Advanced Raster Visualization (F6.0)

Authors

Gennadii Donchyts, Fedor Baart

Overview

This chapter should help users of Earth Engine to better understand raster data by applying visualization algorithms such as hillshading, hill shadows, and custom colormaps. We will also learn how image collection datasets can be explored by animating them as well as by annotating with text labels, using, for example, attributes of images or values queried from images.

Learning Outcomes

- Understanding why perceptually uniform colormaps are better to present data and using them efficiently for raster visualization.
- Using palettes with images before and after remapping values.
- Adding text annotations when visualizing images or features.
- Animating image collections in multiple ways (animated GIFs, exporting video clips, interactive animations with UI controls).
- Adding hillshading and shadows to help visualize raster datasets.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part F1).
- Write a function and `map` it over an `ImageCollection` (Chap. F4.0).
- Inspect an `Image` and an `ImageCollection`, as well as their properties (Chap. F4.1).

Introduction to Theory

Visualization is the step to transform data into a visual representation. You make a visualization as soon as you add your first layer to your map in Google Earth Engine. Sometimes you just want to have a first look at a dataset during the exploration phase. But as you move towards the dissemination phase, where you want to spread your

results, it is good to think about a more structured approach to visualization. A typical workflow for creating visualization consists of the following steps:

- Defining the story (what is the message?)
- Finding inspiration (for example by making a moodboard)
- Choosing a canvas/medium (here, this is the Earth Engine map canvas)
- Choosing datasets (co-visualized or combined using derived indicators)
- Data preparation (interpolating in time and space, filtering/mapping/reducing)
- Converting data into visual elements (shape and color)
- Adding annotations and interactivity (labels, scales, legend, zoom, time slider)

A good standard work on all the choices that one can make while creating a visualization is provided by the Grammar of Graphics (GoG) by Wilkinson (1999). It was the inspiration behind many modern visualization libraries (ggplot, vega). The main concept is that you can subdivide your visualization into several aspects.
In this chapter, we will cover several aspects mentioned in the Grammar of Graphics to convert (raster) data into visual elements. The accurate representation of data is essential in science communication. However, color maps that visually distort data through uneven color gradients or are unreadable to those with color-vision deficiency remain prevalent in science (Crameri, 2020). You will also learn how to add annotation text and symbology, while improving your visualizations by mixing images with hillshading as you explore some of the amazing datasets that have been collected in recent years in Earth Engine.

Practicum

Section 1. Palettes

If you have not already done so, you can add the book's code repository to the Code Editor by entering https://code.earthengine.google.com/?accept_repo=projects/gee-edu/book (or the short URL bit.ly/EEFA-repo) into your browser. The book's scripts will then be available in the script manager panel to view, run, or modify. If you have trouble finding the repo, you can visit bit.ly/EEFA-repo-help for help.

In this section we will explore examples of colormaps to visualize raster data. Colormaps translate values to colors for display on a map. This requires a set of colors (referred to as a "palette" in Earth Engine) and a range of values to map (specified by the min and max values in the visualization parameters).

There are multiple types of colormaps, each used for a different purpose. These include the following:

Sequential: These are probably the most commonly used colormaps, and are useful for ordinal, interval, and ratio data. Also referred to as a linear colormap, a sequential colormap looks like the *viridis* colormap (Fig. F6.0.1) from *matplotlib*. It is popular because it is a perceptual uniform colormap, where an equal interval in values is mapped to an equal interval in the perceptual colorspace. If you have a ratio variable where zero means nothing, you can use a sequential colormap starting at white, transparent, or, when you have a black background, at black—for example, the *turku* colormap from *Crameri* (Fig. F6.0.1). You can use this for variables like population count or gross domestic product.

Diverging: This type of colormap is used for visualizing data where you have positive and negative values and where zero has a meaning. Later in this tutorial, we will use the *balance* colormap from the *cmocean* package (Fig. F6.0.1) to show temperature change.

Circular: Some variables are periodic, returning to the same value after a period of time. For example, the season, angle, and time of day are typically represented as circular variables. For variables like this, a circular colormap is designed to represent the first and last values with the same color. An example is the circular *cet-c2* colormap (Fig. F6.0.1) from the colorcet package.

