieve various cartographic effects. The `mosaic()` method renders layers in the output image according to their order in the input collection. The following example uses `mosaic()` to combine the masked NDWI and the false color composite and obtain a new visualization:

```
# Mosaic the visualization layers and display (or export).
mosaic = ee.ImageCollection([image_rgb, ndwi_rgb]).mosaic()

# Define a map centered on San Francisco Bay.
map_mosaic = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layer to the map and display it.
map_mosaic.add_layer(mosaic, None, 'mosaic')
display(map_mosaic)
```

Clipping

The `image.clip()` method is useful for achieving cartographic effects. The following example clips the mosaic created previously to an arbitrary buffer zone around the city of San Francisco:

```
# Create a circle by drawing a 20000 meter buffer around a point.
roi = ee.Geometry.Point([-122.4481, 37.7599]).buffer(20000)
mosaic_clipped = mosaic.clip(roi)

# Define a map centered on San Francisco.
map_mosaic_clipped = geemap.Map(center=[37.7599, -122.4481], zoom=10)

# Add the image layer to the map and display it.
map_mosaic_clipped.add_layer(mosaic_clipped, None, 'mosaic clipped')
display(map_mosaic_clipped)
```

In the previous example, note that the coordinates are provided to the `Geometry` constructor and the buffer length is specified as 20,000 meters. Learn more about geometries on the Geometries page.

Rendering categorical maps

Palettes are also useful for rendering discrete valued maps, for example a land cover map. In the case of multiple classes, use the palette to supply a different color for each class. (The `image.remap()` method may be useful in this context, to convert arbitrary labels to consecutive integers). The following example uses a palette to render land cover categories:

```
# Load 2012 MODIS land cover and select the IGBP classification.
cover =
ee.Image('MODIS/051/MCD12Q1/2012_01_01').select('Land_Cover_Type_1')

# Define a palette for the 18 distinct land cover classes.
igbp_palette = [
    'aec3d4',  # water
    '152106',
```

```
    '225129',
    '369b47',
    '30eb5b',
    '387242',  # forest
    '6a2325',
    'c3aa69',
    'b76031',
    'd9903d',
    '91af40',  # shrub, grass
    '111149',  # wetlands
    'cdb33b',  # croplands
    'cc0013',  # urban
    '33280d',  # crop mosaic
    'd7cdcc',  # snow and ice
    'f7e084',  # barren
    '6f6f6f',  # tundra
]

# Define a map centered on the United States.
map_palette = geemap.Map(center=[40.413, -99.229], zoom=5)

# Add the image layer to the map and display it. Specify the min and
max labels
# and the color palette matching the labels.
map_palette.add_layer(
    cover, {'min': 0, 'max': 17, 'palette': igbp_palette}, 'IGBP
classes'
)
display(map_palette)
```

Styled Layer Descriptors

You can use a Styled Layer Descriptor (SLD) to render imagery for display. Provide `image.sldStyle()` with an XML description of the symbolization and coloring of the image, specifically the `RasterSymbolizer` element. Learn more about the `RasterSymbolizer` element here. For example, to render the land cover map described in the Rendering categorical maps section with an SLD, use:

```
cover =
ee.Image('MODIS/051/MCD12Q1/2012_01_01').select('Land_Cover_Type_1')

# Define an SLD style of discrete intervals to apply to the image.
sld_intervals = """
<RasterSymbolizer>
  <ColorMap  type="intervals" extended="false" >
    <ColorMapEntry color="#aec3d4" quantity="0" label="Water"/>
    <ColorMapEntry color="#152106" quantity="1" label="Evergreen
Needleleaf Forest"/>
    <ColorMapEntry color="#225129" quantity="2" label="Evergreen
Broadleaf Forest"/>
    <ColorMapEntry color="#369b47" quantity="3" label="Deciduous
Needleleaf Forest"/>
    <ColorMapEntry color="#30eb5b" quantity="4" label="Deciduous
Broadleaf Forest"/>
    <ColorMapEntry color="#387242" quantity="5" label="Mixed Deciduous
Forest"/>
    <ColorMapEntry color="#6a2325" quantity="6" label="Closed
Shrubland"/>
    <ColorMapEntry color="#c3aa69" quantity="7" label="Open
Shrubland"/>
    <ColorMapEntry color="#b76031" quantity="8" label="Woody Savanna"/>
    <ColorMapEntry color="#d9903d" quantity="9" label="Savanna"/>
    <ColorMapEntry color="#91af40" quantity="10" label="Grassland"/>
    <ColorMapEntry color="#111149" quantity="11" label="Permanent
Wetland"/>
    <ColorMapEntry color="#cdb33b" quantity="12" label="Cropland"/>
    <ColorMapEntry color="#cc0013" quantity="13" label="Urban"/>
    <ColorMapEntry color="#33280d" quantity="14" label="Crop, Natural
Veg. Mosaic"/>
    <ColorMapEntry color="#d7cdcc" quantity="15" label="Permanent Snow,
Ice"/>
    <ColorMapEntry color="#f7e084" quantity="16" label="Barren,
Desert"/>
    <ColorMapEntry color="#6f6f6f" quantity="17" label="Tundra"/>
  </ColorMap>
</RasterSymbolizer>"""

# Apply the SLD style to the image.
cover_sld = cover.sldStyle(sld_intervals)

# Define a map centered on the United States.
```

```
map_sld_categorical = geemap.Map(center=[40.413, -99.229], zoom=5)

# Add the image layer to the map and display it.
map_sld_categorical.add_layer(cover_sld, None, 'IGBP classes styled')
display(map_sld_categorical)
```

To create a visualization image with a color ramp, set the type of the `ColorMap` to 'ramp'. The following example compares the 'interval' and 'ramp' types for rendering a DEM:

```
# Load SRTM Digital Elevation Model data.
image = ee.Image('CGIAR/SRTM90_V4')

# Define an SLD style of discrete intervals to apply to the image.
sld_intervals = """
    <RasterSymbolizer>
      <ColorMap type="intervals" extended="false" >
        <ColorMapEntry color="#0000ff" quantity="0" label="0"/>
        <ColorMapEntry color="#00ff00" quantity="100" label="1-100" />
        <ColorMapEntry color="#007f30" quantity="200" label="110-200"
/>
        <ColorMapEntry color="#30b855" quantity="300" label="210-300"
/>
        <ColorMapEntry color="#ff0000" quantity="400" label="310-400"
/>
        <ColorMapEntry color="#ffff00" quantity="1000" label="410-1000"
/>
      </ColorMap>
    </RasterSymbolizer>"""

# Define an sld style color ramp to apply to the image.
sld_ramp = """
    <RasterSymbolizer>
      <ColorMap type="ramp" extended="false" >
        <ColorMapEntry color="#0000ff" quantity="0" label="0"/>
        <ColorMapEntry color="#00ff00" quantity="100" label="100" />
        <ColorMapEntry color="#007f30" quantity="200" label="200" />
        <ColorMapEntry color="#30b855" quantity="300" label="300" />
        <ColorMapEntry color="#ff0000" quantity="400" label="400" />
        <ColorMapEntry color="#ffff00" quantity="500" label="500" />
      </ColorMap>
    </RasterSymbolizer>"""
```

```
# Define a map centered on the United States.
map_sld_interval = geemap.Map(center=[40.413, -99.229], zoom=5)

# Add the image layers to the map and display it.
map_sld_interval.add_layer(
    image.sldStyle(sld_intervals), None, 'SLD intervals'
)
map_sld_interval.add_layer(image.sldStyle(sld_ramp), None, 'SLD ramp')
display(map_sld_interval)
```

SLDs are also useful for stretching pixel values to improve visualizations of continuous data. For example, the following code compares the results of an arbitrary linear stretch with a min-max 'Normalization' and a 'Histogram' equalization:

```
# Load a Landsat 8 raw image.
image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318')

# Define a RasterSymbolizer element with '_enhance_' for a placeholder.
template_sld = """
<RasterSymbolizer>
  <ContrastEnhancement><_enhance_/></ContrastEnhancement>
  <ChannelSelection>
    <RedChannel>
      <SourceChannelName>B5</SourceChannelName>
    </RedChannel>
    <GreenChannel>
      <SourceChannelName>B4</SourceChannelName>
    </GreenChannel>
    <BlueChannel>
      <SourceChannelName>B3</SourceChannelName>
    </BlueChannel>
  </ChannelSelection>
</RasterSymbolizer>"""

# Get SLDs with different enhancements.
equalize_sld = template_sld.replace('_enhance_', 'Histogram')
normalize_sld = template_sld.replace('_enhance_', 'Normalize')

# Define a map centered on San Francisco Bay.
```

```
map_sld_continuous = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layers to the map and display it.
map_sld_continuous.add_layer(
    image, {'bands': ['B5', 'B4', 'B3'], 'min': 0, 'max': 15000},
'Linear'
)
map_sld_continuous.add_layer(image.sldStyle(equalize_sld), None,
'Equalized')
map_sld_continuous.add_layer(
    image.sldStyle(normalize_sld), None, 'Normalized'
)
display(map_sld_continuous)
```

Points of note in reference to using SLDs in Earth Engine:

- OGC SLD 1.0 and OGC SE 1.1 are supported.
- The XML document passed in can be complete, or just the RasterSymbolizer element and down.
- Bands may be selected by their Earth Engine names or index ('1', '2', ...).
- The Histogram and Normalize contrast stretch mechanisms are not supported for floating point imagery.
- Opacity is only taken into account when it is 0.0 (transparent). Non-zero opacity values are treated as completely opaque.
- The OverlapBehavior definition is currently ignored.
- The ShadedRelief mechanism is not currently supported.
- The ImageOutline mechanism is not currently supported.
- The Geometry element is ignored.
- The output image will have histogram_bandname metadata if histogram equalization or normalization is requested.

Thumbnail images

Use the `ee.Image.getThumbURL()` method to generate a PNG or JPEG thumbnail image for an `ee.Image` object. Printing the outcome of an expression ending with a call to `getThumbURL()` results in a URL being printed. Visiting the URL sets Earth

Engine servers to work on generating the requested thumbnail on-the-fly. The image is displayed in a browser when processing completes. It can be downloaded by selecting appropriate options from the image's right-click context menu.

The `getThumbURL()` method includes parameters, described in the visualization parameters table above. Additionally, it takes optional `dimensions`, `region`, and `crs` arguments that control the spatial extent, size, and display projection of the thumbnail.

Parameter	Description	Type
dimensions	Thumbnail dimensions in pixel units. If a single integer is provided, it defines the size of the image's larger aspect dimension and scales the smaller dimension proportionally. Defaults to 512 pixels for the larger image aspect dimension.	A single integer or string in the format: 'WIDTHxHEIGHT'
region	The geospatial region of the image to render. The whole image by default, or the bounds of a provided geometry.	GeoJSON or a 2-D list of at least three point coordinates that define a linear ring
crs	The target projection e.g. 'EPSG:3857'. Defaults to WGS84 ('EPSG:4326').	String
format	Defines thumbnail format as either PNG or JPEG. The default PNG format is implemented as RGBA, where the alpha channel represents valid and invalid pixels, defined by the image's `mask()`. Invalid pixels are transparent.	String; either 'png' or 'jpg'

> The optional JPEG format is implemented as RGB, where invalid image pixels are zero filled across RGB channels.

A single-band image will default to grayscale unless a `palette` argument is supplied. A multi-band image will default to RGB visualization of the first three bands, unless a `bands` argument is supplied. If only two bands are provided, the first band will map to red, the second to blue, and the green channel will be zero filled.

The following are a series of examples demonstrating various combinations of `getThumbURL()` parameter arguments. Click on the URLs printed when you run this script to view the thumbnails.

```
# Fetch a digital elevation model.
image = ee.Image('CGIAR/SRTM90_V4')

# Request a default thumbnail of the DEM with defined linear stretch.
# Set masked pixels (ocean) to 1000 so they map as gray.
thumbnail_1 = image.unmask(1000).getThumbURL({
    'min': 0,
    'max': 3000,
    'dimensions': 500,
})
print('Default extent:', thumbnail_1)

# Specify region by rectangle, define palette, set larger aspect
dimension size.
thumbnail_2 = image.getThumbURL({
    'min': 0,
    'max': 3000,
    'palette': [
        '00A600',
        '63C600',
        'E6E600',
        'E9BD3A',
        'ECB176',
        'EFC2B3',
        'F2F2F2',
    ],
    'dimensions': 500,
```

```
    'region': ee.Geometry.Rectangle([-84.6, -55.9, -32.9, 15.7]),
})
print('Rectangle region and palette:', thumbnail_2)

# Specify region by a linear ring and set display CRS as Web Mercator.
thumbnail_3 = image.getThumbURL({
    'min': 0,
    'max': 3000,
    'palette': [
        '00A600',
        '63C600',
        'E6E600',
        'E9BD3A',
        'ECB176',
        'EFC2B3',
        'F2F2F2',
    ],
    'region': ee.Geometry.LinearRing(
        [[-84.6, 15.7], [-84.6, -55.9], [-32.9, -55.9]]
    ),
    'dimensions': 500,
    'crs': 'EPSG:3857',
})
print('Linear ring region and specified crs:', thumbnail_3)
```

Image Information and Metadata

```
# Load an image.
image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318')

# All metadata.
display('All metadata:', image)

# Get information about the bands as a list.
band_names = image.bandNames()
display('Band names:', band_names)  # ee.List of band names

# Get projection information from band 1.
```

```
b1_proj = image.select('B1').projection()
display('Band 1 projection:', b1_proj)  # ee.Projection object

# Get scale (in meters) information from band 1.
b1_scale = image.select('B1').projection().nominalScale()
display('Band 1 scale:', b1_scale)  # ee.Number

# Note that different bands can have different projections and scale.
b8_scale = image.select('B8').projection().nominalScale()
display('Band 8 scale:', b8_scale)  # ee.Number

# Get a list of all metadata properties.
properties = image.propertyNames()
display('Metadata properties:', properties)  # ee.List of metadata
properties

# Get a specific metadata property.
cloudiness = image.get('CLOUD_COVER')
display('CLOUD_COVER:', cloudiness)  # ee.Number

# Get version number (ingestion timestamp as microseconds since Unix
epoch).
version = image.get('system:version')
display('Version:', version)  # ee.Number
display(
    'Version (as ingestion date):',
    ee.Date(ee.Number(version).divide(1000)).format(),
)  # ee.Date

# Get the timestamp.
ee_date = ee.Date(image.get('system:time_start'))
display('Timestamp:', ee_date)  # ee.Date

# Date objects transferred to the client are milliseconds since UNIX
epoch;
# convert to human readable date with ee.Date.format().
display('Datetime:', ee_date.format())  # ISO standard date string
```

Mathematical Operations

Image math can be performed using operators like `add()` and `subtract()`, but for complex computations with more than a couple of terms, the `expression()` function provides a good alternative. See the following sections for more information on operators and expressions.

Operators

Math operators perform basic arithmetic operations on image bands. They take two inputs: either two images or one image and a constant term, which is interpreted as a single-band constant image with no masked pixels. Operations are performed per pixel for each band.

As a basic example, consider the task of calculating the Normalized Difference Vegetation Index (NDVI) using VIIRS imagery, where `add()`, `subtract()`, and `divide()` operators are used:

```
# Load a VIIRS 8-day surface reflectance composite for May 2024.
viirs202405 = (
    ee.ImageCollection('NASA/VIIRS/002/VNP09H1')
    .filter(ee.Filter.date('2024-05-01', '2024-05-16'))
    .first()
)

# Compute NDVI.
ndvi202405 = (
    viirs202405.select('SurfReflect_I2')
    .subtract(viirs202405.select('SurfReflect_I1'))
    .divide(
        viirs202405.select('SurfReflect_I2').add(
            viirs202405.select('SurfReflect_I1')
        )
    )
)
```

Only the intersection of unmasked pixels between the two inputs are considered and returned as unmasked, all else are masked. In general, if either input has only one band, then it is used against all the bands in the other input. If the inputs have the same number of bands, but not the same names, they're used pairwise in the natural order. The output bands are named for the longer of the two inputs, or if they're equal in length, in the first input's order. The type of the output pixels is the union of the input types.

The following example of multi-band image subtraction demonstrates how bands are matched automatically, resulting in a "change vector" for each pixel for each co-occurring band.

```
# Load a VIIRS 8-day surface reflectance composite for September 2024.
viirs202409 = (
    ee.ImageCollection('NASA/VIIRS/002/VNP09H1')
    .filter(ee.Filter.date('2024-09-01', '2024-09-16'))
    .first()
)

# Compute multi-band difference between the September composite and the
# previously loaded May composite.
diff = viirs202409.subtract(ndvi202405)

m = geemap.Map()
m.add_layer(
    diff,
    {
        'bands': ['SurfReflect_I1', 'SurfReflect_I2',
'SurfReflect_I3'],
        'min': -1,
        'max': 1,
    },
    'difference',
)

# Compute the squared difference in each band.
squared_difference = diff.pow(2)

m.add_layer(
    squared_difference,
    {
        'bands': ['SurfReflect_I1', 'SurfReflect_I2',
'SurfReflect_I3'],
```

```
        'min': 0,
        'max': 0.7,
    },
    'squared diff.',
)
display(m)
```

In the second part of this example, the squared difference is computed using `image.pow(2)`. For the complete list of mathematical operators handling basic arithmetic, trigonometry, exponentiation, rounding, casting, bitwise operations and more, see the API documentation.

API Reference

The API reference is divided into sections:

- The **Client Libraries** section is the API reference for both JavaScript and Python clients.
- The **Code Editor** section is the reference for functionality available through the Earth Engine Code Editor, an online IDE for Earth Engine JavaScript (learn more).
- The **REST API** section is the reference for the Earth Engine restful API.