Semantic: Some colormaps do not map to arbitrary colors but choose colors that provide meaning. We refer to these as *semantic* colormaps. Later in this tutorial, we will use the ice colormap (Fig. F6.0.1) from the *cmocean* package for our ice example.

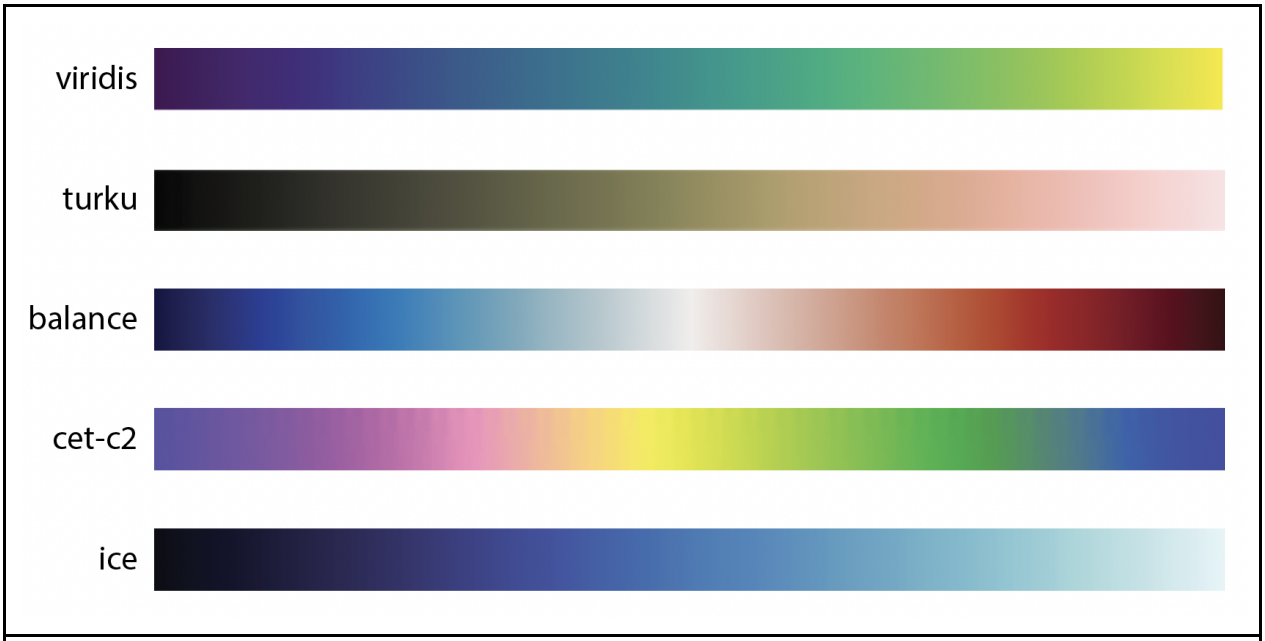

Fig. F6.0.1 Examples of colormaps from a variety of packages: *viridis* from *matplotlib*, *turku* from *Crameri*, *balance* from *cmocean*, *cet-c2* from colorcet and *ice* from *cmocean*

Popular sources of colormaps include:

- cmocean (semantic perceptual uniform colormaps for geophysical applications)
- colorcet (set of perceptual colormaps with varying colors and saturation)
- cpt-city (comprehensive overview of colormaps,
- colorbrewer (colormaps with variety of colors)
- Crameri (stylish colormaps for dark and light themes)

Our first example in this section applies a *diverging* colormap to temperature.

```
// Load the ERA5 reanalysis monthly means.
var era5 = ee.ImageCollection('ECMWF/ERA5_LAND/MONTHLY');

// Load the palettes package.
var palettes = require('users/gena/packages:palettes');

// Select temperature near ground.
era5 = era5.select('temperature_2m');
```