Client Libraries

The open source JavaScript and Python Client libraries (GitHub repo) translate Earth Engine code into request objects sent to Earth Engine servers. The Earth Engine API is identical between the two languages with minor exceptions including syntactic and other differences described here. See this reference for methods available through the client libraries:

- The `ee` package for formulating requests to Earth Engine. See the example code for both JavaScript and Python on each method page.
- `Export` of data to Google Drive, Cloud Storage or Earth Engine assets (learn more).

Code Editor

The Code Editor provides the Earth Engine JavaScript client library plus:

- Display of geographic data on the `Map`.
- The `ui` package for creating user interfaces for Earth Engine apps (learn more).
- Other functions specific to the Code Editor (e.g. `print()`).

REST API

The REST API provides direct access to Earth Engine servers through HTTP. The reference describes available endpoints and the content of requests and responses. It also contains the REST Quickstart and other example workflows.

Expressions

To implement more complex mathematical expressions, consider using `image.expression()`, which parses a text representation of a math operation. The following example uses `expression()` to compute the Enhanced Vegetation Index (EVI):

```
# Load a Landsat 8 image.
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Compute the EVI using an expression.
evi = image.expression(
    '2.5 * ((NIR - RED) / (NIR + 6 * RED - 7.5 * BLUE + 1))',
    {
        'NIR': image.select('B5'),
        'RED': image.select('B4'),
        'BLUE': image.select('B2'),
    },
)
```

```
# Define a map centered on San Francisco Bay.
map_evi = geemap.Map(center=[37.4675, -122.1363], zoom=9)

# Add the image layer to the map and display it.
map_evi.add_layer(
    evi, {'min': -1, 'max': 1, 'palette': ['a6611a', 'f5f5f5',
'4dac26']}, 'evi'
)
display(map_evi)
```

Observe that the first argument to `expression()` is the textual representation of the math operation, the second argument is a dictionary where the keys are variable names used in the expression and the values are the image bands to which the variables should be mapped. Bands in the image may be referred to as `b("band name")` or `b(index)`, for example `b(0)`, instead of providing the dictionary. Bands can be defined from images other than the input when using the band map dictionary. Note that `expression()` uses "floor division", which discards the remainder and returns an integer when two integers are divided. For example `10 / 20 = 0`. To change this behavior, multiply one of the operands by `1.0`: `10 * 1.0 / 20 = 0.5`. Only the intersection of unmasked pixels are considered and returned as unmasked when bands from more than one source image are evaluated. Supported expression operators are listed in the following table.

Type	Symbol	Name
Arithmetic	+ - * / % **	Add, Subtract, Multiply, Divide, Modulus, Exponent
Relational	== != < > <= >=	Equal, Not Equal, Less Than, Greater than, etc.
Logical	&& || ! ^	And, Or, Not, Xor
Ternary	? :	If then else

Relational, Conditional, and Boolean Operations

`ee.Image` objects have a set of relational, conditional, and boolean methods for constructing decision-making expressions. The results of these methods are useful for limiting analysis to certain pixels or regions through masking, developing classified maps, and value reassignment.

Relational and boolean operators

Relational methods include:

`eq()`, `gt()`, `gte()`, `lt()`, and `lte()`

Boolean methods include:

Code Editor (JavaScript)

Colab (Python)

`And()`,`Or()`, and `Not()`

To perform per-pixel comparisons between images, use relational operators. To extract urbanized areas in an image, this example uses relational operators to threshold spectral indices, combining the thresholds with the *and* operator:

```
# Load a Landsat 8 image.
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Create NDVI and NDWI spectral indices.
ndvi = image.normalizedDifference(['B5', 'B4'])
ndwi = image.normalizedDifference(['B3', 'B5'])

# Create a binary layer using logical operations.
bare = ndvi.lt(0.2).And(ndwi.lt(0))

# Define a map centered on San Francisco Bay.
map_bare = geemap.Map(center=[37.7726, -122.3578], zoom=12)
```

```
# Add the masked image layer to the map and display it.
map_bare.add_layer(bare.selfMask(), None, 'bare')
display(map_bare)
```

The binary images that are returned by relational and boolean operators can be used with mathematical operators. This example creates zones of urbanization in a nighttime lights image using relational operators and `add()`:

```
# Load a 2012 nightlights image.
nl_2012 = ee.Image('NOAA/DMSP-OLS/NIGHTTIME_LIGHTS/F182012')
lights = nl_2012.select('stable_lights')

# Define arbitrary thresholds on the 6-bit stable lights band.
zones = lights.gt(30).add(lights.gt(55)).add(lights.gt(62))

# Define a map centered on Paris, France.
map_zones = geemap.Map(center=[48.8683, 2.373], zoom=8)

# Display the thresholded image as three distinct zones near Paris.
palette = ['000000', '0000FF', '00FF00', 'FF0000']
map_zones.add_layer(
    zones, {'min': 0, 'max': 3, 'palette': palette}, 'development
zones'
)
display(map_zones)
```

Conditional operators

Note that the code in the previous example is equivalent to using a ternary operator implemented by `expression()`:

```
# Create zones using an expression, display.
zones_exp = nl_2012.expression(
    "(b('stable_lights') > 62) ? 3 "
    ": (b('stable_lights') > 55) ? 2 "
    ": (b('stable_lights') > 30) ? 1 "
    ': 0'
)
```

```
# Define a map centered on Paris, France.
map_zones_exp = geemap.Map(center=[48.8683, 2.373], zoom=8)

# Add the image layer to the map and display it.
map_zones_exp.add_layer(
    zones_exp, {'min': 0, 'max': 3, 'palette': palette}, 'zones exp'
)
display(map_zones_exp)
```

Observe that in the previous expression example, the band of interest is referenced using the `b()` function, rather than a dictionary of variable names. Learn more about image expressions on this page. Using either mathematical operators or an expression will produce the same result.

Another way to implement conditional operations on images is with the `where()` operator. Consider the need to replace masked pixels with some other data. In the following example, cloudy pixels are replaced by pixels from a cloud-free image using `where()`:

```
# Load a cloudy Sentinel-2 image.
image =
ee.Image('COPERNICUS/S2_SR/20210114T185729_20210114T185730_T10SEG')

# Load another image to replace the cloudy pixels.
replacement = ee.Image(
    'COPERNICUS/S2_SR/20210109T185751_20210109T185931_T10SEG'
)

# Set cloudy pixels (greater than 5% probability) to the other image.
replaced = image.where(image.select('MSK_CLDPRB').gt(5), replacement)

# Define a map centered on San Francisco Bay.
map_replaced = geemap.Map(center=[37.7349, -122.3769], zoom=11)

# Display the images on a map.
vis_params = {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 2000}
map_replaced.add_layer(image, vis_params, 'original image')
map_replaced.add_layer(replaced, vis_params, 'clouds replaced')
```

```
display(map_replaced)
```

Convolutions

To perform linear convolutions on images, use `image.convolve()`. The only argument to convolve is an `ee.Kernel` which is specified by a shape and the weights in the kernel. Each pixel of the image output by `convolve()` is the linear combination of the kernel values and the input image pixels covered by the kernel. The kernels are applied to each band individually. For example, you might want to use a low-pass (smoothing) kernel to remove high-frequency information. The following illustrates a 15x15 low-pass kernel applied to a Landsat 8 image:

```
# Load the Landsat 8 TOA image
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Create an interactive map
Map = geemap.Map(center=[37.8694, -121.9785], zoom=11)

# Display the original image (False color: NIR, Red, Green)
Map.addLayer(
    image,
    {'bands': ['B5', 'B4', 'B3'], 'max': 0.5},
    'Input Image'
)

# Define a boxcar (low-pass) kernel
boxcar = ee.Kernel.square(
    radius=7,
    units='pixels',
    normalize=True
)

# Apply convolution (smoothing)
smooth = image.convolve(boxcar)

# Display the smoothed image
Map.addLayer(
    smooth,
```

```
    {'bands': ['B5', 'B4', 'B3'], 'max': 0.5},
    'Smoothed Image'
)
```

Morphological Operations

Earth Engine implements morphological operations as focal operations, specifically `focalMax()`, `focalMin()`, `focalMedian()`, and `focalMode()` instance methods in the `Image` class. (These are shortcuts for the more general `reduceNeighborhood()`, which can input the pixels in a kernel to any reducer with a numeric output. See this page for more information on reducing neighborhoods). The morphological operators are useful for performing operations such as erosion, dilation, opening and closing. For example, to perform an opening operation, use `focalMin()` followed by `focalMax()`:

```
# Load a Landsat 8 image, select the NIR band, apply threshold
image = (
    ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')
    .select(4)
    .gt(0.2)
)

# Center the map and display the thresholded image
Map.setCenter(-122.1899, 37.5010, 13)
Map.addLayer(image, {}, 'NIR threshold')

# Define a circular kernel
kernel = ee.Kernel.circle(radius=1)

# Perform erosion (focalMin) followed by dilation (focalMax)
opened = (
    image
    .focalMin(kernel=kernel, iterations=2)
    .focalMax(kernel=kernel, iterations=2)
)

# Display the opened image
Map.addLayer(opened, {}, 'opened')
```

Gradients

You can compute the gradient of each band of an image with `image.gradient()`. For example, the following code computes the gradient magnitude and direction of the Landsat 8 panchromatic band:

```
# Load a Landsat 8 image and select the panchromatic band
image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318').select('B8')

# Compute the image gradient in X and Y directions
xyGrad = image.gradient()

# Compute gradient magnitude
gradient = (
    xyGrad.select('x').pow(2)
    .add(xyGrad.select('y').pow(2))
    .sqrt()
)

# Compute gradient direction
direction = xyGrad.select('y').atan2(xyGrad.select('x'))

# Display the results
Map.setCenter(-122.054, 37.7295, 10)
Map.addLayer(direction, {'min': -2, 'max': 2, 'format': 'png'}, 'direction')
Map.addLayer(gradient, {'min': -7, 'max': 7, 'format': 'png'}, 'gradient')
```

Edge detection

Edge detection is applicable to a wide range of image processing tasks. In addition to the edge detection kernels described in the convolutions section, there are several specialized edge detection algorithms in Earth Engine. The Canny edge detection algorithm (Canny 1986) uses four separate filters to identify the diagonal, vertical, and horizontal edges. The calculation extracts the first derivative value for the horizontal and vertical directions and computes the gradient magnitude. Gradients of smaller magnitude are suppressed. To eliminate high-frequency noise, optionally pre-filter the image with a Gaussian kernel. For example

```
# Load a Landsat 8 image and select the panchromatic band
```

```
image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318').select('B8')

# Perform Canny edge detection
canny = ee.Algorithms.CannyEdgeDetector(
  image=image,
  threshold=10,
  sigma=1
)

# Display the result
Map.setCenter(-122.054, 37.7295, 10)
Map.addLayer(canny, {}, 'canny')
```

Note that the `threshold` parameter determines the minimum gradient magnitude and the `sigma` parameter is the standard deviation (SD) of a Gaussian pre-filter to remove high-frequency noise. For line extraction from an edge detector, Earth Engine implements the Hough transform (Duda and Hart 1972). Continuing the previous example, extract lines from the Canny detector with:

```
# Perform Hough transform on the Canny edge result
hough = ee.Algorithms.HoughTransform(canny, 256, 600, 100)

# Display the Hough transform result
Map.addLayer(hough, {}, 'hough')
```

Another specialized algorithm in Earth Engine is `zeroCrossing()`. A zero-crossing is defined as any pixel where the right, bottom, or diagonal bottom-right pixel has the opposite sign. If any of these pixels is of opposite sign, the current pixel is set to 1 (zero-crossing); otherwise it's set to zero. To detect edges, the zero-crossings algorithm can be applied to an estimate of the image second derivative. The following demonstrates using `zeroCrossing()` for edge detectio

```
# Load a Landsat 8 image and select the panchromatic band
image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318').select('B8')
Map.addLayer(image, {'max': 12000})

# Define a "fat" Gaussian kernel
fat = ee.Kernel.gaussian(
  radius=3,
  sigma=3,
  units='pixels',
```

```
    normalize=True,
    magnitude=-1
)

# Define a "skinny" Gaussian kernel
skinny = ee.Kernel.gaussian(
    radius=3,
    sigma=1,
    units='pixels',
    normalize=True
)

# Compute Difference-of-Gaussians (DoG) kernel
dog = fat.add(skinny)

# Compute zero crossings of the second derivative
zeroXings = image.convolve(dog).zeroCrossing()

# Display the result
Map.setCenter(-122.054, 37.7295, 10)
Map.addLayer(
    zeroXings.selfMask(),
    {'palette': 'FF0000'},
    'zero crossings'
)
```

Spectral transformations

There are several spectral transformation methods in Earth Engine. These include instance methods on images such as `normalizedDifference()`, `unmix()`, `rgbToHsv()` and `hsvToRgb()`.

Pan sharpening

Pan sharpening improves the resolution of a multiband image through enhancement provided by a corresponding panchromatic image with finer resolution. The `rgbToHsv()` and `hsvToRgb()` methods are useful for pan sharpening.

```
# Load a Landsat 8 top-of-atmosphere reflectance image.
```

```
image = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')

# Convert the RGB bands to the HSV color space.
hsv = image.select(['B4', 'B3', 'B2']).rgbToHsv()

# Swap in the panchromatic band and convert back to RGB.
sharpened = ee.Image.cat(
    [hsv.select('hue'), hsv.select('saturation'), image.select('B8')]
).hsvToRgb()

# Define a map centered on San Francisco, California.
map_sharpened = geemap.Map(center=[37.76664, -122.44829], zoom=13)

# Add the image layers to the map and display it.
map_sharpened.add_layer(
    image,
    {
        'bands': ['B4', 'B3', 'B2'],
        'min': 0,
        'max': 0.25,
        'gamma': [1.1, 1.1, 1],
    },
    'rgb',
)
map_sharpened.add_layer(
    sharpened,
    {'min': 0, 'max': 0.25, 'gamma': [1.3, 1.3, 1.3]},
    'pan-sharpened',
)
display(map_sharpened)
```

Spectral unmixing

Spectral unmixing is implemented in Earth Engine as the `image.unmix()` method. (For more flexible methods, see the Array Transformations page). The following is an example of unmixing Landsat 5 with predetermined urban, vegetation and water endmembers:

```
# Load a Landsat 5 image and select the bands we want to unmix.
```

```
bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7']
image =
ee.Image('LANDSAT/LT05/C02/T1/LT05_044034_20080214').select(bands)

# Define spectral endmembers.
urban = [88, 42, 48, 38, 86, 115, 59]
veg = [50, 21, 20, 35, 50, 110, 23]
water = [51, 20, 14, 9, 7, 116, 4]

# Unmix the image.
fractions = image.unmix([urban, veg, water])

# Define a map centered on San Francisco Bay.
map_fractions = geemap.Map(center=[37.5010, -122.1899], zoom=10)

# Add the image layers to the map and display it.
map_fractions.add_layer(
    image, {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 128}, 'image'
)
map_fractions.add_layer(fractions, None, 'unmixed')
display(map_fractions)
```

Texture

Earth Engine has several special methods for estimating spatial texture. When the image is discrete valued (not floating point), you can use `image.entropy()` to compute the entropy in a neighborhood:

 # Load a high-resolution NAIP image
image = ee.Image('USDA/NAIP/DOQQ/m_3712213_sw_10_1_20140613')

 # Zoom to San Francisco and display
Map.setCenter(-122.466123, 37.769833, 17)
Map.addLayer(image, {'max': 255}, 'image')

 # Select the NIR band
nir = image.select('N')

 # Define a square kernel
square = ee.Kernel.square(radius=4)

```
# Compute entropy
entropy = nir.entropy(square)

# Display entropy
Map.addLayer(
    entropy,
    {'min': 1, 'max': 5, 'palette': ['0000CC', 'CC0000']},
    'entropy'
)
```

Note that the NIR band is scaled to 8-bits prior to calling `entropy()` since the entropy computation takes discrete valued inputs. The non-zero elements in the kernel specify the neighborhood.

Another way to measure texture is with a gray-level co-occurrence matrix (GLCM). Using the image and kernel from the previous example, compute the GLCM-based contrast as follows:

```
# Compute GLCM texture
glcm = nir.glcmTexture(size=4)

# Select contrast texture
contrast = glcm.select('N_contrast')

# Display contrast
Map.addLayer(
    contrast,
    {'min': 0, 'max': 1500, 'palette': ['0000CC', 'CC0000']},
    'contrast'
)
```

Many measures of texture are output by `image.glcm()`. For a complete reference on the outputs, see Haralick et al. (1973) and Conners et al. (1984).

Local measures of spatial association such as Geary's C (Anselin 1995) can be computed in Earth Engine using `image.neighborhoodToBands()`. Using the image from the previous example:

```
# Create a list of weights for a 9x9 kernel
```

```
row = [1, 1, 1, 1, 1, 1, 1, 1, 1]

# Center of the kernel is zero
centerRow = [1, 1, 1, 1, 0, 1, 1, 1, 1]

# Assemble 9x9 kernel weights
rows = [row, row, row, row, centerRow, row, row, row, row]

# Create the fixed kernel
kernel = ee.Kernel.fixed(
   9, 9, rows,
   -4, -4,
   False
)

# Convert neighborhood to multiple bands
neighs = nir.neighborhoodToBands(kernel)

# Compute local Geary's C
gearys = (
   nir.subtract(neighs)
   .pow(2)
   .reduce(ee.Reducer.sum())
   .divide(9 ** 2)
)

# Display Geary's C
Map.addLayer(
   gearys,
   {'min': 20, 'max': 2500, 'palette': ['0000CC', 'CC0000']},
   "Geary's C"
)
```

Object-based methods

Image objects are sets of connected pixels having the same integer value. Categorical, binned, and boolean image data are suitable for object analysis.

Earth Engine offers methods for labeling each object with a unique ID, counting the number of pixels composing objects, and computing statistics for values of pixels that intersect objects.