Now we can visualize the data. Here we have a temperature difference. That means that zero has a special meaning. By using a divergent colormap we can give zero the color white, which denotes that there is no significant difference. Here we will use the

colormap `Balance` from the `cmocean` package. The color red is associated with warmth, and the color blue is associated with cold. We will choose the minimum and maximum values for the palette to be symmetric around zero (-2, 2) so that white appears in the correct place. For comparison we also visualize the data with a simple `['blue', 'white', 'red']` palette. As you can see (Fig. F6.0.2), the `Balance` colormap has a more elegant and professional feel to it, because it uses a perceptual uniform palette and both saturation and value.

```
// Choose a diverging colormap for anomalies.
var balancePalette = palettes.cmocean.Balance[7];
var threeColorPalette = ['blue', 'white', 'red'];

// Show the palette in the Inspector window.
palettes.showPalette('temperature anomaly', balancePalette);
palettes.showPalette('temperature anomaly', threeColorPalette);

// Select 2 time windows of 10 years.
var era5_1980 = era5.filterDate('1981-01-01', '1991-01-01').mean();
var era5_2010 = era5.filterDate('2011-01-01', '2020-01-01').mean();

// Compute the temperature change.
var era5_diff = era5_2010.subtract(era5_1980);

// Show it on the map.
Map.addLayer(era5_diff, {
    palette: threeColorPalette,
    min: -2,
    max: 2
}, 'Blue White Red palette');

Map.addLayer(era5_diff, {
    palette: balancePalette,
    min: -2,
    max: 2
}, 'Balance palette');
```

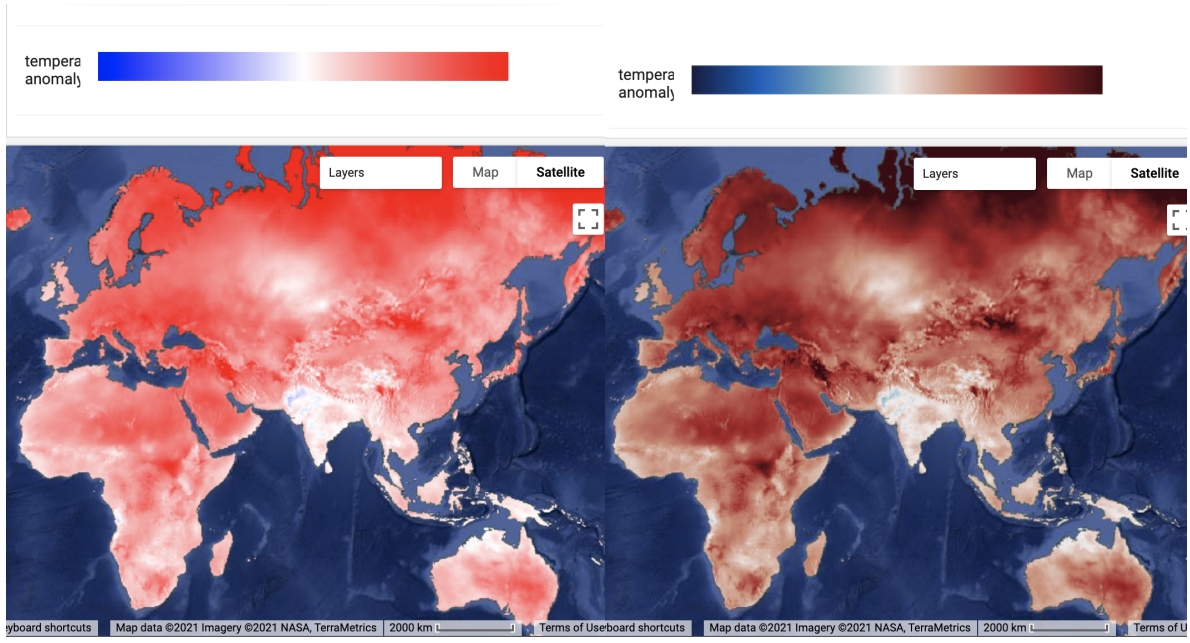

Fig. F6.0.2 Temperature difference of ERA5 (2011–2020, 1981–1990) using the *balance* colormap from *cmocean* (right) versus a basic *blue-white-red* colormap (left)

Code Checkpoint F60a. The book's repository contains a script that shows what your code should look like at this point.

Our second example in this section focuses on visualizing a region of the Antarctic, the Thwaites Glacier. This is one of the fast-flowing glaciers that causes concern because it loses so much mass that it causes the sea level to rise. If we want to visualize this region, we have a challenge. The Antarctic region is in the dark for four to five months each winter. That means that we can't use optical images to see the ice flowing into the sea. We therefore will use radar images. Here we will use a **semantic** colormap to denote the meaning of the radar images.