- `connectedComponents()`: label each object with a unique identifier.
- `connectedPixelCount()`: compute the number of pixels in each object.
- `reduceConnectedComponents()`: compute a statistic for pixels in each object.

Caution: results of object-based methods depend on scale, which is determined by:

- the requested scale of an output (e.g., `Export.image.toAsset()` or `Export.image.toDrive()`).
- functions that require a scale of analysis (e.g., `reduceRegions()` or `reduceToVectors()`).
- Map zoom level.

Take special note of scale determined by Map zoom level. Results of object-based methods will vary when viewing or inspecting image layers in the Map, as each pyramid layer has a different scale. To force a desired scale of analysis in Map exploration, use `reproject()`. However, it is strongly recommended that you NOT use `reproject()` because the entire area visible in the Map will be requested at the set scale and projection. At large extents this can cause too much data to be requested, often triggering errors. Within the image pyramid-based architecture of Earth Engine, scale and projection need only be set for operations that provide `scale` and `crs` as parameters. See Scale of Analysis and Reprojecting for more information.

Thermal hotspots

The following sections provide examples of object-based methods applied to Landsat 8 surface temperature with each section building on the former. Run the next snippet to generate the base image: thermal hotspots (> 303 degrees Kelvin) for a small region of San Francisco.

```
# Make an area of interest geometry centered on San Francisco.
point = ee.Geometry.Point(-122.1899, 37.5010)
aoi = point.buffer(10000)

# Import a Landsat 8 image, subset the thermal band, and clip to the
# area of interest.
kelvin = (
    ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')
    .select(['B10'], ['kelvin'])
```

```
    .clip(aoi)
)

# Threshold the thermal band to set hot pixels as value 1, mask all
else.
hotspots = kelvin.gt(303).selfMask().rename('hotspots')

# Define a map centered on Redwood City, California.
map_objects = geemap.Map(center=[37.5010, -122.1899], zoom=13)

# Add the image layers to the map.
map_objects.add_layer(kelvin, {'min': 288, 'max': 305}, 'Kelvin')
map_objects.add_layer(hotspots, {'palette': 'FF0000'}, 'Hotspots')
```

Label objects

Labeling objects is often the first step in object analysis. Here, the `connectedComponents()` function is used to identify image objects and assign a unique ID to each; all pixels belonging to an object are assigned the same integer ID value. The result is a copy of the input image with an additional "labels" band associating pixels with an object ID value based on connectivity of pixels in the first band of the image.

```
# Uniquely label the hotspot image objects.
object_id = hotspots.connectedComponents(
    connectedness=ee.Kernel.plus(1), maxSize=128
)

# Add the uniquely ID'ed objects to the map.
map_objects.add_layer(object_id.randomVisualizer(), None, 'Objects')
```

Note that the maximum patch size is set to 128 pixels; objects composed of more pixels are masked. The connectivity is specified by an `ee.Kernel.plus(1)` kernel, which defines four-neighbor connectivity; use `ee.Kernel.square(1)` for eight-neighbor.

Object size

Number of pixels

Calculate the number of pixels composing objects using the `connectedPixelCount()` image method. Knowing the number of pixels in an object can be helpful for masking objects by size and calculating object area. The following snippet applies `connectedPixelCount()` to the "labels" band of the `objectId` image defined in the previous section.

```
# Compute the number of pixels in each object defined by the "labels"
band.
object_size = object_id.select('labels').connectedPixelCount(
    maxSize=128, eightConnected=False
)

# Add the object pixel count to the map.
map_objects.add_layer(object_size, None, 'Object n pixels')
```

`connectedPixelCount()` returns a copy of the input image where each pixel of each band contains the number of connected neighbors according to either the four- or eight-neighbor connectivity rule determined by a boolean argument passed to the `eightConnected` parameter. Note that connectivity is determined independently for each band of the input image. In this example, a single-band image (`objectId`) representing object ID was provided as input, so a single-band image was returned with a "labels" band (present as such in the input image), but now the values represent the number of pixels composing objects; every pixel of each object will have the same pixel count value.

Area

Calculate object area by multiplying the area of a single pixel by the number of pixels composing an object (determined by `connectedPixelCount()`). Pixel area is provided by an image generated from `ee.Image.pixelArea()`.

```
# Get a pixel area image.
pixel_area = ee.Image.pixelArea()

# Multiply pixel area by the number of pixels in an object to calculate
# the object area. The result is an image where each pixel
# of an object relates the area of the object in m^2.
object_area = object_size.multiply(pixel_area)

# Add the object area to the map.
map_objects.add_layer(
    object_area,
    {'min': 0, 'max': 30000, 'palette': ['0000FF', 'FF00FF']},
    'Object area m^2',
)
```

The result is an image where each pixel of an object relates the area of the object in square meters. In this example, the `objectSize` image contains a single band, if it were multi-band, the multiplication operation would be applied to each band of the image.

Filter objects by size

Object size can be used as a mask condition to focus your analysis on objects of a certain size (e.g., mask out objects that are too small). Here the `objectArea` image calculated in the previous step is used as a mask to remove objects whose area are less than one hectare.

```
# Threshold the `object_area` image to define a mask that will mask out
# objects below a given size (1 hectare in this case).
area_mask = object_area.gte(10000)
```

```
# Update the mask of the `object_id` layer defined previously using the
# minimum area mask just defined.
object_id = object_id.updateMask(area_mask)
map_objects.add_layer(object_id, None, 'Large hotspots')
```

Zonal statistics

The `reduceConnectedComponents()` method applies a reducer to the pixels composing unique objects. The following snippet uses it to calculate the mean temperature of hotspot objects. `reduceConnectedComponents()` requires an input image with a band (or bands) to be reduced and a band that defines object labels. Here, the `objectID` "labels" image band is added to the `kelvin` temperature image to construct a suitable input image.

```
# Make a suitable image for `reduceConnectedComponents()` by adding a
label
# band to the `kelvin` temperature image.
kelvin = kelvin.addBands(object_id.select('labels'))

# Calculate the mean temperature per object defined by the previously
added
# "labels" band.
patch_temp = kelvin.reduceConnectedComponents(
    reducer=ee.Reducer.mean(), labelBand='labels'
)

# Add object mean temperature to the map and display it.
map_objects.add_layer(
    patch_temp,
    {'min': 303, 'max': 304, 'palette': ['yellow', 'red']},
    'Mean temperature',
)
display(map_objects)
```

The result is a copy of the input image without the band used to define objects, where pixel values represent the result of the reduction per object, per band.

Cumulative Cost Mapping

Use `image.cumulativeCost()` to compute a cost map where every pixel contains the total cost of the lowest cost path to the nearest source location. This process is useful in a variety of contexts such as habitat analysis (Adriaensen et al. 2003), watershed delineation (Melles et al. 2011) and image segmentation (Falcao et al. 2004). Call the cumulative cost function on an image in which each pixel represents the cost per meter to traverse it. Paths are computed through any of a pixel's eight neighbors. Required inputs include a `source` image, in which each non-zero pixel represents a potential source (or start of a path), and a `maxDistance` (in meters) over which to compute paths. The algorithm finds the cumulative cost of all paths less than *maxPixels* = `maxDistance`/*scale* in length, where *scale* is the pixel resolution, or scale of analysis in Earth Engine.

The following example demonstrates computing least-cost paths across a land cover image:

```
# Rectangle representing Bangui, Central African Republic

geometry = ee.Geometry.Rectangle([18.5229, 4.3491, 18.5833, 4.4066])


# Create a source image where the geometry is 1 and everything else is 0

sources = ee.Image().toByte().paint(geometry, 1)
```

```
# Mask the sources image with itself
sources = sources.selfMask()


# Load GlobCover land cover and select the class band
cover = ee.Image('ESA/GLOBCOVER_L4_200901_200912_V2_3').select(0)


# Remap land-cover classes to cost values
beforeRemap = [
    60, 80, 110, 140,
    40, 90, 120, 130, 170,
    50, 70, 150, 160
]


afterRemap = [
    1, 1, 1, 1,
```

```
    2, 2, 2, 2, 2,
    3, 3, 3, 3
]

cost = cover.remap(beforeRemap, afterRemap, 0)

# Compute cumulative cost distance (up to 80 km)
cumulativeCost = cost.cumulativeCost(
    source=sources,
    maxDistance=80 * 1000
)

# Display the results
Map.setCenter(18.71, 4.2, 9)
Map.addLayer(cover, {}, 'Globcover')
Map.addLayer(cumulativeCost, {'min': 0, 'max':
5e4}, 'accumulated cost')
Map.addLayer(geometry, {'color': 'FF0000'},
'source geometry')
```

The result should look something like Figure 1, in which each output pixel represents the accumulated cost to the nearest source. Note that discontinuities can appear in places where the least cost path to the nearest source exceeds *maxPixels* in length.

Registering Images

The Earth Engine image registration algorithm is designed to be a final, post-ortho, fine-grained step in aligning images. It is assumed that the images to be registered have already gone through initial alignment stages, so they are already within a few degrees of rotation of one another, and differ by only small translations. The registration uses a "rubber-sheet" technique, allowing local image warping to correct for orthorectification errors and other artifacts from earlier processing. The underlying alignment technique is image correlation, so the bands for the input and reference images must be visually similar in order for the algorithm to compute an accurate alignment.

Image displacement

There are two steps to registering an image: Determining the displacement image using `displacement()`, and then applying it with `displace()`. The required inputs are the pair of images to register, and a maximum displacement parameter (`maxOffset`).

The `displacement()` algorithm takes a reference image, a maximum displacement parameter (`maxOffset`), and two optional parameters that modify the algorithm behaviour. The output is a displacement image with bands `dx` and `dy` which give the X and Y components (in meters) of the displacement vector at each pixel.

All bands of the calling and reference images are used for matching during registration, so the number of bands must be exactly equal. The input bands must be visually similar for registration to succeed. If that is not the case, it may be possible to pre-process them (e.g. smoothing, edge detection) to make them appear more similar. The registration computations are performed using a multiscale,

coarse-to-fine process, with (multiscale) working projections that depend on three of the projections supplied to the algorithm:

1. the default projection of the calling image (P_c)
2. the default projection of the reference image (P_r)
3. the output projection (P_o)

The highest resolution working projection (P_w will be in the CRS of P_r, at a scale determined by the coarsest resolution of these 3 projections, to minimize computation. The results from P_r are then resampled to be in the projection specified by the input 'projection' parameter.

The output is a displacement image with the following bands:

`dx`

For a given reference image pixel location, this band contains the distance in the X direction that must be travelled to arrive at the matching location in the calling image. Units are in geodesic meters.

`dy`

For a given reference image pixel location, this band contains the distance in the Y direction that must be travelled to arrive at the matching location in the calling image. Units are in geodesic meters.

`confidence`

This is a per-pixel estimate of displacement confidence (where 0 is low confidence and 1 is high confidence) based on the correlation scores in regions where valid matches were found. In regions where no matches were found, confidence is estimated from nearby correlations using a Gaussian kernel to provide higher weight to nearby correlations.

The following example computes the magnitude and angle of displacement between two high-resolution Terra Bella images:

```
# Load the two images to be registered
image1 =
ee.Image('SKYSAT/GEN-A/PUBLIC/ORTHO/MULTISPECTRAL/s01_20150502T082736Z')
image2 =
ee.Image('SKYSAT/GEN-A/PUBLIC/ORTHO/MULTISPECTRAL/s01_20150305T081019Z')
```

```
# Use bicubic resampling during registration
image1Orig = image1.resample('bicubic')
image2Orig = image2.resample('bicubic')

# Select the red band for registration
image1RedBand = image1Orig.select('R')
image2RedBand = image2Orig.select('R')

# Determine displacement by matching red bands
displacement = image2RedBand.displacement(
  referenceImage=image1RedBand,
  maxOffset=50.0,
  patchWidth=100.0
)

# Compute offset magnitude and direction
offset = displacement.select('dx').hypot(displacement.select('dy'))
angle = displacement.select('dx').atan2(displacement.select('dy'))

# Display results
Map.addLayer(offset, {'min': 0, 'max': 20}, 'offset')
Map.addLayer(angle, {'min': -3.14159, 'max': 3.14159}, 'angle')
Map.setCenter(37.44, 0.58, 15)
```

Warping an image

There are two ways to warp an image to match another image: `displace()` or `register()`. The `displace()` algorithm takes a displacement image having `dx` and `dy` bands as the first two bands, and warps the image accordingly. The output image will be the result of warping the bands of the input image by the offsets present in the displacement image. Using the displacements computed in the previous example:

```
# Use the computed displacement to register all original bands
registered = image2Orig.displace(displacement)

# Visualization parameters
```

```
visParams = {'bands': ['R', 'G', 'B'], 'max': 4000}

# Display results
Map.addLayer(image1Orig, visParams, 'Reference')
Map.addLayer(image2Orig, visParams, 'Before Registration')
Map.addLayer(registered, visParams, 'After Registration')
```

If you don't need the displacement bands, Earth Engine provides the `register()` method, which is a shortcut for calling `displacement()` followed by `displace()`. For example:

```
alsoRegistered = image2Orig.register(
    referenceImage=image1Orig,
    maxOffset=50.0,
    patchWidth=100.0
)

Map.addLayer(alsoRegistered, visParams, 'Also Registered')
```

In this example, the results of `register()` differ from the results of `displace()`. This is because a different set of bands was used in the two approaches: `register()` always uses all bands of the input images, while the `displacement()` example used only the red band before feeding the result to `displace()`. Note that when multiple bands are used, if band variances are very different this could over-weight the high-variance bands, since the bands are jointly normalized when their spatial correlation scores are combined. This illustrates the importance of selecting band(s) that are visually the most similar when registering. As in the previous example, use `displacement()` and `displace()` for control over which bands are used to compute displacement.

ImageCollection Overview

An `ImageCollection` is a stack or sequence of images.

Construct from a collection ID

An `ImageCollection` can be loaded by pasting an Earth Engine asset ID into the `ImageCollection` constructor. You can find `ImageCollection` IDs in the data catalog. For example, to load the Sentinel-2 surface reflectance collection:

```
sentinel_collection = ee.ImageCollection('COPERNICUS/S2_SR')
```

This collection contains every Sentinel-2 image in the public catalog. There are a lot. Usually you want to filter the collection as shown

As with Images, there are a variety of ways to get information about an `ImageCollection`. The collection can be printed directly to the console, but the console printout is limited to 5000 elements. Collections larger than 5000 images will need to be filtered before printing. Printing a large collection will be correspondingly slower. The following example shows various ways of getting information about image collections programmatically:

```
# Load a Landsat 8 ImageCollection for a single path-row.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
    .filterDate('2014-03-01', '2014-08-01')
)
display('Collection:', collection)

# Get the number of images.
count = collection.size()
display('Count:', count)

# Get the date range of images in the collection.
range = collection.reduceColumns(ee.Reducer.minMax(),
['system:time_start'])
display('Date range:', ee.Date(range.get('min')),
ee.Date(range.get('max')))
```

```
# Get statistics for a property of the images in the collection.
sun_stats = collection.aggregate_stats('SUN_ELEVATION')
display('Sun elevation statistics:', sun_stats)

# Sort by a cloud cover property, get the least cloudy image.
image = ee.Image(collection.sort('CLOUD_COVER').first())
display('Least cloudy image:', image)

# Limit the collection to the 10 most recent images.
recent = collection.sort('system:time_start', False).limit(10)
display('Recent images:', recent)
```

Construct from an image list

The constructor `ee.ImageCollection()` or the convenience method `ee.ImageCollection.fromImages()` create image collections from lists of images. You can also create new image collections by merging existing collections. For example:

```
# Create arbitrary constant images.
constant_1 = ee.Image(1)
constant_2 = ee.Image(2)

# Create a collection by giving a list to the constructor.
collection_from_constructor = ee.ImageCollection([constant_1,
constant_2])
display('Collection from constructor:', collection_from_constructor)

# Create a collection with fromImages().
collection_from_images = ee.ImageCollection.fromImages(
    [ee.Image(3), ee.Image(4)]
)
display('Collection from images:', collection_from_images)

# Merge two collections.
merged_collection =
collection_from_constructor.merge(collection_from_images)
display('Merged collection:', merged_collection)
```

```
# Create a toy FeatureCollection
features = ee.FeatureCollection(
    [ee.Feature(None, {'foo': 1}), ee.Feature(None, {'foo': 2})]
)

# Create an ImageCollection from the FeatureCollection
# by mapping a function over the FeatureCollection.
images = features.map(lambda feature:
ee.Image(ee.Number(feature.get('foo'))))

# Display the resultant collection.
display('Image collection:', images)
```

Note that in this example an `ImageCollection` is created by mapping a function that returns an `Image` over a `FeatureCollection`. Learn more about mapping in the Mapping over an ImageCollection section. Learn more about feature collections from the FeatureCollection section.