Let's start by importing the dataset of radar images. We will use the images from the Sentinel-1 constellation of the Copernicus program. This satellite uses a C-band synthetic-aperture radar and has near-polar coverage. The radar senses images using a polarity for the sender and receiver. The collection has images of four different possible combinations of sender/receiver polarity pairs. The image that we'll use has a band of the Horizontal/Horizontal polarity (HH).

```
// An image of the Thwaites glacier.
var imageId =
```

```
'COPERNICUS/S1_GRD/S1B_EW_GRDM_1SSH_20211216T041925_20211216T042029_030045_03965B_AF0A';

// Look it up and select the HH band.
var img = ee.Image(imageId).select('HH');
```

For the next step, we will use the palette library. We will stylize the radar images to look like optical images, so that viewers can contrast ice and sea ice from water (Lhermitte, 2020). We will use the `Ice` colormap from the `cmocean` package (Thyng, 2016).

```
// Use the palette library.
var palettes = require('users/gena/packages:palettes');

// Access the ice palette.
var icePalette = palettes.cmocean.Ice[7];

// Show it in the console.
palettes.showPalette('Ice', icePalette);

// Use  it to visualize the radar data.
Map.addLayer(img, {
    palette: icePalette,
    min: -15,
    max: 1
}, 'Sentinel-1 radar');

// Zoom to the grounding line of the Thwaites Glacier.
Map.centerObject(ee.Geometry.Point([-105.45882094907664, -
    74.90419580705336
]), 8);
```

If you zoom in (F6.0.3) you can see how long cracks have recently appeared near the pinning point (a peak in the bathymetry that functions as a buttress, see Wild, 2022) of the glacier.

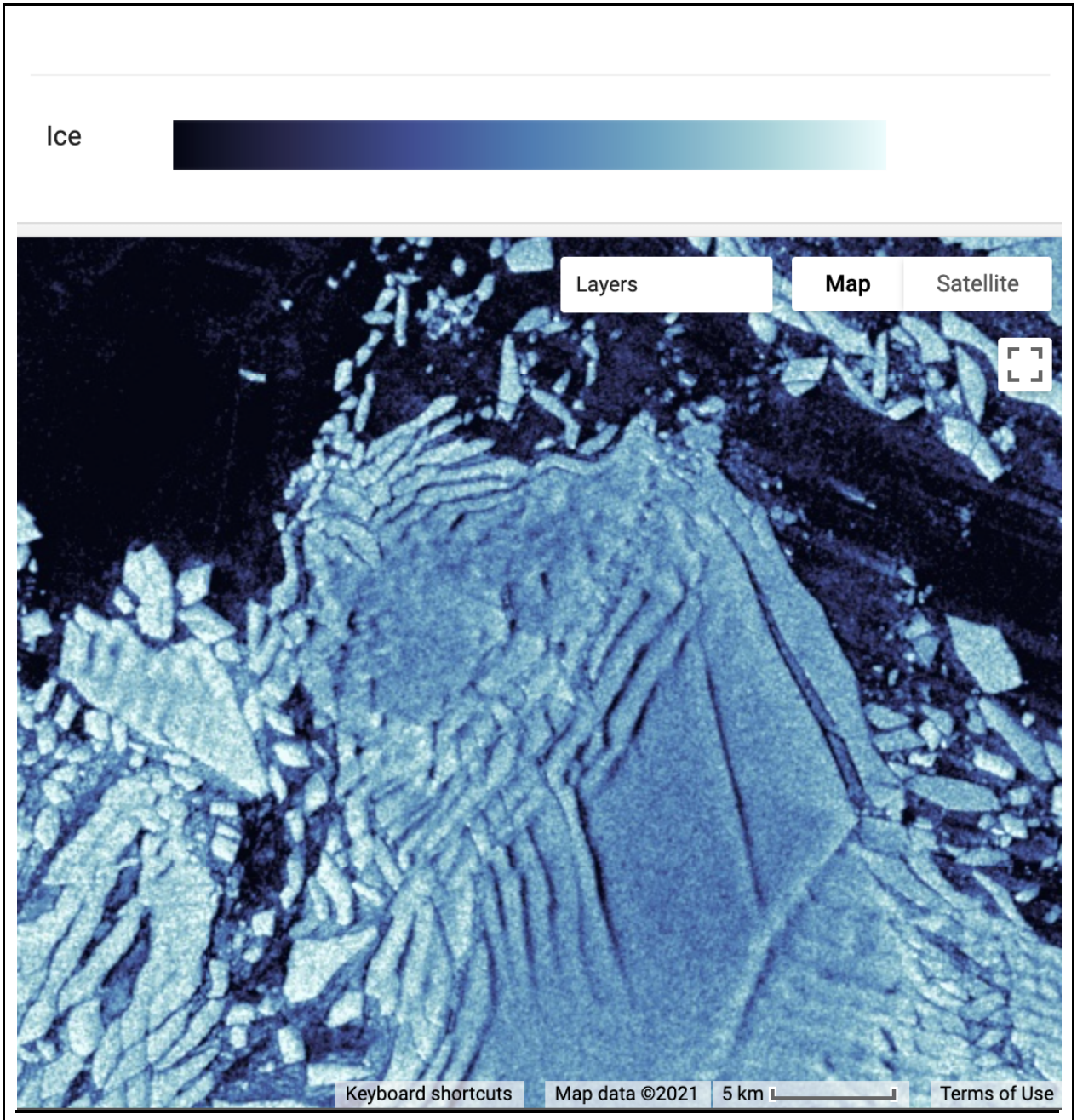

Fig. F6.0.3. Ice observed in Antarctica by the Sentinel-1 satellite. The image is rendered using the *ice* color palette stretched to backscatter amplitude values [-15; 1].

Code Checkpoint F60b. The book's repository contains a script that shows what your code should look like at this point.

Section 2. Remapping and Palettes

Classified rasters in Earth Engine have metadata attached that can help with analysis and visualization. This includes lists of the names, values, and colors associated with class. These are used as the default color palette for drawing a classification, as seen next. The USGS National Land Cover Database (NLCD) is one such example. Let's access the NLCD dataset, name it `nlcd`, and view it (Fig. F6.0.4) with its built-in palette.

```
// Advanced remapping using NLCD.
// Import NLCD.
var nlcd = ee.ImageCollection('USGS/NLCD_RELEASES/2016_REL');

// Use Filter to select the 2016 dataset.
var nlcd2016 = nlcd.filter(ee.Filter.eq('system:index', '2016'))
    .first();

// Select the land cover band.
var landcover = nlcd2016.select('landcover');

// Map the NLCD land cover.
Map.addLayer(landcover, null, 'NLCD Landcover');
```