Construct from a COG list

Create an `ImageCollection` from GeoTiffs in Cloud Storage. For example:

```
# All the GeoTiffs are in this folder.
uri_base = (
    'gs://gcp-public-data-landsat/LC08/01/001/002/'
    + 'LC08_L1GT_001002_20160817_20170322_01_T2/'
)

# List of URIs, one for each band.
uris = ee.List([
    uri_base + 'LC08_L1GT_001002_20160817_20170322_01_T2_B2.TIF',
    uri_base + 'LC08_L1GT_001002_20160817_20170322_01_T2_B3.TIF',
    uri_base + 'LC08_L1GT_001002_20160817_20170322_01_T2_B4.TIF',
    uri_base + 'LC08_L1GT_001002_20160817_20170322_01_T2_B5.TIF',
])

# Make a collection from the list of images.
images = uris.map(lambda uri: ee.Image.loadGeoTIFF(uri))
collection = ee.ImageCollection(images)

# Get an RGB image from the collection of bands.
rgb = collection.toBands().rename(['B2', 'B3', 'B4', 'B5'])
m = geemap.Map()
m.center_object(rgb)
m.add_layer(rgb, {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 20000},
'rgb')
m
```

Construct from a Zarr v2 array

Create an `ImageCollection` from a Zarr v2 array in Cloud Storage by taking image slices along a higher dimension. For example:

```
time_start = 1000000
time_end = 1000048
zarr_v2_array_images = ee.ImageCollection.loadZarrV2Array(

uri='gs://gcp-public-data-arco-era5/ar/full_37-1h-0p25deg-chunk-1.zarr-v3/evaporation/.zarray',
    proj='EPSG:4326',
    axis=0,
    starts=[time_start],
    ends=[time_end],
)

display(zarr_v2_array_images)

m = geemap.Map()
m.add_layer(
    zarr_v2_array_images, {'min': -0.0001, 'max': 0.00005},
'Evaporation'
)
m
```

ImageCollection Visualization

Images composing an `ImageCollection` can be visualized as either an animation or a series of thumbnails referred to as a “filmstrip”. These methods provide a quick assessment of the contents of an `ImageCollection` and an effective medium for witnessing spatiotemporal change (Figure 1).

- `getVideoThumbURL()` produces an animated image series
- `getFilmstripThumbURL()` produces a thumbnail image series

The following sections describe how to prepare an `ImageCollection` for visualization, provide example code for each collection visualization method, and cover several advanced animation techniques.

Collection preparation

Filter, composite, sort, and style images within a collection to display only those of interest or emphasize a phenomenon. Any `ImageCollection` can be provided as input to the visualization functions, but a curated collection with consideration of inter- and intra-annual date ranges, observation interval, regional extent, quality and representation can achieve better results.

Filtering

Filter an image collection to include only relevant data that supports the purpose of the visualization. Consider dates, spatial extent, quality, and other properties specific to a given dataset.

For instance, filter a Sentinel-2 surface reflectance collection by:

a single date range,

```
s2col = (
  ee.ImageCollection('COPERNICUS/S2_SR')
  .filterDate('2018-01-01', '2019-01-01')
)
```

```
s2col = (
    ee.ImageCollection('COPERNICUS/S2_SR')
    .filter(ee.Filter.calendarRange(171, 242, 'day_of_year'))
)
```

```
s2col = (
    ee.ImageCollection('COPERNICUS/S2_SR')
    .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 50))
)
```

```
s2col = (
    ee.ImageCollection('COPERNICUS/S2_SR')
    .filterDate('2018-01-01', '2019-01-01')
    .filterBounds(ee.Geometry.Point(-122.1, 37.2))
    .filter('CLOUDY_PIXEL_PERCENTAGE < 50')
)
```

Compositing

Composite intra- and inter-annual date ranges to reduce the number of images in a collection and improve quality. For example, suppose you were to create a visualization of annual NDVI for Africa. One option is to simply filter a MODIS 16-day NDVI collection to include all 2018 observations.

```
ndviCol = (
    ee.ImageCollection('MODIS/006/MOD13A2')
    .filterDate('2018-01-01', '2019-01-01')
    .select('NDVI')
)
```

Inter-annual composite by filter and reduce

Visualization of the above collection shows considerable noise in the forested regions where cloud cover is heavy (Figure 2a). A better representation can be achieved by reducing serial date ranges by median across all years in the MODIS collection.

```
# Make a day-of-year sequence from 1 to 365 with a 16-day step
doyList = ee.List.sequence(1, 365, 16)

# Import MODIS NDVI collection
ndviCol = ee.ImageCollection('MODIS/006/MOD13A2').select('NDVI')

# Map over the DOY list to build composite images
ndviCompList = doyList.map(
  lambda startDoy: ndviCol
    .filter(
      ee.Filter.calendarRange(
        ee.Number(startDoy),
        ee.Number(startDoy).add(15),
        'day_of_year'
      )
    )
    .reduce(ee.Reducer.median())
)

# Convert the Image list to an ImageCollection
ndviCompCol = ee.ImageCollection.fromImages(ndviCompList)
```

The animation resulting from this collection is less noisy, as each image represents the median of a 16-day NDVI composite for 20+ years of data (Figure 1b). See this tutorial for more information on this animation.

Intra-annual composite by filter and reduce

The previous example applies inter-annual compositing. It can also be helpful to composite a series of intra-annual observations. For example, Landsat data are

collected every sixteen days for a given scene per sensor, but often some portion of the images are obscured by clouds. Masking out the clouds and compositing several images from the same season can produce a more cloud-free representation. Consider the following example where Landsat 5 images from July and August are composited using median for each year from 1985 to 2011.

```
# Assemble Landsat 5 Surface Reflectance collection
lsCol = (
    ee.ImageCollection('LANDSAT/LT05/C02/T1_L2')
    .filterBounds(ee.Geometry.Point(-122.9, 43.6))
    .filter(ee.Filter.dayOfYear(182, 243))
    # Add observation year as a property
    .map(lambda img: img.set('year', ee.Image(img).date().get('year')))
)

# Define scaling and masking function
def maskL457sr(image):
    # QA masks
    qaMask = (
        image.select('QA_PIXEL')
        .bitwiseAnd(int('11111', 2))
        .eq(0)
    )
    saturationMask = image.select('QA_RADSAT').eq(0)

    # Apply scaling factors
    opticalBands = image.select('SR_B.').multiply(0.0000275).add(-0.2)
    thermalBand = image.select('ST_B6').multiply(0.00341802).add(149.0)

    # Replace original bands and apply masks
    return (
        image.addBands(opticalBands, None, True)
        .addBands(thermalBand, None, True)
        .updateMask(qaMask)
        .updateMask(saturationMask)
    )

# Get unique observation years
years = (
    ee.List(lsCol.aggregate_array('year'))
    .distinct()
    .sort()
```

```
)

# Build annual median composites
lsCompList = years.map(
    lambda year: (
        lsCol
        .filterMetadata('year', 'equals', year)
        .map(maskL457sr)
        .reduce(ee.Reducer.median())
        .set('year', year)
    )
)

# Convert Image list to ImageCollection
lsCompCol = ee.ImageCollection.fromImages(lsCompList)
```

Intra-annual composite by join and reduce

Note that the previous two compositing methods map over a `List` of days and years to incrementally define new dates to filter and composite over. Applying a join is another method for achieving this operation. In the following snippet, a unique year collection is defined and then a `saveAll` join is applied to identify all images that correspond to a given year. Images belonging to a given year are grouped into a `List` object which is stored as a property of the respective year representative in the distinct year collection. Annual composites are generated from these lists by reducing `ImageCollections` defined by them in a function mapped over the distinct year collection.

```
# Assemble Landsat 5 Surface Reflectance collection
lsCol = (
    ee.ImageCollection('LANDSAT/LT05/C02/T1_L2')
    .filterBounds(ee.Geometry.Point(-122.9, 43.6))
    .filter(ee.Filter.dayOfYear(182, 243))
    # Add observation year as a property
    .map(lambda img: img.set('year', ee.Image(img).date().get('year')))
)

# Make a distinct year collection (one image per year)
distinctYears = lsCol.distinct('year').sort('year')
```

```
# Define a one-to-many join on the 'year' property
filter_ = ee.Filter.equals(leftField='year', rightField='year')
join = ee.Join.saveAll('year_match')

# Apply the join
joinCol = join.apply(distinctYears, lsCol, filter_)

# Define scaling and masking function
def maskL457sr(image):
    qaMask = (
        image.select('QA_PIXEL')
        .bitwiseAnd(int('11111', 2))
        .eq(0)
    )
    saturationMask = image.select('QA_RADSAT').eq(0)

    opticalBands = image.select('SR_B.').multiply(0.0000275).add(-0.2)
    thermalBand = image.select('ST_B6').multiply(0.00341802).add(149.0)

    return (
        image.addBands(opticalBands, None, True)
        .addBands(thermalBand, None, True)
        .updateMask(qaMask)
        .updateMask(saturationMask)
    )

# Build annual median composites using the joined collections
lsCompList = joinCol.map(
    lambda img: (
        ee.ImageCollection.fromImages(img.get('year_match'))
        .map(maskL457sr)
        .reduce(ee.Reducer.median())
        .copyProperties(img, ['year'])
    )
)

# Convert to ImageCollection
lsCompCol = ee.ImageCollection(lsCompList)
```

Same-day composite by join and reduce

An additional case for compositing is to create spatially contiguous image mosaics. Suppose your region of interest spans two Landsat rows within the same path and your objective is to display an image mosaic of the two images for each Landsat 8 orbit in 2017 and 2018. Here, after filtering the collection by path and row, a join operation is used to mosaic Landsat images from the same obit, defined by acquisition date.

```
lsCol = (
   ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
   .filterDate('2017-01-01', '2019-01-01')
   .filter('WRS_PATH == 38 && (WRS_ROW == 28 || WRS_ROW == 29)')
   .map(
      lambda img: img.set(
         'date',
         img.date().format('yyyy-MM-dd')
      )
   )
)

# Get distinct acquisition dates
distinctDates = lsCol.distinct('date').sort('date')

# Define join on 'date' property
filter_ = ee.Filter.equals(leftField='date', rightField='date')
join = ee.Join.saveAll('date_match')

# Apply the join
joinCol = join.apply(distinctDates, lsCol, filter_)

# Mosaic images acquired on the same date
lsColMos = ee.ImageCollection(
   joinCol.map(
      lambda col: ee.ImageCollection
         .fromImages(col.get('date_match'))
         .mosaic()
   )
)
```

Sorting

Sort a collection by time to ensure proper chronological sequence, or order by a property of your choice. By default, the visualization frame series is sorted in natural order of the collection. The arrangement of the series can be altered using the `sort` collection method, whereby an `Image` property is selected for sorting in either ascending or descending order. For example, to sort by time of observation, use the ubiquitous `system:time_start` property.

```
s2col = (
   ee.ImageCollection('COPERNICUS/S2_SR')
   .filterBounds(ee.Geometry.Point(-122.1, 37.2))
   .sort('system:time_start')
)
```

perhaps the order should be defined by increasing cloudiness, as in this case of Sentinel-2 imagery.

```
s2col = (
   ee.ImageCollection('COPERNICUS/S2_SR')
   .filterBounds(ee.Geometry.Point(-122.1, 37.2))
   .sort('CLOUDY_PIXEL_PERCENTAGE')
)
```

Order can also be defined by a derived property, such as mean regional NDVI. Here, regional NDVI is added as a property to each image in a mapped function, followed by a sort on the new property.

```
# Define area of interest
aoi = ee.Geometry.Point(-122.1, 37.2).buffer(1e4)

# Filter MODIS NDVI collection and compute regional mean NDVI
ndviCol = (
   ee.ImageCollection('MODIS/061/MOD13A1')
   .filterDate('2018-01-01', '2019-01-01')
   .select('NDVI')
   .map(
      lambda img: img.set(
         'meanNdvi',
         img.reduceRegion(
            reducer=ee.Reducer.mean(),
            geometry=aoi,
            scale=500
```

```
        ).get('NDVI')
      )
    )
    # Sort by descending mean NDVI
    .sort('meanNdvi', False)
)
```

Image visualization

Image visualization transforms numbers into colors. There are three ways to control how image data are represented as color in collection visualization methods:

1. Provide visualization arguments directly to `getVideoThumbURL` and `getFilmstripThumbURL`.
2. Map the `visualize` image method over the image collection prior to application of `getVideoThumbURL` and `getFilmstripThumbURL`.
3. Map the `sldStyle` image method over the image collection prior to application of `getVideoThumbURL` and `getFilmstripThumbURL`. See Styled Layer Descriptor for more information.

The examples in this guide use options 1 and 2, where visualization is achieved by mapping three image bands of a multi-band image to color channels red, green, and blue or grading values of a single band linearly along a color palette. Visualization parameters include:

Parameter	Description	Type
bands	Comma-delimited list of three band names to be mapped to RGB	list
min	Value(s) to map to 0	number or list of three numbers, one for each band

max	Value(s) to map to 255	number or list of three numbers, one for each band
gain	Value(s) by which to multiply each pixel value	number or list of three numbers, one for each band
bias	Value(s) to add to each DN	number or list of three numbers, one for each band
gamma	Gamma correction factor(s)	number or list of three numbers, one for each band
palette	List of CSS-style color strings (single-band images only)	comma-separated list of hex strings
opacity	The opacity of the layer (0.0 is fully transparent and 1.0 is fully opaque)	number

Use the `bands` argument to select the band(s) you wish to visualize. Provide a list of either one or three band names. With regard to multi-band images, the first three bands are selected by default. Band name order determines color assignment; the first, second, and third listed bands are mapped to red, green, and blue, respectively.

Data range scaling is an important consideration when visualizing images. By default, floating point data values between 0 and 1 (inclusive) are scaled between 0 and 255 (inclusive). Values outside this range are forced to 0 and 255 depending on whether they are less than 0 or greater than 1, respectively. With regard to integer data, the full capacity defined by its type is scaled between 0 and 255 (e.g., signed 16-bit data has a range from −32,768 to 32,767, which scales to [0, 255], by default). Accepting the defaults can often result in visualizations with little to no contrast between image features. Use `min` and `max` to improve contrast and emphasize a particular data range. A good rule of thumb is to set `min` and `max` to values that

represent the 2nd and 98th percentile of the data within your area of interest. See the following example of calculating these values for a digital elevation model.

```
# Import SRTM global elevation model
demImg = ee.Image('USGS/SRTMGL1_003')

# Define rectangular area of interest
aoi = ee.Geometry.Polygon(
    [[
        [-103.84153083119054, 49.083004219142886],
        [-103.84153083119054, 25.06838270664608],
        [-85.64817145619054, 25.06838270664608],
        [-85.64817145619054, 49.083004219142886]
    ]],
    None,
    False
)

# Calculate 2nd and 98th percentile elevation values
percentClip = demImg.reduceRegion(
    reducer=ee.Reducer.percentile([2, 98]),
    geometry=aoi,
    scale=500,
    maxPixels=3e7
)

# Get dictionary keys
keys = percentClip.keys()

# Print visualization limits
print('Set vis min to:',
      ee.Number(percentClip.get(keys.get(0))).round())
print('Set vis max to:',
      ee.Number(percentClip.get(keys.get(1))).round())
```

The `palette` parameter defines the colors to represent the 8-bit visualization image. It applies only to single-band representations; specifying it with a multi-band image results in an error. If the data are single-band or you would like to visualize a single band from a multi-band image, set the `forceRgbOutput` parameter to `true` (unnecessary if the `palette` argument is provided). Use the `min` and `max` parameters to define the range of values to linearly scale between 0 and 255.

An example of mapping a visualization function over a single-band image collection follows. A MODIS NDVI collection is imported, visualization arguments for the `visualization` method are set, and a function that transforms values into RGB image representations is mapped over the NDVI collection.

```
# Filter MODIS NDVI image collection
ndviCol = (
    ee.ImageCollection('MODIS/061/MOD13A1')
    .filterDate('2018-01-01', '2019-01-01')
    .select('NDVI')
)

# Visualization arguments
visArgs = {
    'min': 0,
    'max': 9000,
    'palette': [
        'FFFFFF', 'CE7E45', 'DF923D', 'F1B555', 'FCD163', '99B718', '74A901',
        '66A000', '529400', '3E8601', '207401', '056201', '004C00', '023B01',
        '012E01', '011D01', '011301'
    ]
}

# Function to convert image to RGB visualization and copy properties
def visFun(img):
    return img.visualize(visArgs).copyProperties(img, img.propertyNames())

# Map visualization function over the collection
ndviColVis = ndviCol.map(visFun)
```

Here is an example of mapping a visualization function over a multi-band image collection:

```
# Assemble Sentinel-2 Surface Reflectance collection
s2col = (
    ee.ImageCollection('COPERNICUS/S2_SR')
    .filterBounds(ee.Geometry.Point(-96.9037, 48.0395))
    .filterDate('2019-06-01', '2019-10-01')
)
```

```
# Visualization arguments
visArgs = {
  'bands': ['B11', 'B8', 'B3'],
  'min': 300,
  'max': 3500
}

# Function to convert image to RGB visualization and copy properties
def visFun(img):
  return img.visualize(visArgs).copyProperties(img, img.propertyNames())

# Map visualization function over the collection
s2colVis = s2col.map(visFun)
```

In this case, no palette argument is provided because three bands are given, which define intensity for each RGB layer. Note that both examples use the `min` and `max` parameters to control what values are stretched to the limits of the 8-bit RGB data.

Video thumb

The `getVideoThumbURL()` function generates an animation from all images in an `ImageCollection` where each image represents a frame. The general workflow for producing an animation is as follows:

1. Define a `Geometry` whose bounds determine the regional extent of the animation.
2. Define an `ImageCollection`.
3. Consider image visualization: either map an image visualization function over the collection or add image visualization arguments to the set of animation arguments.
4. Define animation arguments and call the `getVideoThumbURL` method.

The result of `getVideoThumbURL` is a URL. Print the URL to the console and click it to start Earth Engine servers generating the animation on-the-fly in a new browser tab. Alternatively, view the animation in the Code Editor console by calling the `ui.Thumbnail` function on the collection and its corresponding animation arguments. Upon rendering, the animation is available for downloading by right clicking on it and selecting appropriate options from its context menu.