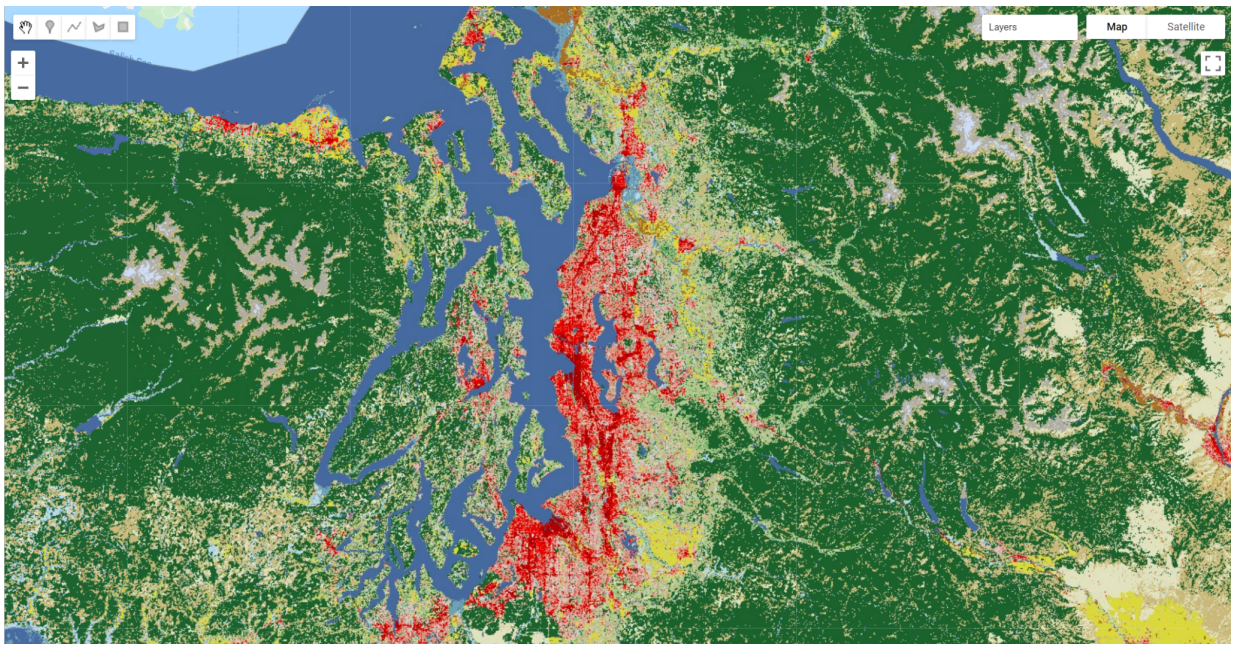

Fig. F6.0.4 The NLCD visualized with default colors for each class

But suppose you want to change the display palette. For example, you might want to have multiple classes displayed using the same color, or use different colors for some classes. Let's try having all three urban classes display as dark red (`'ab0000'`).

```
// Now suppose we want to change the color palette.
var newPalette = ['466b9f', 'd1def8', 'dec5c5',
    'ab0000', 'ab0000', 'ab0000',
    'b3ac9f', '68ab5f', '1c5f2c',
    'b5c58f', 'af963c', 'ccb879',
    'dfdfc2', 'd1d182', 'a3cc51',
    '82ba9e', 'dcd939', 'ab6c28',
    'b8d9eb', '6c9fb8'
];

// Try mapping with the new color palette.
Map.addLayer(landcover, {
    palette: newPalette
}, 'NLCD New Palette');
```

However, if you map this, you will see an unexpected result (Fig. F6.0.5).

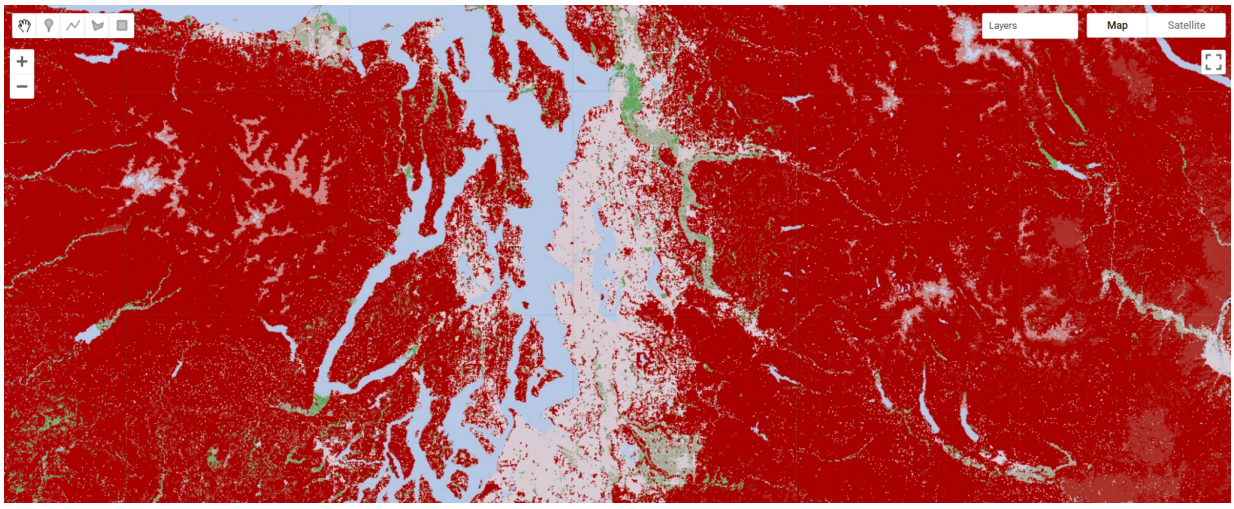

Fig. F6.0.5 Applying a new palette to a multi-class layer has some unexpected results

This is because the numeric codes for the different classes are not sequential. Thus, Earth Engine stretches the given palette across the whole range of values and produces an unexpected color palette. To fix this issue, we will create a new index for the class values so that they are sequential.

```
// Extract the class values and save them as a list.
var values = ee.List(landcover.get('landcover_class_values'));

// Print the class values to console.
print('raw class values', values);

// Determine the maximum index value
var maxIndex = values.size().subtract(1);

// Create a new index for the remap
var indexes = ee.List.sequence(0, maxIndex);

// Print the updated class values to console.
print('updated class values', indexes);

// Remap NLCD and display it in the map.
var colorized = landcover.remap(values, indexes)
    .visualize({
        min: 0,
        max: maxIndex,
        palette: newPalette
    });
Map.addLayer(colorized, {}, 'NLCD Remapped Colors');
```

Using this remapping approach, we can properly visualize the new color palette (Fig. F6.0.6).

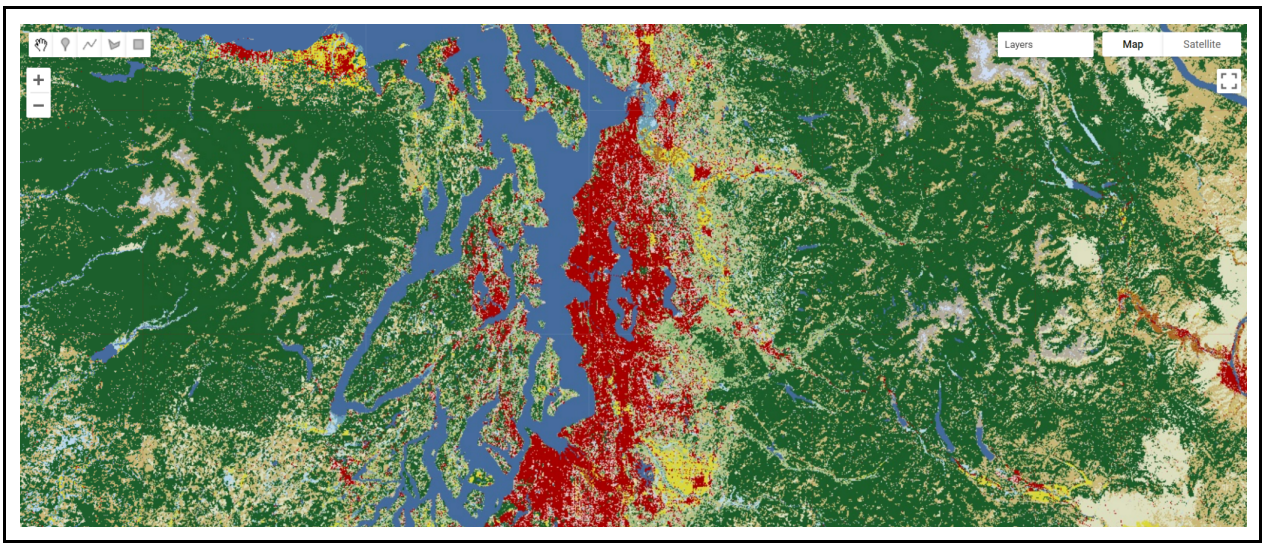

Fig. F6.0.6 Expected results of the new color palette. All urban areas are now correctly showing as dark red and the other land cover types remain their original color.

Code Checkpoint F60c. The book's repository contains a script that shows what your code should look like at this point.

Section 3. Annotations

Annotations are the way to visualize data on maps to provide additional information about raster values or any other data relevant to the context. In this case, this additional information is usually shown as geometries, text labels, diagrams, or other visual elements. Some annotations in Earth Engine can be added by making use of the `ui` portion of the Earth Engine API, resulting in graphical user interface elements such as labels or charts added on top of the map. However, it is frequently useful to render annotations as a part of images, such as by visualizing various image properties or to highlight specific areas.

In many cases, these annotations can be mixed with output images generated outside of Earth Engine, for example, by post-processing exported images using Python libraries or by annotating using GIS applications such as QGIS or ArcGIS. However, annotations could also be also very useful to highlight and/or label specific areas directly within the Code Editor. Earth Engine provides a sufficiently rich API to turn vector features and geometries into raster images which can serve as annotations. We recommend checking the `ee.FeatureCollection.style` function in the Earth Engine documentation to learn how geometries can be rendered.