The following example illustrates generating an animation depicting global temperatures over the course of 24 hours. Note that this example includes visualization arguments along with animation arguments, as opposed to first mapping a visualization function over the `ImageCollection`. Upon running this script, an animation similar to Figure 3 should appear in the Code Editor console.

```
# Define area of interest (global non-polar extent)
aoi = ee.Geometry.Polygon(
    [[
        [-179.0, 78.0],
        [-179.0, -58.0],
        [179.0, -58.0],
        [179.0, 78.0]
    ]],
    None,
    False
)

# Import hourly predicted temperature image collection
tempCol = (
    ee.ImageCollection('NOAA/GFS0P25')
    .filterDate('2018-12-22', '2018-12-23')
    .limit(24)
    .select('temperature_2m_above_ground')
)

# Animation arguments
videoArgs = {
    'dimensions': 768,
    'region': aoi,
    'framesPerSecond': 7,
    'crs': 'EPSG:3857',
    'min': -40.0,
    'max': 35.0,
    'palette': ['blue', 'purple', 'cyan', 'green', 'yellow', 'red']
}

# Generate and print the animation URL
print(tempCol.getVideoThumbURL(videoArgs))
```

Filmstrip

The `getFilmstripThumbUrl` function generates a single static image representing the concatenation of all images in an `ImageCollection` into a north-south series. The sequence of filmstrip frames follow the natural order of the collection.

The result of `getFilmstripThumbUrl` is a URL. Print the URL to the console and click it to start Earth Engine servers generating the image on-the-fly in a new browser tab. Upon rendering, the image is available for downloading by right clicking on it and selecting appropriate options from its context menu.

The following code snippet uses the same collection as the video thumb example above. Upon running this script, a filmstrip similar to Figure 4 should appear in the Code Editor console.

```
# Define area of interest (global non-polar extent)
aoi = ee.Geometry.Polygon(
    [[
        [-179.0, 78.0],
        [-179.0, -58.0],
        [179.0, -58.0],
        [179.0, 78.0]
    ]],
    None,
    False
)

# Import hourly predicted temperature image collection
tempCol = (
    ee.ImageCollection('NOAA/GFS0P25')
    .filterDate('2018-12-22', '2018-12-23')
    .limit(24)
    .select('temperature_2m_above_ground')
)

# Filmstrip arguments
filmArgs = {
    'dimensions': 128,
    'region': aoi,
    'crs': 'EPSG:3857',
    'min': -40.0,
    'max': 35.0,
```

```
    'palette': ['blue', 'purple', 'cyan', 'green', 'yellow', 'red']
}

# Generate and print the filmstrip URL
print(tempCol.getFilmstripThumbURL(filmArgs))
```

Advanced techniques

The following sections describe how to use clipping, opacity, and layer compositing to enhance visualizations by adding polygon borders, emphasizing regions of interest, and comparing images within a collection.

Note that all of the following examples in this section use the same base `ImageCollection` defined here:

```
# Import hourly predicted temperature image collection
tempCol = (
    ee.ImageCollection('NOAA/GFS0P25')
    .filterDate('2018-12-22', '2018-12-23')
    .limit(24)
    .select('temperature_2m_above_ground')
)

# Visualization arguments
visArgs = {
    'min': -40.0,
    'max': 35.0,
    'palette': ['blue', 'purple', 'cyan', 'green', 'yellow', 'red']
}

# Convert each image to RGB visualization
tempColVis = tempCol.map(
    lambda img: img.visualize(visArgs)
)

# Import country features and filter to South America
southAmCol = (
```

```
    ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017')
    .filterMetadata('wld_rgn', 'equals', 'South America')
)

# Define animation region (South America with buffer)
southAmAoi = ee.Geometry.Rectangle(
    [-103.6, -58.8, -18.4, 17.4],
    None,
    False
)
```

Overlays

Multiple images can be overlaid using the `blend` `Image` method where overlapping pixels from two images are blended based on their masks (opacity).

Vector overlay

Adding administrative boundary polygons and other geometries to an image can provide valuable spatial context. Consider the global daily surface temperature animation above (Figure 3). The boundaries between land and ocean are somewhat discernible, but they can be made explicit by adding a polygon overlay of countries.

Vector data (`Features`) are drawn to images by applying the `paint` method. Features can be painted to an existing image, but the better practice is to paint them to a blank image, style it, and then blend the result with other styled image layers. Treating each layer of a visualization stack independently affords more control over styling.

The following example demonstrates painting South American country borders to a blank `Image` and blending the result with each `Image` of the global daily temperature collection (Figure 5). The overlaid country boundaries distinguish land from water and provide context to patterns of temperature.

```
# Define an empty image to paint features to
empty = ee.Image().byte()

# Paint country feature edges and visualize as black lines
southAmOutline = (
    empty
```

```
    .paint(
      featureCollection=southAmCol,
      color=1,
      width=1
    )
    .visualize({'palette': '000000'})
)

# Blend country outlines with each temperature visualization image
tempColOutline = tempColVis.map(
    lambda img: img.blend(southAmOutline)
)

# Animation arguments
videoArgs = {
    'dimensions': 768,
    'region': southAmAoi,
    'framesPerSecond': 7,
    'crs': 'EPSG:3857'
}

# Generate and print animation URL
print(tempColOutline.getVideoThumbURL(videoArgs))
```

Image overlay

Several images can be overlaid to achieve a desired style. Suppose you want to emphasize a region of interest. You can create a muted copy of an image visualization as a base layer and then overlay a clipped version of the original visualization. Building on the previous example, the following script produces

```
# Define an empty image to paint features to
empty = ee.Image().byte()

# Paint country feature edges and visualize as black lines
southAmOutline = (
    empty
    .paint(
      featureCollection=southAmCol,
      color=1,
      width=1
```

```
    )
    .visualize({'palette': '000000'})
)

# Overlay country outlines on temperature visualization images
tempColOutline = tempColVis.map(
    lambda img: img.blend(southAmOutline)
)

# Define a partially opaque grey dulling layer
dullLayer = ee.Image.constant(175).visualize(
    {
        'opacity': 0.6,
        'min': 0,
        'max': 255,
        'forceRgbOutput': True
    }
)

# Apply two-step blending for final styling
finalVisCol = tempColOutline.map(
    lambda img: (
        img
        .blend(dullLayer)
        .blend(img.clipToCollection(southAmCol))
    )
)

# Animation arguments
videoArgs = {
    'dimensions': 768,
    'region': southAmAoi,
    'framesPerSecond': 7,
    'crs': 'EPSG:3857'
}

# Generate and print animation URL
print(finalVisCol.getVideoThumbURL(videoArgs))
```

Transitions

Customize an image collection to produce animations that reveal differences between two images within a collection using fade, flicker, and slide transitions. Each of the following examples use the same base visualization generated by the following script:

```
# Define area of interest (global non-polar extent)
aoi = ee.Geometry.Polygon(
    [[
        [-179.0, 78.0],
        [-179.0, -58.0],
        [179.0, -58.0],
        [179.0, 78.0]
    ]],
    None,
    False
)

# Import hourly predicted temperature image collection
temp = ee.ImageCollection('NOAA/GFS0P25')

# Northern summer solstice temperature image
summerSolTemp = (
    temp
    .filterDate('2018-06-21', '2018-06-22')
    .filterMetadata('forecast_hours', 'equals', 12)
    .first()
    .select('temperature_2m_above_ground')
)

# Northern winter solstice temperature image
winterSolTemp = (
    temp
    .filterDate('2018-12-22', '2018-12-23')
    .filterMetadata('forecast_hours', 'equals', 12)
    .first()
    .select('temperature_2m_above_ground')
)

# Combine solstice images into a collection
tempCol = ee.ImageCollection([
    summerSolTemp.set('season', 'summer'),
    winterSolTemp.set('season', 'winter')
```

```
])

# Import international boundaries
countries = ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017')

# Visualization arguments
visArgs = {
  'min': -40.0,
  'max': 35.0,
  'palette': ['blue', 'purple', 'cyan', 'green', 'yellow', 'red']
}

# Convert images to RGB visualization, clip to countries, mask oceans
tempColVis = tempCol.map(
  lambda img: (
    img
    .visualize(visArgs)
    .clipToCollection(countries)
    .unmask(0)
    .copyProperties(img, img.propertyNames())
  )
)
```

ImageCollection Information and Metadata

As with Images, there are a variety of ways to get information about an `ImageCollection`. The collection can be printed directly to the console, but the console printout is limited to 5000 elements. Collections larger than 5000 images will need to be filtered before printing. Printing a large collection will be correspondingly slower. The following example shows various ways of getting information about image collections programmatically:

```
# Load a Landsat 8 ImageCollection for a single path-row.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
```

```
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
    .filterDate('2014-03-01', '2014-08-01')
)
display('Collection:', collection)

# Get the number of images.
count = collection.size()
display('Count:', count)

# Get the date range of images in the collection.
range = collection.reduceColumns(ee.Reducer.minMax(),
['system:time_start'])
display('Date range:', ee.Date(range.get('min')),
ee.Date(range.get('max')))

# Get statistics for a property of the images in the collection.
sun_stats = collection.aggregate_stats('SUN_ELEVATION')
display('Sun elevation statistics:', sun_stats)

# Sort by a cloud cover property, get the least cloudy image.
image = ee.Image(collection.sort('CLOUD_COVER').first())
display('Least cloudy image:', image)

# Limit the collection to the 10 most recent images.
recent = collection.sort('system:time_start', False).limit(10)
display('Recent images:', recent)
```

Filtering an ImageCollection

As illustrated in the Get Started section and the ImageCollection Information section, Earth Engine provides a variety of convenience methods for filtering image collections. Specifically, many common use cases are handled by `imageCollection.filterDate()`, and `imageCollection.filterBounds()`. For general purpose filtering, use `imageCollection.filter()` with an `ee.Filter` as an argument. The following example demonstrates both convenience methods and `filter()` to identify and remove images with high cloud cover from an `ImageCollection`.

```
# Load Landsat 8 data, filter by date, month, and bounds.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    # Three years of data
    .filterDate('2015-01-01', '2018-01-01')
    # Only Nov-Feb observations
    .filter(ee.Filter.calendarRange(11, 2, 'month'))
    # Intersecting ROI
    .filterBounds(ee.Geometry.Point(25.8544, -18.08874))
)

# Also filter the collection by the CLOUD_COVER property.
filtered = collection.filter(ee.Filter.eq('CLOUD_COVER', 0))

# Create two composites to check the effect of filtering by
CLOUD_COVER.
bad_composite = collection.mean()
good_composite = filtered.mean()

# Display the composites.
m = geemap.Map()
m.set_center(25.8544, -18.08874, 13)
m.add_layer(
    bad_composite,
    {'bands': ['B3', 'B2', 'B1'], 'min': 0.05, 'max': 0.35, 'gamma':
1.1},
    'Bad composite',
)
m.add_layer(
    good_composite,
    {'bands': ['B3', 'B2', 'B1'], 'min': 0.05, 'max': 0.35, 'gamma':
1.1},
    'Good composite',
)
m
```

Mapping over an ImageCollection

To apply a function to every `Image` in an `ImageCollection` use `imageCollection.map()`. The only argument to `map()` is a function which takes one parameter: an `ee.Image`. For example, the following code adds a timestamp band to every image in the collection.

```
# Load a Landsat 8 collection for a single path-row, 2021 images only.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterDate('2021', '2022')
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
)


# This function adds a band representing the image timestamp.
def add_time(image):
  return image.addBands(image.getNumber('system:time_start'))


# Map the function over the collection and display the result.
display(collection.map(add_time))
```

Note that in the predefined function, the `getNumber()` method is used to create a new `Image` from the numerical value of a property. As discussed in the Reducing and Compositing sections, having the time band is useful for linear modeling of change and for making composites.

The mapped function is limited in the operations it can perform. Specifically, it can't modify variables outside the function; it can't print anything; it can't use JavaScript and Python 'if' or 'for' statements. However, you can use `ee.Algorithms.If()` to perform conditional operations in a mapped function. For example:

```
# Load a Landsat 8 collection for a single path-row, 2021 images only.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterDate('2021', '2022')
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
)
```

```
# This function uses a conditional statement to return the image if
# the solar elevation > 40 degrees. Otherwise it returns a "zero
image".
def conditional(image):
  return ee.Algorithms.If(
      ee.Number(image.get('SUN_ELEVATION')).gt(40), image, ee.Image(0)
  )


# Map the function over the collection and print the result. Expand the
# collection and note that 7 of the 22 images are now "zero images'.
display('Expand this to see the result', collection.map(conditional))
```

Reducing an ImageCollection

To composite images in an `ImageCollection`, use `imageCollection.reduce()`. This will composite all the images in the collection to a single image representing, for example, the min, max, mean or standard deviation of the images. (See the Reducers section for more information about reducers). For example, to create a median value image from a collection:

```
# Load a Landsat 8 collection for a single path-row.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
    .filterDate('2014-01-01', '2015-01-01')
)

# Compute a median image and display.
median = collection.median()
m = geemap.Map()
m.set_center(-122.3578, 37.7726, 12)
m.add_layer(median, {'bands': ['B4', 'B3', 'B2'], 'max': 0.3},
'Median')
```

```
m
```

At each location in the output image, in each band, the pixel value is the median of all unmasked pixels in the input imagery (the images in the collection). In the previous example, `median()` is a convenience method for the following call:

```
# Reduce the collection with a median reducer.
median = collection.reduce(ee.Reducer.median())

# Display the median image.
m.add_layer(
    median,
    {'bands': ['B4_median', 'B3_median', 'B2_median'], 'max': 0.3},
    'Also median',
)
m
```

Note that the band names differ as a result of using `reduce()` instead of the convenience method. Specifically, the names of the reducer have been appended to the band names.

More complex reductions are also possible using `reduce()`. For example, to compute the long term linear trend over a collection, use one of the linear regression reducers. The following code computes the linear trend of MODIS Enhanced Vegetation Index (EVI):

```
# This function adds a band representing the image timestamp.
def add_time(image):
  return image.addBands(
      image.metadata('system:time_start')
      # Convert milliseconds from epoch to years to aid in
      # interpretation of the following trend calculation.
      .divide(1000 * 60 * 60 * 24 * 365)
  )


# Load a MODIS collection, filter to several years of 16 day mosaics,
# and map the time band function over it.
collection = (
    ee.ImageCollection('MODIS/006/MYD13A1')
    .filterDate('2004-01-01', '2010-10-31')
    .map(add_time)
```

```
)

# Select the bands to model with the independent variable first.
trend = collection.select(['system:time_start', 'EVI']).reduce(
    # Compute the linear trend over time.
    ee.Reducer.linearFit()
)

# Display the trend with increasing slopes in green, decreasing in red.
m.set_center(-96.943, 39.436, 5)
m = geemap.Map()
m.add_layer(
    trend,
    {
        'min': 0,
        'max': [-100, 100, 10000],
        'bands': ['scale', 'scale', 'offset'],
    },
    'EVI trend',
)
m
```

Note that the output of the reduction in this example is a two banded image with one band for the slope of a linear regression (`scale`) and one band for the intercept (`offset`). Explore the API documentation to see a list of the reducers that are available to reduce an `ImageCollection` to a single `Image`.

Compositing and Mosaicking

In general, compositing refers to the process of combining spatially overlapping images into a single image based on an aggregation function. Mosaicking refers to the process of spatially assembling image datasets to produce a spatially continuous image. In Earth Engine, these terms are used interchangeably, though both compositing and mosaicking are supported. For example, consider the task of compositing multiple images in the same location. For example, using one National Agriculture Imagery Program (NAIP) Digital Orthophoto Quarter Quadrangle (DOQQ) at different times, the following example demonstrates making a maximum value composite:

```
# Load three NAIP quarter quads in the same location, different times.
```

```
naip_2004_2012 = (
    ee.ImageCollection('USDA/NAIP/DOQQ')
    .filterBounds(ee.Geometry.Point(-71.08841, 42.39823))
    .filterDate('2004-07-01', '2012-12-31')
    .select(['R', 'G', 'B'])
)

# Temporally composite the images with a maximum value function.
composite = naip_2004_2012.max()
m.set_center(-71.12532, 42.3712, 12)
m.add_layer(composite, {}, 'max value composite')
m
```

Consider the need to mosaic four different DOQQs at the same time, but different locations. The following example demonstrates that using `imageCollection.mosaic()`:

```
# Load four 2012 NAIP quarter quads, different locations.
naip_2012 = (
    ee.ImageCollection('USDA/NAIP/DOQQ')
    .filterBounds(
        ee.Geometry.Rectangle(-71.17965, 42.35125, -71.08824, 42.40584)
    )
    .filterDate('2012-01-01', '2012-12-31')
)

# Spatially mosaic the images in the collection and display.
mosaic = naip_2012.mosaic()
m = geemap.Map()
m.set_center(-71.12532, 42.3712, 12)
m.add_layer(mosaic, {}, 'spatial mosaic')
```

Note that there is some overlap in the DOQQs in the previous example. The `mosaic()` method composites overlapping images according to their order in the collection (last on top). To control the source of pixels in a mosaic (or a composite), use image masks. For example, the following uses thresholds on spectral indices to mask the image data in a mosaic:

```
# Load a NAIP quarter quad, display.
naip = ee.Image('USDA/NAIP/DOQQ/m_4207148_nw_19_1_20120710')
```

```
m = geemap.Map()
m.set_center(-71.0915, 42.3443, 14)
m.add_layer(naip, {}, 'NAIP DOQQ')

# Create the NDVI and NDWI spectral indices.
ndvi = naip.normalizedDifference(['N', 'R'])
ndwi = naip.normalizedDifference(['G', 'N'])

# Create some binary images from thresholds on the indices.
# This threshold is designed to detect bare land.
bare_1 = ndvi.lt(0.2).And(ndwi.lt(0.3))
# This detects bare land with lower sensitivity. It also detects
shadows.
bare_2 = ndvi.lt(0.2).And(ndwi.lt(0.8))

# Mask and mosaic visualization images. The last layer is on top.
mosaic = ee.ImageCollection([
    # NDWI > 0.5 is water. Visualize it with a blue palette.
    ndwi.updateMask(ndwi.gte(0.5)).visualize(
        min=0.5, max=1, palette=['00FFFF', '0000FF']
    ),
    # NDVI > 0.2 is vegetation. Visualize it with a green palette.
    ndvi.updateMask(ndvi.gte(0.2)).visualize(
        min=-1, max=1, palette=['FF0000', '00FF00']
    ),
    # Visualize bare areas with shadow (bare_2 but not bare_1) as gray.

bare_2.updateMask(bare_2.And(bare_1.Not())).visualize(palette=['AAAAAA'
]),
    # Visualize the other bare areas as white.
    bare_1.updateMask(bare_1).visualize(palette=['FFFFFF']),
]).mosaic()
m.add_layer(mosaic, {}, 'Visualization mosaic')
```

To make a composite which maximizes an arbitrary band in the input, use `imageCollection.qualityMosaic()`. The `qualityMosaic()` method sets each pixel in the composite based on which image in the collection has a maximum value for the specified band. For example, the following code demonstrates making a greenest pixel composite and a recent value composite:

```
# Define a function that scales and masks Landsat 8 surface reflectance
images.
```

```
def prep_sr_l8(image):
  # Develop masks for unwanted pixels (fill, cloud, cloud shadow).
  qa_mask = image.select('QA_PIXEL').bitwiseAnd(int('11111', 2)).eq(0)
  saturation_mask = image.select('QA_RADSAT').eq(0)

  # Helper function to create image from scaling factors.
  def get_factor_img(factor_names):
    factor_list = image.toDictionary().select(factor_names).values()
    return ee.Image.constant(factor_list)

  # Apply the scaling factors to the appropriate bands.
  scale_img = get_factor_img(
      ['REFLECTANCE_MULT_BAND_.|TEMPERATURE_MULT_BAND_ST_B10']
  )
  offset_img = get_factor_img(
      ['REFLECTANCE_ADD_BAND_.|TEMPERATURE_ADD_BAND_ST_B10']
  )
  scaled =
image.select('SR_B.|ST_B10').multiply(scale_img).add(offset_img)

  # Replace original bands with scaled bands and apply masks.
  return (
      image.addBands(scaled, None, True)
      .updateMask(qa_mask)
      .updateMask(saturation_mask)
  )


# This function masks clouds and adds quality bands to Landsat 8
images.
def add_quality_bands(image):
  # Normalized difference vegetation index.
  ndvi = image.normalizedDifference(['SR_B5', 'SR_B4'])
  # Image timestamp as milliseconds since Unix epoch.
  millis = (

ee.Image(image.getNumber('system:time_start')).rename('millis').toFloat
()
  )
  return prep_sr_l8(image).addBands([ndvi, millis])


# Load a 2014 Landsat 8 ImageCollection.
```

```
# Map the cloud masking and quality band function over the collection.
collection = (
    ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterDate('2014-06-01', '2014-12-31')
    .map(add_quality_bands)
)

# Create a cloud-free, most recent value composite.
recent_value_composite = collection.qualityMosaic('millis')

# Create a greenest pixel composite.
greenest_pixel_composite = collection.qualityMosaic('nd')

# Display the results.
m = geemap.Map()
m.set_center(-122.374, 37.8239, 12)  # San Francisco Bay
viz_params = {'bands': ['SR_B5', 'SR_B4', 'SR_B3'], 'min': 0, 'max':
0.4}
m.add_layer(recent_value_composite, viz_params, 'Recent value
composite')
m.add_layer(greenest_pixel_composite, viz_params, 'Greenest pixel
composite')

# Compare to a cloudy image in the collection.
cloudy = ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140825')
m.add_layer(
    cloudy, {'bands': ['B5', 'B4', 'B3'], 'min': 0, 'max': 0.4},
'Cloudy'
)
m
```

Iterating over an ImageCollection

Although `map()` applies a function to every image in a collection, the function visits every image in the collection independently. For example, suppose you want to compute a cumulative anomaly (A_t) at time t from a time series. To obtain a recursively defined series of the form $A_t = f(Image_t, A_{t-1})$, mapping won't work because the function (f) depends on the previous result (A_{t-1}). For example, suppose you want to compute a series of cumulative Normalized Difference Vegetation Index (NDVI) anomaly images relative to a baseline. Let $A_0 = 0$ and $f(Image_t, A_{t-1}) = Image_t + A_{t-1}$ where A_{t-1} is the cumulative anomaly up to time $t-1$ and $Image_t$ is the anomaly at time t. Use `imageCollection.iterate()` to make this recursively defined

`ImageCollection`. In the following example, the function `accumulate()` takes two parameters: an image in the collection, and a list of all the previous outputs. With each call to `iterate()`, the anomaly is added to the running sum and the result is added to the list. The final result is passed to the `ImageCollection` constructor to get a new sequence of images:

```
import altair as alt
# Load MODIS EVI imagery.
collection = ee.ImageCollection('MODIS/006/MYD13A1').select('EVI')

# Define reference conditions from the first 10 years of data.
reference = collection.filterDate('2001-01-01', '2010-12-31').sort(
    # Sort chronologically in descending order.
    'system:time_start',
    False,
)

# Compute the mean of the first 10 years.
mean = reference.mean()

# Compute anomalies by subtracting the 2001-2010 mean from each image
in a
# collection of 2011-2014 images. Copy the date metadata over to the
# computed anomaly images in the new collection.
series = collection.filterDate('2011-01-01', '2014-12-31').map(
    lambda image: image.subtract(mean).set(
        'system:time_start', image.get('system:time_start')
    )
)

# Display cumulative anomalies.
m = geemap.Map()
m.set_center(-100.811, 40.2, 5)
m.add_layer(
    series.sum(),
    {'min': -60000, 'max': 60000, 'palette': ['FF0000', '000000',
'00FF00']},
    'EVI anomaly',
)
display(m)
```

```
# Get the timestamp from the most recent image in the reference
collection.
time_0 = reference.first().get('system:time_start')

# Use imageCollection.iterate() to make a collection of cumulative
anomaly over time.
# The initial value for iterate() is a list of anomaly images already
processed.
# The first anomaly image in the list is just 0, with the time_0
timestamp.
first = ee.List([
    # Rename the first band 'EVI'.
    ee.Image(0)
    .set('system:time_start', time_0)
    .select([0], ['EVI'])
])

# This is a function to pass to Iterate().
# As anomaly images are computed, add them to the list.
def accumulate(image, list):
  # Get the latest cumulative anomaly image from the end of the list
with
  # get(-1).  Since the type of the list argument to the function is
unknown,
  # it needs to be cast to a List.  Since the return type of get() is
unknown,
  # cast it to Image.
  previous = ee.Image(ee.List(list).get(-1))
  # Add the current anomaly to make a new cumulative anomaly image.
  added = image.add(previous).set(
      # Propagate metadata to the new image.
      'system:time_start',
      image.get('system:time_start'),
  )
  # Return the list with the cumulative anomaly inserted.
  return ee.List(list).add(added)

# Create an ImageCollection of cumulative anomaly images by iterating.
# Since the return type of iterate is unknown, it needs to be cast to a
List.
cumulative = ee.ImageCollection(ee.List(series.iterate(accumulate,
first)))
```

```
# Predefine the chart titles.
title = 'Cumulative EVI anomaly over time'

# Chart some interesting locations.
def display_chart(region, collection):
  reduced = (
      collection.filterBounds(region)
      .sort('system:time_start')
      .map(
          lambda image: ee.Feature(
              None,
              image.reduceRegion(ee.Reducer.first(), region, 500).set(
                  'time', image.get('system:time_start')
              ),
          )
      )
  )
  reduced_dataframe = ee.data.computeFeatures(
      {'expression': reduced, 'fileFormat': 'PANDAS_DATAFRAME'}
  )
  alt.Chart(reduced_dataframe).mark_line().encode(
      alt.X('time:T').title('Time'),
      alt.Y('EVI:Q').title('Cumulative EVI anomaly'),
  ).properties(title=title).display()

pt_1 = ee.Geometry.Point(-65.544, -4.894)
display('Amazon rainforest:')
display_chart(pt_1, cumulative)

pt_2 = ee.Geometry.Point(116.4647, 40.1054)
display('Beijing urbanization:')
display_chart(pt_2, cumulative)

pt_3 = ee.Geometry.Point(-110.3412, 34.1982)
display('Arizona forest disturbance and recovery:')
display_chart(pt_3, cumulative)
```

Charting these sequences indicates whether NDVI is stabilizing relative to previous disturbances or whether NDVI is trending to a new state. Learn more about charts in Earth Engine from the Charts section.

The iterated function is limited in the operations it can perform. Specifically, it can’t modify variables outside the function; it can’t print anything; it can’t use JavaScript ‘if’ or ‘for’ statements. Any results you wish to collect or intermediate information you wish to carry over to the next iteration must be in the function’s return value. You can use `ee.Algorithms.If()` to perform conditional operations.

Geometry Overview

Earth Engine handles vector data with the `Geometry` type. The GeoJSON spec describes in detail the type of geometries supported by Earth Engine, including `Point` (a list of coordinates in some projection), `LineString` (a list of points), `LinearRing` (a closed `LineString`), and `Polygon` (a list of `LinearRing`s where the first is a shell and subsequent rings are holes). Earth Engine also supports `MultiPoint`, `MultiLineString`, and `MultiPolygon`. The GeoJSON GeometryCollection is also supported, although it has the name `MultiGeometry` within Earth Engine.

Creating Geometry objects

You can create geometries interactively using the Code Editor geometry tools. See the Earth Engine Code Editor page for more information. To create a `Geometry` programmatically, provide the constructor with the proper list(s) of coordinates. For example:

```
# Point geometry
point = ee.Geometry.Point([1.5, 1.5])

# LineString geometry
lineString = ee.Geometry.LineString(
    [[-35, -10], [35, -10], [35, 10], [-35, 10]]
)

# LinearRing geometry
linearRing = ee.Geometry.LinearRing(
    [[-35, -10], [35, -10], [35, 10], [-35, 10], [-35, -10]]
)

# Rectangle geometry
rectangle = ee.Geometry.Rectangle([-40, -20, 40, 20])

# Polygon geometry
polygon = ee.Geometry.Polygon([
    [[-5, 40], [65, 40], [65, 60], [-5, 60], [-5, 60]]
])
```

In the previous examples, note that the distinction between a `LineString` and a `LinearRing` is that the `LinearRing` is “closed” by having the same coordinate at both the start and end of the list.

An individual `Geometry` may consist of multiple geometries. To break a multi-part `Geometry` into its constituent geometries, use `geometry.geometries()`. For example:

```
# Create a multi-part geometry
multiPoint = ee.Geometry.MultiPoint(
    [[-121.68, 39.91], [-97.38, 40.34]]
)
```

```
# Get the individual geometries as a list
geometries = multiPoint.geometries()

# Get each geometry from the list
pt1 = geometries.get(0)
pt2 = geometries.get(1)

# Print the geometries
print('Point 1', pt1)
print('Point 2', pt2)
```

Geodesic vs. Planar Geometries

A geometry created in Earth Engine is either geodesic (i.e. edges are the shortest path on the surface of a sphere) or planar (i.e. edges are the shortest path in a 2-D Cartesian plane). No one planar coordinate system is suitable for global collections of features, so Earth Engine's geometry constructors build geodesic geometries by default. To make a planar geometry, constructors have a `geodesic` parameter that can be set to **false**:

```
planarPolygon = ee.Geometry(polygon, None, False)
```

Geometry Visualization and Information

Visualizing geometries

To visualize a geometry, add it to the map. For example:

```
# Create a geodesic polygon
polygon = ee.Geometry.Polygon([
    [[-5, 40], [65, 40], [65, 60], [-5, 60], [-5, 60]]
])

# Create a planar polygon
planarPolygon = ee.Geometry(polygon, None, False)

# Display the polygons
Map.centerObject(polygon)
Map.addLayer(polygon, {'color': 'FF0000'}, 'geodesic polygon')
Map.addLayer(planarPolygon, {'color': '000000'}, 'planar polygon')
```

Geometry information and metadata

To view information about a geometry, print it. To access the information programmatically, Earth Engine provides several methods. For example, to get information about the polygon created previously, use:

```
print('Polygon printout:', polygon)
```

```
# Print polygon area in square kilometers
print('Polygon area:', polygon.area().divide(1000 * 1000))

# Print polygon perimeter length in kilometers
print('Polygon perimeter:', polygon.perimeter().divide(1000))

# Print the geometry as a GeoJSON string
print('Polygon GeoJSON:', polygon.toGeoJSONString())

# Print the GeoJSON geometry type
print('Geometry type:', polygon.type())

# Print the coordinates as lists
print('Polygon coordinates:', polygon.coordinates())

# Print whether the geometry is geodesic
print('Geodesic?', polygon.geodesic())
```

Geometric Operations

Earth Engine supports a wide variety of operations on `Geometry` objects. These include operations on individual geometries such as computing a buffer, centroid, bounding box, perimeter, convex hull, etc. For example:

```
# Create a geodesic polygon
polygon = ee.Geometry.Polygon([
```

```
    [[-5, 40], [65, 40], [65, 60], [-5, 60], [-5, 60]]
])

# Compute a buffer of the polygon (1,000,000 meters)
buffer = polygon.buffer(1000000)

# Compute the centroid of the polygon
centroid = polygon.centroid()

# Display results
Map.addLayer(buffer, {}, 'buffer')
Map.addLayer(centroid, {}, 'centroid')
```

Observe from the previous example that the buffer distance is specified in meters.

Supported geometric operations also include relational computations between geometries such as intersection, union, difference, distance, contains, etc. To test some of these relations, geometries use the “even-odd” rule by default. By the even-odd rule, a point is inside the polygon if a line from that point to some point known to be outside the polygon crosses an odd number of other edges. The inside of a polygon is everything inside the shell and not inside a hole. As a simple example, a point within a circular polygon must cross exactly one edge to escape the polygon. Geometries can optionally use the "left-inside" rule, if necessary. Imagine walking the points of a ring in the order given; the inside will be on the left.

To demonstrate the difference between geometries created with the "left-inside" rule (`evenOdd: false`) and those created with the "even-odd" rule, the following example compares a point to two different polygons:

```
# Create a left-inside polygon
holePoly = ee.Geometry.Polygon(
    coords=[
```

```
        [[-35, -10], [-35, 10], [35, 10], [35, -10], [-35, -10]]
    ],
    evenOdd=False
)

# Create an even-odd version of the polygon
evenOddPoly = ee.Geometry(
    geoJson=holePoly,
    evenOdd=True
)

# Create a point to test insideness
pt = ee.Geometry.Point([1.5, 1.5])

# Check insideness
print(holePoly.contains(pt))     # false
print(evenOddPoly.contains(pt))  # true
```

The previous example demonstrates how the order of coordinates provided to the `Polygon` constructor affects the result when a left-inside polygon is constructed. Specifically, the point is outside the left-inside polygon but inside the even-odd polygon.

The following example computes and visualizes derived geometries based on the relationship between two polygons:

```
# Create two circular geometries
poly1 = ee.Geometry.Point([-50, 30]).buffer(1e6)
```

```
poly2 = ee.Geometry.Point([-40, 30]).buffer(1e6)

# Display polygon 1 (red) and polygon 2 (blue)
Map.setCenter(-45, 30)
Map.addLayer(poly1, {'color': 'FF0000'}, 'poly1')
Map.addLayer(poly2, {'color': '0000FF'}, 'poly2')

# Compute the intersection (green)
intersection = poly1.intersection(poly2, ee.ErrorMargin(1))
Map.addLayer(intersection, {'color': '00FF00'}, 'intersection')

# Compute the union (magenta)
union = poly1.union(poly2, ee.ErrorMargin(1))
Map.addLayer(union, {'color': 'FF00FF'}, 'union')

# Compute the difference (yellow)
diff1 = poly1.difference(poly2, ee.ErrorMargin(1))
Map.addLayer(diff1, {'color': 'FFFF00'}, 'diff1')

# Compute symmetric difference (black)
symDiff = poly1.symmetricDifference(poly2, ee.ErrorMargin(1))
Map.addLayer(symDiff, {'color': '000000'}, 'symmetric difference')
```

In these examples, note that that `maxError` parameter is set to one meter for the geometry operations. The `maxError` is the maximum allowable error, in meters, from

transformations (such as projection or reprojection) that may alter the geometry. If one of the geometries is in a different projection from the other, Earth Engine will do the computation in a spherical coordinate system, with a projection precision given by `maxError`. You can also specify a specific projection in which to do the computation, if necessary.

Feature Overview

A `Feature` in Earth Engine is defined as a GeoJSON Feature. Specifically, a `Feature` is an object with a `geometry` property storing a `Geometry` object (or null) and a `properties` property storing a dictionary of other properties.

Creating Feature objects

To create a `Feature`, provide the constructor with a `Geometry` and (optionally) a dictionary of other properties. For example:

```
# Create an ee.Geometry.

polygon = ee.Geometry.Polygon(

    [[[-35, -10], [35, -10], [35, 10], [-35, 10], [-35, -10]]]

)



# Create a Feature from the Geometry.

poly_feature = ee.Feature(polygon, {'foo': 42, 'bar': 'tart'})
```

As with a `Geometry`, a `Feature` may be printed or added to the map for inspection and visualization:

```
display(poly_feature)

m = geemap.Map()

m.add_layer(poly_feature, {}, 'feature')

display(m)
```

A `Feature` need not have a `Geometry` and may simply wrap a dictionary of properties. For example:

```
# Create a dictionary of properties, some of which may be computed values.

dic = {'foo': ee.Number(8).add(88), 'bar': 'nihao'}


# Create a null geometry feature with the dictionary of properties.

nowhere_feature = ee.Feature(None, dic)
```

In this example, note that the dictionary supplied to the `Feature` contains a computed value. Creating features in this manner is useful for exporting long-running computations with a `Dictionary` result (e.g. `image.reduceRegion()`). See the FeatureCollections and Importing Table Data or Exporting guides for details.