For textual annotation, we will make use of an external package `'users/gena/packages:text'` that provides a way to render strings into raster images directly using the Earth Engine raster API. It is beyond the scope of the current tutorials to explain the implementation of this package, but internally this package makes use of bitmap fonts which are ingested into Earth Engine as raster assets and are used to turn every character of a provided string into image glyphs, which are then translated to desired coordinates.

The API of the *text* package includes the following mandatory and optional arguments:

```
/**
```

```
 * Draws a string as a raster image at a given point.
 *
 * @param {string} str - string to draw
 * @param {ee.Geometry} point - location the the string will be drawn
 * @param {{string, Object}} options - optional properties used to
style text
 *
 * The options dictionary may include one or more of the following:
 *     fontSize       - 16|18|24|32 - the size of the font (default:
16)
 *     fontType       - Arial|Consolas - the type of the font
(default: Arial)
 *     alignX         - left|center|right (default: left)
 *     alignY         - top|center|bottom (default: top)
 *     textColor      - text color string (default: ffffff - white)
 *     textOpacity    - 0-1, opacity of the text (default: 0.9)
 *     textWidth      - width of the text (default: 1)
 *     outlineColor   - text outline color string (default: 000000 -
black)
 *     outlineOpacity - 0-1, opacity of the text outline (default:
0.4)
 *     outlineWidth   - width of the text outlines (default: 0)
 */
```

To demonstrate how to use this API, let's render a simple `'Hello World!'` text string placed at the map center using default text parameters. The code for this will be:

```
// Include the text package.
var text = require('users/gena/packages:text');

// Configure map (change center and map type).
Map.setCenter(0, 0, 10);
Map.setOptions('HYBRID');

// Draw text string and add to map.
var pt = Map.getCenter();
var scale = Map.getScale();
var image = text.draw('Hello World!', pt, scale);
Map.addLayer(image);
```

Running the above script will generate a new image containing the `'Hello World!'` string placed in the map center. Notice that before calling the text.draw() function we configure the map to be centered at specific coordinates (0,0) and zoom level 10 because map parameters such as center and scale are passed as arguments to that text.draw() function. This ensures that the resulting image containing string characters is scaled properly.

When exporting images containing rendered text strings, it is important to use proper scale to avoid distorted text strings that are difficult to read, depending on the selected font size, as shown in Fig. F6.0.7.

Code Checkpoint F60d. The book's repository contains a script that shows what your code should look like at this point.

Fig. F6.0.7 Results of the `text.draw` call, scaled to 1x: `var scale = Map.getScale()*1;` (left), 2x: `var scale = Map.getScale()*2;` (center), and 0.5x: `var scale = Map.getScale()*0.5;` (right)

These artifacts can be avoided to some extent by specifying a larger font size (e.g., 32). However, it is better to render text at the native 1:1 scale to achieve best results. The same applies to the text color and outline: They may need to be adjusted to achieve the best result. Usually, text needs to be rendered using colors that have opposite brightness and colors when compared to the surrounding background. Notice that in the above example, the map was configured to have a dark background (`'HYBRID'`) to ensure that the white text (default color) would be visible. Multiple parameters listed in the above API documentation can be used to adjust text rendering. For example, let's switch font size, font type, text, and outline parameters to render the same string, as below. Replace the existing one-line `text.draw` call in your script with the following code, and then run it again to see the difference (Fig. F6.0.8):

```
var image = text.draw('Hello World!', pt, scale, {
    fontSize: 32,
    fontType: 'Consolas',
    textColor: 'black',
    outlineColor: 'white',
    outlineWidth: 1,
    outlineOpacity: 0.8
});

// Add the text image to the map.
Map.addLayer(image);
```

Code Checkpoint F60e. The book's repository contains a script that shows what your code should look like at this point.

Fig. F6.0.8 Rendering text with adjusted parameters (font type: Consolas, fontSize: 32, textColor: 'black', outlineWidth: 1, outlineColor: 'white', outlineOpacity: 0.8)

Of course, non-optional parameters such as `pt` and `scale`, as well as the text string, do not have to be hard-coded in the script; instead, they can be acquired by the code using, for example, properties coming from a `FeatureCollection`. Let's demonstrate this by showing the cloudiness of Landsat 8 images as text labels rendered in the center of every image. In addition to annotating every image with a cloudiness text string, we will also draw yellow outlines to indicate image boundaries. For convenience, we can also