Each `Feature` has one primary `Geometry` stored in the `geometry` property. Additional geometries may be stored in other properties. `Geometry` methods such as intersection and buffer also exist on `Feature` as a convenience for getting the primary `Geometry`, applying the operation, and setting the result as the new primary `Geometry`. The result will retain all the other properties of the `Feature` on which the method is called. There are also methods for getting and setting the non-geometry properties of the `Feature`. For example:

```
# Make a feature and set some properties.

feature = (

    ee.Feature(ee.Geometry.Point([-122.22599, 37.17605]))

    .set('genus', 'Sequoia')

    .set('species', 'sempervirens')

)


# Get a property from the feature.

species = feature.get('species')

display(species)
```

```
# Set a new property.

feature = feature.set('presence', 1)


# Overwrite the old properties with a new dictionary.

new_dic = {'genus': 'Brachyramphus', 'species': 'marmoratus'}

feature = feature.set(new_dic)


# Check the result.

display(feature)
```

FeatureCollection Overview

Groups of related features can be combined into a `FeatureCollection`, to enable additional operations on the entire set such as filtering, sorting and rendering. Besides just simple features (geometry + properties), feature collections can also contain other collections.

The `FeatureCollection` constructor

One way to create a `FeatureCollection` is to provide the constructor with a list of features. The features don't need to have the same geometry type or the same properties. For example:

```
# Make a list of Features.

features = [

    ee.Feature(

        ee.Geometry.Rectangle(30.01, 59.80, 30.59, 60.15), {'name':
'Voronoi'}

    ),
```

```
    ee.Feature(ee.Geometry.Point(-73.96, 40.781), {'name':
'Thiessen'}),

    ee.Feature(ee.Geometry.Point(6.4806, 50.8012), {'name':
'Dirichlet'}),

]


# Create a FeatureCollection from the list and print it.

from_list = ee.FeatureCollection(features)

display(from_list)
```

Individual geometries can also be turned into a `FeatureCollection` of just one `Feature`:

Individual geometries can also be turned into a `FeatureCollection` of just one `Feature`:

```
# Create a FeatureCollection from a single geometry and print it.

from_geom = ee.FeatureCollection(ee.Geometry.Point(16.37, 48.225))

display(from_geom)
```

Table Datasets

Earth Engine hosts a variety of table datasets. To load a table dataset, provide the table ID to the `FeatureCollection` constructor. For example, to load RESOLVE Ecoregions data:

```
fc = ee.FeatureCollection('RESOLVE/ECOREGIONS/2017')

m = geemap.Map()

m.set_center(12.17, 20.96, 3)

m.add_layer(fc, {}, 'ecoregions')

display(m)
```

Random Samples

To get a collection of random points in a specified region, you can use:

```
# Define an arbitrary region in which to compute random points.

region = ee.Geometry.Rectangle(-119.224, 34.669, -99.536, 50.064)


# Create 1000 random points in the region.

random_points = ee.FeatureCollection.randomPoints(region)


# Display the points.

m = geemap.Map()
```

```
m.center_object(random_points)

m.add_layer(random_points, {}, 'random points')

display(m)
```

Feature and FeatureCollection Visualization

As with images, geometries and features, feature collections can be added to the map directly with `Map.addLayer()`. The default visualization will display the vectors with solid black lines and semi-opaque black fill. To render the vectors in color, specify the `color` parameter. The following displays the 'RESOLVE' ecoregions (Dinerstein et al. 2017) as the default visualization and in red:

```
# Load a FeatureCollection from a table dataset: 'RESOLVE' ecoregions.

ecoregions = ee.FeatureCollection('RESOLVE/ECOREGIONS/2017')


# Display as default and with a custom color.

m = geemap.Map()

m.set_center(-76.2486, 44.8988, 8)

m.add_layer(ecoregions, {}, 'default display')

m.add_layer(ecoregions, {'color': 'FF0000'}, 'colored')

m
```

For additional display options, use `featureCollection.draw()`. Specifically, parameters `pointRadius` and `strokeWidth` control the size of points and lines, respectively, in the rendered `FeatureCollection`:

```
m.add_layer(ecoregions.draw(color='006600', strokeWidth=5), {},
'drawn')
```

The output of `draw()` is an image with red, green and blue bands set according to the specified `color` parameter.

For more control over how a `FeatureCollection` is displayed, use `image.paint()` with the `FeatureCollection` as an argument. Unlike `draw()`, which outputs a three-band, 8-bit display image, `image.paint()` outputs an image with the specified numeric value 'painted' into it. Alternatively, you can supply the name of a property in the `FeatureCollection` which contains the numbers to paint. The `width` parameter behaves the same way: it can be a constant or the name of a property with a number for the line width. For example:

```
# Create an empty image into which to paint the features, cast to byte.

empty = ee.Image().byte()



# Paint all the polygon edges with the same number and width, display.

outline = empty.paint(featureCollection=ecoregions, color=1, width=3)

m.add_layer(outline, {'palette': 'FF0000'}, 'edges')
```

Note that the empty image into which you paint the features needs to be cast prior to painting. This is because a constant image behaves as a constant: it is clamped to the initialization value. To color the feature edges with values set from a property of the features, set the color parameter to the name of the property with numeric values:

```
# Paint the edges with different colors, display.

outlines = empty.paint(featureCollection=ecoregions, color='BIOME_NUM',
width=4)

palette = ['FF0000', '00FF00', '0000FF']

m.add_layer(outlines, {'palette': palette, 'max': 14}, 'different color
edges')
```

Both the color and width with which the boundaries are drawn can be set with properties. For example:

```
# Paint the edges with different colors and widths.

outlines = empty.paint(

    featureCollection=ecoregions, color='BIOME_NUM', width='NNH'

)

m.add_layer(

    outlines, {'palette': palette, 'max': 14}, 'different color, width
edges'

)
```

If the `width` parameter is not provided, the interior of the features is painted:

```
# Paint the interior of the polygons with different colors.

fills = empty.paint(featureCollection=ecoregions, color='BIOME_NUM')

m.add_layer(fills, {'palette': palette, 'max': 14}, 'colored fills')
```

To render both the interior and edges of the features, paint the empty image twice:

```
# Paint both the fill and the edges.

filled_outlines = empty.paint(ecoregions,
'BIOME_NUM').paint(ecoregions, 0, 2)

m.add_layer(

    filled_outlines,
```

```
    {'palette': ['000000'] + palette, 'max': 14},
    'edges and fills',
)
```

FeatureCollection Information and Metadata

Methods for getting information from feature collection metadata are the same as those for image collections.

Metadata aggregation

You can use the aggregation shortcuts to count the number of features or summarize an attribute:

```
# Load watersheds from a data table.
sheds = (
    ee.FeatureCollection('USGS/WBD/2017/HUC06')
    # Filter to the continental US.
    .filterBounds(ee.Geometry.Rectangle(-127.18, 19.39, -62.75, 51.29))
    # Convert 'areasqkm' property from string to number.
    .map(
        lambda feature: feature.set(
            'areasqkm', ee.Number.parse(feature.get('areasqkm'))
        )
    )
```

```
)


# Display the table and print its first element.
m = geemap.Map()
m.add_layer(sheds, {}, 'watersheds')
display(m)
display('First watershed:', sheds.first())


# Print the number of watersheds.
display('Count:', sheds.size())


# Print stats for an area property.
display('Area stats:', sheds.aggregate_stats('areasqkm'))
```

Column information

Knowing the names and datatypes of `FeatureCollection` columns can be helpful (e.g., filtering a collection by metadata). The following example prints column names and datatypes for a collection of point features representing protected areas.

```
# Import a protected areas point feature collection.
wdpa = ee.FeatureCollection('WCMC/WDPA/current/points')


# Fetch collection metadata (`.limit(0)`). The printed object is a
# dictionary where keys are column names and values are datatypes.
wdpa.limit(0).getInfo()['columns']
```

Filtering a FeatureCollection

Filtering a `FeatureCollection` is analogous to filtering an `ImageCollection`. (See the Filtering an ImageCollection section). There are the `featureCollection.filterDate()`, and `featureCollection.filterBounds()` convenience methods and the `featureCollection.filter()` method for use with any applicable `ee.Filter`. For example:

```
# Load watersheds from a data table.
sheds = (
    ee.FeatureCollection('USGS/WBD/2017/HUC06')
    # Convert 'areasqkm' property from string to number.
    .map(
        lambda feature: feature.set(
            'areasqkm', ee.Number.parse(feature.get('areasqkm'))
        )
    )
)

# Define a region roughly covering the continental US.
continental_us = ee.Geometry.Rectangle(-127.18, 19.39, -62.75, 51.29)

# Filter the table geographically: only watersheds in the continental
US.
filtered = sheds.filterBounds(continental_us)

# Check the number of watersheds after filtering for location.
display('Count after filter:', filtered.size())

# Filter to get only larger continental US watersheds.
```

```
large_sheds = filtered.filter(ee.Filter.gt('areasqkm', 25000))


# Check the number of watersheds after filtering for size and location.

display('Count after filtering by size:', large_sheds.size())
```

Mapping over a FeatureCollection

To apply the same operation to every `Feature` in a `FeatureCollection`, use `featureCollection.map()`. For example, to add another area attribute to every feature in a watersheds `FeatureCollection`, use:

```
# Load watersheds from a data table.

sheds = ee.FeatureCollection('USGS/WBD/2017/HUC06')


# Map an area calculation function over the FeatureCollection.

area_added = sheds.map(

    lambda feature: feature.set(

        {'areaHa': feature.geometry().area().divide(100 * 100)}

    )

)


# Print the first feature from the collection with the added property.

display('First feature:', area_added.first())
```

In the previous example, note that a new property is set based on a computation with the feature's geometry. Properties can also be set using a computation involving existing properties.

An entirely new `FeatureCollection` can be generated with `map()`. The following example converts the watersheds to centroids:

```
# This function creates a new feature from the centroid of the
geometry.

def get_centroid(feature):

  # Keep this list of properties.

  keep_properties = ['name', 'huc6', 'tnmid', 'areasqkm']

  # Get the centroid of the feature's geometry.

  centroid = feature.geometry().centroid()

  # Return a new Feature, copying properties from the old Feature.

  return ee.Feature(centroid).copyProperties(feature, keep_properties)


# Map the centroid getting function over the features.

centroids = sheds.map(get_centroid)


# Display the results.

m = geemap.Map()

m.set_center(-96.25, 40, 4)

m.add_layer(centroids, {'color': 'FF0000'}, 'centroids')

m
```

Reducing a FeatureCollection

To aggregate data in the properties of a `FeatureCollection`, use `featureCollection.reduceColumns()`. For example, to check the area properties in the watersheds `FeatureCollection`, this code computes the Root Mean Square Error (RMSE) relative to the Earth Engine computed area:

```
# Load watersheds from a data table and filter to the continental US.

sheds = ee.FeatureCollection('USGS/WBD/2017/HUC06').filterBounds(

    ee.Geometry.Rectangle(-127.18, 19.39, -62.75, 51.29)
```

```
)


# This function computes the squared difference between an area
property

# and area computed directly from the feature's geometry.

def area_diff(feature):

  # Compute area in sq. km directly from the geometry.

  area = feature.geometry().area().divide(1000 * 1000)

  # Compute the difference between computed area and the area property.

  diff = area.subtract(ee.Number.parse(feature.get('areasqkm')))

  # Return the feature with the squared difference set to the 'diff'
property.

  return feature.set('diff', diff.pow(2))


# Calculate RMSE for population of difference pairs.

rmse = (

    ee.Number(

        # Map the difference function over the collection.

        sheds.map(area_diff)

        # Reduce to get the mean squared difference.

        .reduceColumns(ee.Reducer.mean(), ['diff']).get('mean')

    )

    # Compute the square root of the mean square to get RMSE.

    .sqrt()

)
```

```
# Print the result.

display('RMSE=', rmse)
```

In this example, note that the return value of `reduceColumns()` is a dictionary with key `'mean'`. To get the mean, cast the result of `dictionary.get()` to a number with `ee.Number()` before trying to call `sqrt()` on it. For more information about ancillary data structures in Earth Engine, see this tutorial.

To overlay features on imagery, use `featureCollection.reduceRegions()`. For example, to compute the volume of precipitation in continental US watersheds, use `reduceRegions()` followed by a `map()`:

```
# Load an image of daily precipitation in mm/day.

precip = ee.Image(ee.ImageCollection('NASA/ORNL/DAYMET_V3').first())


# Load watersheds from a data table and filter to the continental US.

sheds = ee.FeatureCollection('USGS/WBD/2017/HUC06').filterBounds(

    ee.Geometry.Rectangle(-127.18, 19.39, -62.75, 51.29)

)


# Add the mean of each image as new properties of each feature.

with_precip = precip.reduceRegions(sheds, ee.Reducer.mean()).filter(

    ee.Filter.notNull(['prcp'])

)



# This function computes total rainfall in cubic meters.

def prcp_volume(feature):

  # Precipitation in mm/day -> meters -> sq. meters.
```

```
  volume = (
      ee.Number(feature.get('prcp'))
      .divide(1000)
      .multiply(feature.geometry().area())
  )
  return feature.set('volume', volume)

high_volume = (
    # Map the function over the collection.
    with_precip.map(prcp_volume)
    # Sort descending and get only the 5 highest volume watersheds.
    .sort('volume', False).limit(5)
    # Extract the names to a list.
    .reduceColumns(ee.Reducer.toList(), ['name']).get('list')
)

# Print the resulting FeatureCollection.
display(high_volume)
```

Vector to Raster Interpolation

Interpolation from vector to raster in Earth Engine creates an `Image` from a `FeatureCollection`. Specifically, Earth Engine uses numeric data stored in a property of the features to interpolate values at new locations outside of the features. The interpolation results in a continuous `Image` of interpolated values up to the distance specified.

Inverse Distance Weighted Interpolation

The inverse distance weighting (IDW) function in Earth Engine is based on the method described by Basso et al. (1999). An additional control parameter is added in the form of a decay factor (`gamma`) on the inverse distance. Other parameters include the mean and standard deviation of the property to interpolate and the maximum range distance over which to interpolate. The following example creates an interpolated surface of methane concentration to fill spatial gaps in the original raster dataset. The `FeatureCollection` is generated by sampling a two-week methane composite.

```
# Import two weeks of S5P methane and composite by mean

ch4 = (

   ee.ImageCollection('COPERNICUS/S5P/OFFL/L3_CH4')

   .select('CH4_column_volume_mixing_ratio_dry_air')

   .filterDate('2019-08-01', '2019-08-15')

   .mean()

   .rename('ch4')

)


# Define area of interest

aoi = ee.Geometry.Polygon(

   [[

      [-95.68487605978851, 43.09844605027055],

      [-95.68487605978851, 37.39358590079781],

      [-87.96148738791351, 37.39358590079781],

      [-87.96148738791351, 43.09844605027055]

   ]],

   None,

   False
```

```
)

# Sample methane composite
samples = (
    ch4
    .addBands(ee.Image.pixelLonLat())
    .sample(
        region=aoi,
        numPixels=1500,
        scale=1000,
        projection='EPSG:4326'
    )
    .map(
        lambda sample: ee.Feature(
            ee.Geometry.Point([
                sample.get('longitude'),
                sample.get('latitude')
            ]),
            {'ch4': sample.get('ch4')}
        )
    )
)

# Combine mean and standard deviation reducers
```

```
combinedReducer = ee.Reducer.mean().combine(
    reducer2=ee.Reducer.stdDev(),
    sharedInputs=True
)

# Estimate global mean and standard deviation
stats = samples.reduceColumns(
    reducer=combinedReducer,
    selectors=['ch4']
)

# Inverse Distance Weighted interpolation (valid to 70 km)
interpolated = samples.inverseDistance(
    range=7e4,
    propertyName='ch4',
    mean=stats.get('mean'),
    stdDev=stats.get('stdDev'),
    gamma=0.3
)

# Visualization arguments
band_viz = {
    'min': 1800,
    'max': 1900,
```

```
    'palette': [
      '0D0887', '5B02A3', '9A179B', 'CB4678',
      'EB7852', 'FBB32F', 'F0F921'
    ]
}

# Display results
Map.centerObject(aoi, 7)
Map.addLayer(ch4, band_viz, 'CH4')
Map.addLayer(interpolated, band_viz, 'CH4 Interpolated')
```

Kriging

Kriging is an interpolation method that uses a modeled estimate of semi-variance to create an image of interpolated values that is an optimal combination of the values at known locations. The Kriging estimator requires parameters that describe the shape of a semi-variogram fit to the known data points.

A **variogram** is the graphical representation of the spatial dependence between pairs of data points, commonly used in geostatistics and spatial statistics. The term is sometimes used synonymously with **semivariogram**, but the latter is also used by some authors to refer to half of a variogram, and should therefore be avoided.[1] Likewise, the term *semivariance* can be misleading, since the values shown in a variogram are entire variances of observations at a given spatial separation (lag).[1]

The variogram is the key function in geostatistics as it will be used to fit a model of the temporal/spatial correlation of the observed phenomenon. One is thus making a distinction between the *experimental variogram* that is a visualization of a possible spatial/temporal correlation and the *variogram model* that is further used to define the weights of the kriging function. Note that the experimental variogram is an

empirical estimate of the covariance of a Gaussian process. As such, it may not be positive definite and hence not directly usable in kriging, without constraints or further processing. This explains why only a limited number of variogram models are used: most commonly, the linear, the spherical, the Gaussian, and the exponential models.

For example, in gold mining, a variogram will give a measure of how much two samples taken from the mining area will vary in gold percentage depending on the distance between those samples. Samples taken far apart will vary more than samples taken close to each other.

Definition

"Semivariance" redirects here. For the measure of downside risk, see Variance § Semivariance.

The semivariogram

$\gamma(h)$

was first defined by Matheron (1963) as half the average squared difference between a function and a translated copy of the function separated at distance

h

.[2][3] Formally

$$\gamma(h)=12\mathbb{V}[f(M+h)-f(M)]2dM,$$

where

M

is a point in the geometric field

V

, and

$f(M)$

 is the value at that point. The triple integral is over 3 dimensions.

h

 is the separation distance (e.g., in meters or km) of interest. For example, the value

f(M)

 could represent the iron content in soil, at some location

M

 (with geographic coordinates of latitude, longitude, and elevation) over some region

V

 with element of volume

dV

. To obtain the semivariogram for a given

γ(h)

, all pairs of points at that exact distance would be sampled. In practice it is impossible to sample everywhere, so the empirical variogram is used instead.

The variogram is twice the semivariogram and can be defined, differently, as the variance of the difference between field values at two locations (

s1

 and

s2

, note change of notation from

M

 to

s

and

f

to

Z

) across realizations of the field (Cressie 1993):

$$2\gamma(s1,s2)=\operatorname{var}(Z(s1)-Z(s2))=E[((Z(s1)-Z(s2))-E[Z(s1)-Z(s2)])2].$$

If the spatial random field has constant mean

μ

, this is equivalent to the expectation for the squared increment of the values between locations

s1

and

s2

(Wackernagel 2003) (where

s1

and

s2

are points in space and possibly time):

$$2\gamma(s1,s2)=E[(Z(s1)-Z(s2))2].$$

In the case of a stationary process, the variogram and semivariogram can be represented as a function

$\gamma_s(h) = \gamma(0, 0+h)$

 of the difference

$h = s_2 - s_1$

 between locations only, by the following relation (Cressie 1993):

$$\gamma(s_1, s_2) = \gamma_s(s_2 - s_1).$$

If the process is furthermore isotropic, then the variogram and semivariogram can be represented by a function

$\gamma_i(h) := \gamma_s(he_1)$

 of the distance

$h = \|s_2 - s_1\|$

 only (Cressie 1993):

$$\gamma(s_1, s_2) = \gamma_i(h).$$

The indexes

i

 or

s

 are typically not written. The terms are used for all three forms of the function. Moreover, the term "variogram" is sometimes used to denote the semivariogram, and the symbol

γ

 is sometimes used for the variogram, which brings some confusion.[4]

Properties

According to (Cressie 1993, Chiles and Delfiner 1999, Wackernagel 2003) the theoretical variogram has the following properties:

- The semivariogram is nonnegative
- $\gamma(s1,s2) \geq 0$
- , since it is the expectation of a square.
- The semivariogram
- $\gamma(s1,s1) = \gamma i(0) = E((Z(s1) - Z(s1))2) = 0$
- at distance 0 is always 0, since
- $Z(s1) - Z(s1) = 0$
- .
- A function is a semivariogram if and only if it is a conditionally negative definite function, i.e. for all weights
- $w1,\ldots,wN$
- subject to
- $\sum i=1N wi = 0$
- and locations
- $s1,\ldots,sN$
- it holds:

$$\sum i=1N \sum j=1N wi\gamma(si,sj)wj \leq 0$$

which corresponds to the fact that the variance

$\mathrm{var}(X)$

of

$X = \sum i=1N wiZ(xi)$

is given by the negative of this double sum and must be nonnegative.[*disputed* – *discuss*]

- If the covariance function C of a stationary process exists, it is related to variogram by

$$2\gamma(s1,s2)=C(s1,s1)+C(s2,s2)-2C(s1,s2)$$

- If the variance V and correlation function c of a stationary process exist, they are related to semivariogram by

$$\gamma(s1,s2)=V(1-c(s1,s2))$$

- Conversely, the covariance function C of a stationary process can be obtained from the semivariogram and variance as

$$C(s1,s2)=V-\gamma(s1,s2)$$

- If a stationary random field has no spatial dependence (i.e.
- $C(h)=0$
- $h\neq 0$
-), the semivariogram is the constant
- $\mathrm{var}(Z(s))$
- everywhere except at the origin, where it is zero.
- The semivariogram is a symmetric function,
- $\gamma(s1,s2)=E[|Z(s1)-Z(s2)|2]=\gamma(s2,s1)$
- .
- Consequently, the isotropic semivariogram is an even function
- $\gamma s(h)=\gamma s(-h)$
- .
- If the random field is stationary and ergodic, the
- $\lim_{h\to\infty}\gamma s(h)=\mathrm{var}(Z(s))$
- corresponds to the variance of the field. The limit of the semivariogram with increasing distance is also called its *sill*.
- As a consequence the semivariogram might be non continuous only at the origin. The height of the jump at the origin is sometimes referred to as *nugget* or nugget effect.

Parameters

In summary, the following parameters are often used to describe variograms:

- *nugget*
- n
- : The height of the jump of the semivariogram at the discontinuity at the origin.
- *sill*
- s
- : Limit of the variogram tending to infinity lag distances.
- *range*
- r
- : The distance in which the difference of the variogram from the sill becomes negligible. In models with a fixed sill, it is the distance at which this is first reached; for models with an asymptotic sill, it is conventionally taken to be the distance when the semivariance first reaches 95% of the sill.

Empirical variogram

Generally, an empirical variogram is needed for measured data, because sample information

Z

is not available for every location. The sample information for example could be concentration of iron in soil samples, or pixel intensity on a camera. Each piece of sample information has coordinates

s=(x,y)

for a 2D sample space where

x

and

y

are geographical coordinates. In the case of the iron in soil, the sample space could be 3 dimensional. If there is temporal variability as well (e.g., phosphorus content in a lake) then

s

 could be a 4 dimensional vector

(x,y,z,t)

. For the case where dimensions have different units (e.g., distance and time) then a scaling factor

B

 can be applied to each to obtain a modified Euclidean distance.[5]

Sample observations are denoted

Z(si)=zi

. Observations may be taken at

M

 total different locations (the sample size). This would provide as set of observations

z1,…,zM

 at locations

s1,…,sM

. Generally, plots show the semivariogram values as a function of separation distance

hk

 for multiple steps

k=1,…

. In the case of empirical semivariogram, separation distance interval

hk±δ

 is used rather than exact distances, and usually isotropic conditions are assumed (i.e., that

γ

is only a function of

h

and does not depend on other variables such as center position). Then, the empirical semivariogram

γ^(h±δ)

can be calculated for each bin:

$$\hat{\gamma}(h_k \pm \delta) := \frac{1}{2N_k} \sum_{(i,j) \in S_k} |z_i - z_j|^2$$

Or in other words, each pair of points separated by

hk

(plus or minus some bin width tolerance range

δ

) are found. These form the set of points

$$S_k = S(h_k \pm \delta) \equiv \{(s_i, s_j) : h_k - \delta < |s_i - s_j| < h_k + \delta; i, j = 1, \ldots, M\}$$

The number of these points in this bin is

Nk=|Sk|

(the set size). Then for each pair of points

i,j

, the square of the difference in the observation (e.g., soil sample content or pixel intensity) is found (

|zi−zj|2

). These squared differences are added together and normalized by the natural number

Nk

. By definition the result is divided by 2 for the semivariogram at this separation.

For computational speed, only the unique pairs of points are needed. For example, for 2 observations pairs [

(za,zb),(zc,zd)

] taken from locations with separation

h±δ

only [

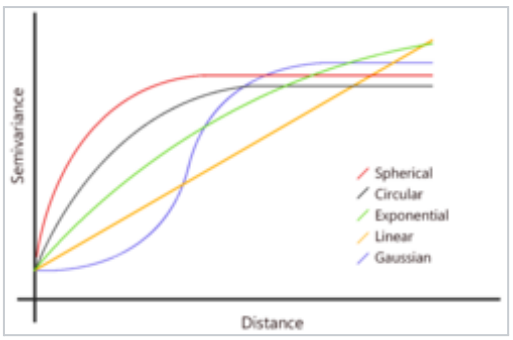
(za,zb),(zc,zd)

] need to be considered, as the pairs [

(zb,za),(zd,zc)

] do not provide any additional information.

Variogram models



Typical semivariogram functions in kriging.[6]

The empirical variogram cannot be computed at every lag distance

h

and due to variation in the estimation it is not ensured that it is a valid variogram, as defined above. However some geostatistical methods such as kriging need valid semivariograms. In applied geostatistics the empirical variograms are thus often approximated by model function ensuring validity. Some important models are:[7][8]

- The exponential variogram model
- $\gamma(h)=(s-n)(1-\exp(-h/(ra)))+n1(0,\infty)(h)$.
- The spherical variogram model
- $\gamma(h)=(s-n)((3h2r-h32r3)1(0,r)(h)+1[r,\infty)(h))+n1(0,\infty)(h)$.
- The Gaussian variogram model
- $\gamma(h)=(s-n)(1-\exp(-h2r2a))+n1(0,\infty)(h)$.

The parameter

a

has different values in different references, due to the ambiguity in the definition of the range (e.g.

$a=1/3$

).[8] The indicator function

$1A(h)$

is 1 if

$h\in A$

and 0 otherwise.

Applications

The empirical variogram is used in geostatistics as a first estimate of the variogram model needed for spatial interpolation by kriging.

- Empirical variograms for the spatiotemporal variability of column-averaged carbon dioxide was used to determine coincidence criteria for satellite and ground-based measurements.[5]
- Empirical variograms were calculated for the density of a heterogeneous material (Gilsocarbon).[9]
- Empirical variograms are calculated from observations of strong ground motion from earthquakes.[10] These models are used for seismic risk and loss assessments of spatially-distributed infrastructure.[11]

Related concepts

The squared term in the variogram, for instance

$$(Z(s1)-Z(s2))2$$

, can be replaced with different powers: A *madogram* is defined with the absolute difference,

$$|Z(s1)-Z(s2)|$$

, and a *rodogram* is defined with the square root of the absolute difference,

$$|Z(s1)-Z(s2)|0.5$$

. Estimators based on these lower powers are said to be more resistant to outliers. They can be generalized as a "variogram of order α",

$$2\gamma(s1,s2)=E[|Z(s1)-Z(s2)|\alpha]$$

,

in which a variogram is of order 2, a madogram is a variogram of order 1, and a rodogram is a variogram of order 0.5.[12]

When a variogram is used to describe the correlation of different variables it is called *cross-variogram*. Cross-variograms are used in co-kriging. Should the variable be binary or represent classes of values, one is then talking about *indicator variograms*. Indicator variograms are used in indicator kriging.

The following example samples a sea surface temperature (SST) image at random locations, then interpolates SST from the sample using Kriging:

```
# Load sea surface temperature (SST) image
sst = (
    ee.Image('NOAA/AVHRR_Pathfinder_V52_L3/20120802025048')
    .select('sea_surface_temperature')
    .rename('sst')
    .divide(100)
)

# Define geometry for sampling
geometry = ee.Geometry.Rectangle([-65.60, 31.75, -52.18, 43.12])

# Sample SST at random locations
samples = (
    sst
    .addBands(ee.Image.pixelLonLat())
    .sample(region=geometry, numPixels=1000)
    .map(
        lambda sample: ee.Feature(
            ee.Geometry.Point([
                sample.get('longitude'),
                sample.get('latitude')
```

```
            ]),
            {'sst': sample.get('sst')}
        )
    )
)

# Kriging interpolation
interpolated = samples.kriging(
    propertyName='sst',
    shape='exponential',
    range=100 * 1000,
    sill=1.0,
    nugget=0.1,
    maxDistance=100 * 1000,
    reducer='mean'
)

# Visualization parameters
colors = [
    '00007F', '0000FF', '0074FF',
    '0DFFEA', '8CFF41', 'FFDD00',
    'FF3700', 'C30000', '790000'
]
```

```
vis = {'min': -3, 'max': 40, 'palette': colors}


# Display layers
Map.setCenter(-60.029, 36.457, 5)
Map.addLayer(interpolated, vis, 'Interpolated')
Map.addLayer(sst, vis, 'Raw SST')
Map.addLayer(samples, {}, 'Samples', False)
```

The size of the neighborhood in which to perform the interpolation is specified by the `maxDistance` parameter. Larger sizes will result in smoother output but slower computations.

Reducer Overview

Reducers are the way to aggregate data over time, space, bands, arrays and other data structures in Earth Engine. The `ee.Reducer` class specifies how data is aggregated. The reducers in this class can specify a simple statistic to use for the aggregation (e.g. minimum, maximum, mean, median, standard deviation, etc.), or a more complex summary of the input data (e.g. histogram, linear regression, list). Reductions may occur over time (`imageCollection.reduce()`), space (`image.reduceRegion()`, `image.reduceNeighborhood()`), bands (`image.reduce()`), or the attribute space of a `FeatureCollection` (`featureCollection.reduceColumns()` or `FeatureCollection` methods that start with `aggregate_`).

Reducers have inputs and outputs

Reducers take an input dataset and produce a single output. When a single input reducer is applied to a multi-band image, Earth Engine automatically replicates the reducer and applies it separately to each band. As a result, the output image has the

same number of bands as the input image; each band in the output is the reduction of pixels from the corresponding band in the input data. Some reducers take tuples of input datasets. These reducers will not be automatically replicated for each band. For example, `ee.Reducer.LinearRegression()` takes multiple predictor datasets (representing independent variables in the regression) in a particular order (see Regression reducers).

Some reducers produce multiple outputs, for example `ee.Reducer.minMax()`, `ee.Reducer.histogram()` or `ee.Reducer.toList()`. For example:

```
# Load and filter the Sentinel-2 image collection.
collection = (
    ee.ImageCollection('COPERNICUS/S2_HARMONIZED')
    .filterDate('2016-01-01', '2016-12-31')
    .filterBounds(ee.Geometry.Point([-81.31, 29.90]))
)

# Reduce the collection.
extrema = collection.reduce(ee.Reducer.minMax())
```

This will produce an output with twice the number of bands of the inputs, where band names in the output have '_min' or '_max' appended to the band name.

The output type should match the computation. For example, a reducer applied to an `ImageCollection` has an `Image` output. Because the output is interpreted as a pixel value, you must use reducers with a numeric output to reduce an `ImageCollection` (reducers like `toList()` or `histogram()` won't work).

Reducers use weighted inputs

By default, reductions over pixel values are weighted by their mask, though this behavior can be changed (see the Weighting section). Pixels with mask equal to 0 will not be used in the reduction.

Combining reducers

If your intent is to apply multiple reducers to the same inputs, it's good practice to `combine()` the reducers for efficiency. Specifically, calling `combine()` on a reducer with `sharedInputs` set to `true` will result in only a single pass over the data. For example, to compute the mean and standard deviation of pixels in an image, you could use something like this:

```
# Load a Landsat 8 image.

image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318')


# Combine the mean and standard deviation reducers.

reducers = ee.Reducer.mean().combine(

    reducer2=ee.Reducer.stdDev(), sharedInputs=True

)


# Use the combined reducer to get the mean and SD of the image.

stats = image.reduceRegion(reducer=reducers, bestEffort=True)


# Display the dictionary of band means and SDs.

display(stats)
```

ImageCollection Reductions

Consider the example of needing to take the median over a time series of images represented by an `ImageCollection`. To reduce an `ImageCollection`, use `imageCollection.reduce()`. This reduces the collection of images to an individual image as illustrated in Figure 1. Specifically, the output is computed pixel-wise, such that each pixel in the output is composed of the median value of all the images in the collection at that location. To get other statistics, such as mean, sum, variance, an arbitrary percentile, etc., the appropriate reducer should be selected and applied. (See the **Docs** tab in the Code Editor for a list of all the reducers currently available). For basic statistics like min, max, mean, etc., `ImageCollection` has shortcut methods like `min()`, `max()`, `mean()`, etc. They function in exactly the same way as calling `reduce()`, except the resultant band names will not have the name of the reducer appended.

For an example of reducing an `ImageCollection`, consider a collection of Landsat 5 images, filtered by path and row. The following code uses `reduce()` to reduce the collection to one `Image` (here a median reducer is used simply for illustrative purposes):

```
# Load an image collection, filtered so it's not too much data.
collection = (
    ee.ImageCollection('LANDSAT/LT05/C02/T1')
    .filterDate('2008-01-01', '2008-12-31')
    .filter(ee.Filter.eq('WRS_PATH', 44))
    .filter(ee.Filter.eq('WRS_ROW', 34))
)

# Compute the median in each band, each pixel.
# Band names are B1_median, B2_median, etc.
median = collection.reduce(ee.Reducer.median())
```

```
# The output is an Image.  Add it to the map.

vis_param = {'bands': ['B4_median', 'B3_median', 'B2_median'], 'gamma':
1.6}

m = geemap.Map()

m.set_center(-122.3355, 37.7924, 9)

m.add_layer(median, vis_param)

m
```

This returns a multi-band `Image`, each pixel of which is the median of all unmasked pixels in the `ImageCollection` at that pixel location. Specifically, the reducer has been repeated for each band of the input imagery, meaning that the median is computed independently in each band. Note that the band names have the name of the reducer appended: `'B1_median'`, `'B2_median'`, etc. The output should look something like Figure 2.

For more information about reducing image collections, see the reducing section of the `ImageCollection` docs. In particular, note that images produced by reducing an `ImageCollection` have no projection. This means that you should explicitly set the scale on any computations involving computed images output by an `ImageCollection` reduction

Image Reductions

To reduce an `Image`, use `image.reduce()`. Reducing an image functions in an analogous way to `imageCollection.reduce()`, except the bands of the image are input to the reducer rather than the images in the collection. The output is also an image with number of bands equal to number of reducer outputs. For example:

```
# Load an image and select some bands of interest.

image = ee.Image('LANDSAT/LC08/C02/T1/LC08_044034_20140318').select(

    ['B4', 'B3', 'B2']

)
```

```
# Reduce the image to get a one-band maximum value image.
max_value = image.reduce(ee.Reducer.max())


# Display the result.
m = geemap.Map()
m.center_object(image, 10)
m.add_layer(max_value, {'max': 13000}, 'Maximum value image')
m
```